Muddle Documentation

Release 2.5

Richard Watts

2016-07-18

Contents

Contents:

CHAPTER 1

Introduction to Muddle

Muddle is a build system for systems. It specialises in taking a set of packages and welding them together into a firmware image for an embedded system. The assembly of these packages is directed by an executable build description written in Python.

Muddle was designed specifically for use in producing an embedded system from scratch. It is aimed at people who start with a new board and need to assemble the operating system and user space software for it. It does not provide pre-packaged solutions in the manner of Yocto or open-embedded.

Among other things, muddle supports:

- Specifying the dependencies of packages, and thus the order of build and rebuild.
- Building the Linux kernel, and generating CPIO initrds from source. Note that this does not require sudo at any stage.
- Building separate targets from the same packages, sharing binaries and object files as necessary.
- Interacting with version control systems (currently, svn, bazaar and git, though adding another is no more than a couple of day's work)
- Producing binary distributions.
- Producing source code releases. Packages can be marked with their license-type, allowing automated generation of GPL-compliant releases. Muddle makes it easier to make such build subsets buildable, rather than just providing a source dump.
- Managing maintenance release branches of all packages in the build tree (still under development)

Muddle is used internally at Kynesim, as well as by other people. If you have problems with it, or ideas on how it could be improved, please do raise an issue, or contact us directly.

The muddle source code lives at https://github.com/kynesim/muddle.

Documentation lives at http://muddle.readthedocs.org/, thanks to ReadTheDocs, who are wonderful people. The documentation should be updated every time a change is pushed to the main source code repository.

The Kynesim blog also periodically has articles on using muddle.

Traditionally, muddle is also licenced to kill and to serve drinks after hours.

CHAPTER 2

Quick start: so you've been asked to work on a project using muddle

This is intended as a short introduction to the absolute minimum you're likely to need.

2.1 Step 1: getting muddle

The very first thing you need is muddle itself.

Getting muddle needs git. If you don't have git on your system, and you're on a Debian based system (Debian, Ubuntu, Linux Mint, etc.), then you can do:

```
$ sudo apt-get install git gitk
```

(the `gitk` program is an invaluable UI for looking at the state of git checkouts - it's always worth checking it out as well as git itself).

Then decide where to put muddle. I have a `sw` directory for useful software checkouts, so I would do:

```
$ cd sw
$ git clone https://code.google.com/p/muddle/
```

which creates me a directory `~/sw/muddle`.

Note: Sometimes (luckily not often) the Google code repositories give errors. If you can't clone muddle from code.google.com, try using the mirror on github, which should normally be up-to-date:

```
git clone https://github.com/tibs/muddle.mirror.git muddle
```

To *use* muddle, you can then either:

1. just type `~/sw/muddle/muddle` - this is the simplest thing to do, but the longest to type.
2. add an alias to your `.bashrc` or equivalent:

```
   alias muddle="${HOME}/sw/muddle/muddle"
```

3. add `~/sw/muddle` to your PATH:

```
   export PATH=${PATH}:${HOME}/sw/muddle
```

4. add a link - for instance, if you have `~/bin` on your path, do:

```
    cd ~/bin
    ln -s ~/sw/muddle/muddle .
```

Personally, I use the second option, but all are sensible.

You should now be able to do:

```
$ muddle help
```

and get meaningful output. To find help on a particular muddle command, you can, for instance, do:

```
$ muddle help checkout
```

Note: Some people also like to keep muddle in `/opt` - for instance, as `/opt/muddle`.

Note: Muddle is actively developed, and we normally assume that you will keep it up-to-date with the current state of that development. Unless your project says to do otherwise, it's normally worth doing a `git pull` once a week or so – for instance:

```
$ pushd ~/sw/muddle
$ git pull
```

2.2 Step 2: getting your project's build description

You should have been given a `muddle init` line to use to bootstrap your project.

For simplicity in explaining things, I'm going to assume that you keep your current projects in a directory called `~/work`, and this new project is `project99`.

Muddle wants each project to live in its own directory, so we would do:

```
$ mkdir ~/work/project99
$ cd ~/work/project99
```

and then type the `muddle init` line, something of the form:

```
$ muddle init git+https://code.google.com/p/raw-cctv-replay builds/01.py
```

The first argument to `git init` says where the project build description is kept (for the RAW project, that's on Google code), and the second argument says where the build description can be found within the `src` directory.

Note: Please use the `muddle init` command for your own project, unless you want to work on RAW, which is a rather large project.

Note: Some builds check for particular software on the system, and use `sudo apt-get` to install things if necessary. The build documentation should always tell you about this.

After `muddle init` has succeeded, you should have two directories in your project directory:

- `.muddle` which contains the "state" of the muddle build, and

- `src`, which will typically contain `builds/01.py` (or whatever else was named by that final argument to `muddle init`.

Note: If the `muddle init` command goes wrong, then you will need to delete both directories before trying it again.

The build description is a Python file (for some projects, more than one) which describes the build to muddle. It is traditionally kept in the `src/builds` directory, and for historical reasons the main build description file is called `01.py`.

2.3 Step 3: building the project the first time

The simplest way of building the project for the first time is just to use the "muddle" command directly - it will realise that it needs to checkout all the source code and build it from scratch. So:

```
$ cd ~/sw/project99
$ muddle
```

Alternatively, some people prefer to separate the checkout and build phases, so that all the internet access is done in one go. This can be done with:

```
$ cd ~/sw/project99
$ muddle checkout _all
$ muddle
```

After this has finished, you should have a much fuller `src` directory, and also `obj`, `install` and `deploy` directories. If you're building software to put on an embedded system, that will generally be in one or more of the `deploy` directories, and your project documentation should tell you what to do next.

2.4 The muddle directories and its phases

Muddle keeps the build tree in several different directories.

- `.muddle` is where information about the build tree is kept, for instance which packages have been built.
- `src` is where the source code is kept, as checked out with `muddle checkout _all`.
- `obj` is where muddle builds things. A well-behaved muddle build always builds "out of tree" in `obj`, so that building things doesn't change anything in the `src` directory.
- `install` is where the muddle build puts its results, those programs, libraries and so on that are going to be used to make a deployment.
- `deploy` is where deployments are assembled. These are the parts of the final system that will be put onto the target hardware.

This means that muddle has three phases to its work:

1. Checkout, which populates the `src` directories.
2. Build, which builds in `obj` and puts the results into `install`. Building will also check things out into `src` if necessary.
3. Deploy, which puts together things from `install` and stores the results in `deploy`. Deployment will also build things if necessary.

Warning: Remember that building (phase 2) does not do deployment (phase 3). It is up to the user to decide when the packages have built correctly and are in a suitable state for deployment.

2.5 Step 4: getting the latest version of code and rebuilding

Eventually, some of the software in your project will be updated, and you will need to retrieve the new versions of the source code, and rebuild it.

This is typically done from the "top level" directory - i.e., the directory containing the `src` and `.muddle` directories. So, with our project:

```
$ cd ~/work/project99
$ muddle pull _all
```

Hopefully that will succeed without any problems. If there are any problems, they will be summarised at the end of the command output.

Once the source code is updated, you can then rebuild all packages by doing:

```
$ muddle distrebuild _all
```

or just rebuild everything affected by the updated sources:

```
$ muddle distrebuild _just_pulled
```

If you also want to redeploy - that is, rebuild the `deploy` directories - then you can:

1. Redeploy *all* deployments:

   ```
   $ muddle redeploy _all
   ```

2. Redeploy the default deployments, the same as the very first `muddle` command:

   ```
   $ muddle redeploy _default_deployments
   ```

3. Redeploy something specific:

   ```
   $ muddle redeploy <some-specific-deployment>
   ```

2.6 Step 5: editing code and rebuilding

If you need to edit code, then it may be worth your getting a deeper knowledge of how muddle works, and a good starting point is probably the "Muddle and its directories" chapter.

There is more than one way to use muddle - this very short introduction is only that.

Throughout this section, we assume that you are in a checkout source directory - for instance:

```
$ cd ~/work/project99
$ cd src/some_program-v1.9
```

2.6.1 Edit and rebuild

The normal edit/rebuild cycle is fairly simple - edit:

```
$ gvim sourcecode.c
```

and then rebuild:

```
$ muddle rebuild
```

Muddle knows what package uses this source checkout (if the checkout is `some_program-v1.9` then the package is probably `some_program`), and so it will rebuild the package that uses that source code.

Note: Inside `src/`, just typing `muddle` does a *build*, not a rebuild. Muddle cannot tell that you've changed the source code, so it still thinks that the last build of the code is valid, so you need to tell it explicitly to do a rebuild.

Doing a `rebuild` also tells muddle that any packages that depend on this one will also need rebuilding - this is important in the next section.

If you also need to reconfigure the package, then you need to "distclean" it first. So:

```
$ muddle distclean
$ muddle rebuild
```

There's a convenient command that does those together:

```
$ muddle distrebuild
```

2.6.2 Build anything else that needs it

Once you've finished the "edit, compile" loop, and are happy with the state of the package, there may be other things needing building.

You *could* just do `muddle rebuild _all`, but that would rebuild every single package. However, muddle knows which packages depend on the package you've just been working on, and so:

```
$ muddle build _all
```

should just build those packages that need it.

2.6.3 Commit your changes

Once the code for your checkout is correct, you can commit it using the appropairte version control commands (there is a `muddle commit`, but I tend just to use the VCS directly) - for instance:

```
$ git add sourcecode.c
$ git commit
```

and then push it back to the far repository:

```
$ muddle push
```

You could, of course, also use the approriate version control command directly, but `muddle push` keeps track of which remote repository the build description says should be used, which may or may not be the same as the repository that was originally pulled from.

2.6.4 Consider redeploying

Building does not redeploy. This is easy to forget, but deployment is deployment of a system for use on the target hardware, and it is not normally appropriate to do it every time some component of the system is rebuilt (not least because the overall system may not work again until more than one checkout/package is altered).

So it is up to the user to decide when to redeploy, at which time:

```
$ muddle redeploy
```

will redeploy the deployment containing the package that uses this source checkout (i.e., what should be the minimum necessary redeployment), whilst:

```
$ muddle redeploy _all
```

will redeploy everything, and:

```
$ muddle redeploy _default_deployments
```

will redeploy, well, the default deployments, just as the very first `muddle` command would have done.

2.7 Step 6: finding out more

If you want to progress to the next level of use of muddle, then reading the chapter "Muddle and its directories" is a useful next step. There are various "frequently asked questions" answered in the "Jottings" section as well.

CHAPTER 3

Welcome to Muddle

Note: This is Richard's original document about Muddle. As such, it is not intended as a gentle introduction. If you are after lighter fare, you may want to come back to this later on.

Muddle is a package-level build system. It aims to do for software and firmware distributions what Make does for individual software packages.

Muddle's design philosophy is the same as UNIX's: mechanism, not policy. As a result you'll discover that muddle will let you do nearly anything you like. You'll also discover that doing anything you like sometimes doesn't make you popular among your peers.

3.1 Labels

(Nearly) everything in muddle is described by a label. A label is structured as:

```
<type>:<name>{<role>}/<tag>[<flags>]
```

Any component of a label may be wildcarded with '*' and a role (only!) may be omitted. All components are made up of the characters [A-Z0-9a-z*-_] only. names beginning with an underscore are reserved by muddle.

Muddle does not constrain the values of labels - you may use any string for any component. However, certain labels are special:

- `<type>` is typically

`checkout`	Describes a unit of source control
`package`	Describes a unit of software build
`deployment`	Describes a unit of software installation

- `<tag>` is typically

`checked_out`	The state of a copy having been taken from revision control
`preconfig`	Preconfiguration checks have been made on a package
`configured`	A package has been configured
`built`	A package has been built
`installed`	A package has been installed
`postinstalled`	A package has been postinstalled
`deployed`	A deployment has been created

Muddle's central algorithms revolve around describing the relationships between components by linking labels in rules. Rules are similar to make rules:

```
target: dependencies
   <command>
```

Except that the target and dependency lists contain (possibly wildcarded) labels rather than simple strings.

3.2 Muddle's view of the build process

Muddle believes that a build process consists of the following steps:

1. Check a number of checkouts out of source control (and put them into `$MUDDLE_ROOT/src`)
2. Build a set of packages from them. Each package may be built in a number of roles - intended for different use cases, so you may have a `libc/arm` and a `libc/x86`, for example. (packages are built in `$MUDDLE_ROOT/obj/<pkg>/</role>`)
3. Install each role into a different installation directory. (packages are installed into `$MUDDLE_ROOT/install/<role>`)
4. Build these role installations into a series of objects suitable for installation on some target machine (deployments are installed into `$MUDDLE_ROOT/deploy`)

There is often a 1-1 association between checkouts and packages, roles and deployments, but muddle does not enforce this - and in fact, many to many associations between these objects can be used to implement common workflows - picking different elements of a single role to populate 'lite' and 'full' versions of firmware, or using the same checkout to build different architectures, for example.

3.3 Build Descriptions and the muddle database

To build your system, Muddle needs a machine-readable description of it and it needs some filespace in which to build. You provide filespace by initialising a muddle build tree in some convenient filesystem location. You provide the description of what to build with a build description.

A build description is a perfectly normal piece of python containing a subroutine:

```
def describe_to(builder)
```

which is called to load information about your software into an object called a 'builder' (see `muddled/mechanics.py`) which can then build your software.

Several examples are provided in the examples/ directory and utility APIs are provided to build packages controlled by make, to ensure that the host has certain Debian packages installed, to build a target filesystem with the right file permissions, etc.

The build description is stored in a checkout in revision control along with everything else in your system - muddle will automatically check it out when needed.

Apart from some bootstrapping rules, the build description checkout is a perfectly normal checkout and you can manipulate it with the same tools you would any other checkout.

It is strongly advised that you call your build description checkout 'builds'. Again, muddle will not enforce this - it's just good practice.

Note that your build descriptions are perfectly sensible python programs in their own right and are welcome to do almost anything they like - specifically, you are encouraged to use inheritance and imports to place the common parts of your build descriptions in a single file. This can greatly aid configuration management.

Muddle will track your build using a database stored in a `.muddle` directory. The structure of this database is well-known and you are actively encouraged to edit it by hand to resolve any problems you might have:

`.muddle/Description` Path from the src directory to the build description file for this build.

`.muddle/RootRepository` URL for the root of the repository from which to fetch checkouts (you can override this in your build description if you want to - it's just a useful default)

`.muddle/tags/..` If a label is asserted, there will be a file corresponding to it here. You can touch these files to synthetically assert labels or remove them to retract them.

`.muddle/instructions/<pkg>/</role>` Instruction files - these hold pending install directions for the deploy step to use.

3.4 Multiple Files and Inheritance in Build Descriptions

Muddle automatically adds the build description's checkout directory to sys.path before it imports your build description. This allows you to treat the build description directory as a python package from which you can import additional build description helper files at will - e.g. in builds/foo.py:

```
import bar

def describe_to(builder):
    bar.do_common_setup()
    ...
```

3.5 Environment variables, sudo and instructions

In order to successfully build your package, your makefile (or whatever) is going to need to know a few things. Among them are commonly:

- Where should I install my object files/binaries?
- Where will those directories be on the target system?
- How do I set permissions/change ownerships of my created directories?

Some of these can be answered by setting environment variables. Muddle sets a number of variables itself:

`MUDDLE_ROOT` The directory with the .muddle directory in it.

`MUDDLE_LABEL`

- `MUDDLE_KIND`
- `MUDDLE_NAME`
- `MUDDLE_ROLE`
- `MUDDLE_TAG`

The label we're currently building, and its components.

`MUDDLE_INSTALL` Where do we install package files to? (typically under `$MUDDLE_ROOT/install`)

`MUDDLE_DEPLOY_FROM` Where do we deploy from (typically `$MUDDLE_ROOT/install/<role>`)

`MUDDLE_DEPLOY_TO` Where do we deploy to? (typically `$MUDDLE_ROOT/deploy/<deployment>`)

MUDDLE How to call muddle itself

MUDDLE_INSTRUCT, MUDDLE_UNINSTRUCT Used by the instruction system - see below.

MUDDLE_INCLUDE_DIRS Space-separated list of include directories for this package and all its dependents. This is a slightly vexed issue, since depending on tools roles, for example, results in your builds for the target machine bringing in libraries for the host. As such, there is an exclude list of roles whose dependencies on each other don't affect MUDDLE_INCLUDE_DIRS or MUDDLE_LIB_DIRS: see builder.roles_do_not_share_libraries()

MUDDLE_LIB_DIRS As `MUDDLE_INCLUDE_DIRS` but with library directories.

MUDDLE_KERNEL_DIR If there was a `$(MUDDLE_OBJ)/kerneldir` directory, the last one of those. Used by the `linux_kernel` builder to point module builds at the right directory for invoking module builds.

MUDDLE_KERNEL_SOURCE_DIR If there was a `$(MUDDLE_OBJ)/kernelsourcedir` directory, the last one encountered - usually a symlink to the kernel source.

MUDDLE_PKGCONFIG_DIRS `$(MUDDLE_OBJ)/lib/pkgconfig` directories - for use as a `PKG_CONFIG_PATH`.

MUDDLE_OBJ Package object directory, whose subdirectories include:

- `MUDDLE_OBJ_OBJ`, where you put actual objects.
- `MUDDLE_OBJ_INCLUDE`, where you put include files to be picked up by other packages
- `MUDDLE_OBJ_LIB`, where you put library files to be picked up by other packages.

And the facility to associate environments with a (possibly wildcarded) label. This allows you to associate any extra environment variables you want from your build description to various labels - all packages, for example, or all roles.

Take a look at `muddled/env_store.py` and `muddled/mechanism.py` for details.

This doesn't tell you where your package will end up on the target system. It doesn't tell you because it doesn't know. You'll typically be required to give this information when you ask muddle to create a deployment, and most deployments and package builders define:

- `MUDDLE_TARGET_LOCATION`

  To tell you where the deployment will end up.

- `MUDDLE_SRC`

  The location of your checkout if one can be sensibly derived - the Make package builder, for example, has an N to 1 correspondence between packages and checkouts, so `$(MUDDLE_SRC)` is defined as the checkout from which this package is being built. The package builder which imports debian binaries doesn't, so `MUDDLE_SRC` is left undefined for it.

Even so, there are things that can't be done by your makefile.

Creation of `initrds` with proper filesystems and changes of ownership, for example, cannot (in general) be done by makefiles or package builds of other kinds because they are running as a mortal user and those operations require root privilege.

Traditionally, build systems have got around this by `sudo`-ing at random times and expecting you to either have passwordless `sudo` access (a security risk) or type in your password at irregular interfaces (which is just annoying).

Muddle uses things called instructions.

The idea of instructions is that during your makefile, you run a command like:

```
$(MUDDLE_INSTRUCT) instr-file.xml
```

where `instr-file.xml` contains a series of instructions about what to do after deployment, potentially as root. There are examples in `examples/c` and `examples/d` (and see `muddled/filespec.py` for a detailed description of what you can do with filespecs).

`$(MUDDLE_INSTRUCT)` causes muddle to take the specified XML, check its syntax, and stash a copy of the commands contained therein in `.muddle/instructions` (see, that's what it's for .. :-)).

When the deployment has copied all its files to `deploy/` , it looks for stored instructions from the packages and roles that it incorporated and obeys them (it can also register some of its own, of course).

If root privilege is required to complete the deployment, having assembled the instructions it needs to run, the deployment can ask for your password just once and leave you alone the rest of the time. This means you can have passworded `sudo` access and leave your machine happily building whilst you go for coffee without coming back to discover that your machine's been sitting there for 15 minutes waiting for you to type your password.

For completeness, if you're cleaning your package you should probably not leave these cached instructions lying about:

```
$(MUDDLE_UNINSTRUCT)
```

will do the right thing.

3.6 Use Of Libraries

Library code presents a fairly serious problem: most libraries install in essentially two parts - a set of binaries needed to use the library and a set needed to build it.

As such, there is a convention that package directories should be structured:

`obj/<pkg>/<role>/obj`	Object files for the package.
`obj/<pkg>/<role>/include`	Include files.
`obj/<pkg>/<role>/lib`	Library files.

Makefiles can use:

```
CFLAGS += $(MUDDLE_INCLUDE_DIRS:%=-I%)
LDFLAGS += $(MUDDLE_LIB_DIRS:%=-L%)
```

to include the appropriate directories.

3.7 Version control

Muddle is version-control agnostic.

```
muddle help vcs
```

will tell you which version control systems your copy of muddle currently supports. Feel free to add your favourites!

Every VCS is slightly different so muddle's one-model-fits-all approach is sometimes a little awkward. Sorry about that - improvements gratefully recieved - but it does basically work, and it's a lot better than having to bolt your VCS on over the top (or, worse, switch VCS!)

As usual, muddle itself doesn't restrict you here: your project needn't use the same repository, or indeed the same VCS for all its checkouts. However, most do and to help with this muddle has the idea of a root repository stored in the `.muddle/RootRepository` file. This is typically initialised to the repository you got your build description from and it's strongly advised that you leave it there.

Muddle describes repositories with three elements:

- repository URL
- relative path (rest)
- revision

The repository URL is always of the form:

```
[vcs]+[scheme]://[host]/[file]
```

and the precise meaning is delegated to the VCS plugin involved (see `muddled/vcs/*`):

`file+file:///<path>`	Copy files from the given path (useful for building the examples)
`bzr+[URL]`	The bazaar repository at [URL]
`git+[URL]`	The git repository at [URL]
`svn+[URL]`	The Subversion repository at [URL]
`cvs+pserver://[host]/[path]`	The CVS repository at host and path.

The relative path is generally tacked onto the end of the URL for retrieval purposes - so each checkout for bazaar, for example, is a separate bazaar repository - however, for CVS it is the CVS module name - and if you want to use perforce you'll probably want to use it this way there too.

The revision specifies which revision or branch we should check out - with git it's the branch, with bazaar it could be a tag, with CVS it's the (probably sticky?) tag. The special value 'HEAD' means the current head.

By default, you'd specify a checkout with:

```
muddled.checkouts.simple.relative(builder, co_name)
```

which just checks out HEAD of repository = `RootRepository`, relative path = `co_name`, but you can specify your own repositories, revisions, etc. to construct more complex arrangements of checkouts.

A word of caution here: it's possible to get really quite tangled in complex repository layouts and this can make configuration management a nightmare. Muddle is happy to let you shoot yourself in this particular foot, but you might prefer not to.

3.8 The build process

A quick start :-). To build, say, example D, assuming you've checked muddle out in `$MUDDLE_DIR` and added that directory to your path:

```
$ cd /somewhere/convenient
$ muddle init file+file:///$MUDDLE_DIR/examples/d builds/01.py
```

This initialises a muddle build tree with:

```
file+file:///$MUDDLE_DIR/examples/d
```

as its repository and a build description of `builds/01.py`.

The astute will notice that you haven't told muddle which actual repository the build description is in - you've only told it where the repository root is and where the build description *file* is.

Muddle guesses (and at this point it's a very good idea not to try and contradict it - sorry!) that `builds/01.py` means `repository file+file:///$MUDDLE_DIR/examples/d/builds, file 01.py`.

Note that `muddle init` checks out the build description for the sake of politeness - it doesn't really need to, but it feels it probably ought to for the sake of form.

```
$ muddle
```

This is a bit of a cheat. When run with no arguments, muddle attempts to 'do what you mean' (see 'DWIM' below). In this case, you're in the build root and the build description has said:

```
builder.by_default_deploy("example_d")
```

which says 'when someone runs muddle, and you can't think of something better to do, try to deploy the deployment `example_d`'.

As a result, muddle will:

- Fetch `d_co` (the single checkout in this build)
- Build the `d_pkg` package that depends on it and install it in role `x86`.
- Deploy `example_d`, which depends on the `x86` role which `d_pkg` is in.

You'll need to type your password since it wants to `chown` the resulting `hello_world` executable to `root:root`, and you should end up with:

```
rrw@minervois:~/tmp/m4$ ls -lR deploy/
deploy/:
total 4
drwxr-xr-x 3 rrw rrw 4096 2009-06-05 19:15 example_d

deploy/example_d:
total 4
drwxr-xr-x 2 rrw rrw 4096 2009-06-05 19:15 bin

deploy/example_d/bin:
total 4
-rwxr-xr-x 1 root root 888 2009-06-05 19:15 hello_world
```

...which shows the built, deployed `hello_world` binary with the right ownership and permissions. If you're feeling adventurous, you can even run it:

```
rrw@minervois:~/tmp/m4$ ./deploy/example_d/bin/hello_world
Hello, muddle test D world!
rrw@minervois:~/tmp/m4$
```

Amazing, huh? :-)

3.9 Available Muddle Commands

The list of available muddle commands, together with documentation, can be got from:

```
$ muddle help
```

It's best to read the documentation from there, rather than anything here.

A rebuild occurs when you pretend that an object has been updated and then try to remake everything that depends on it.

A build does not imply a deployment! (this helps to avoid endless prompts to enter your password).

3.10 DWIM

When you invoke muddle with no arguments, it will try to guess what you meant. To help it, the build description gives it:

- A list of default roles.
- A list of default deployments.
- Knowledge of where it is in the build tree (the same information as you can get with "muddle where".

The rules it uses are these:

If you invoke muddle from a checkout Build all the packages that depend on this checkout, in all the default roles.

If you invoke muddle from a package build directory Rebuild this package and role.

If you invoke muddle from a package install directory Rebuild every package in this role.

If you invoke muddle from a deployment directory Redeploy this deployment

If you invoke muddle from the root Build the default roles and deployments, if they are defined.

If you don't give an action command like build, rebuild, etc., an argument, muddle will apply the same rules to guess what it should be building, redeploying, etc.

The choice of verb above is a carefully tuned compromise between forcing long build times on (not-so-)innocents who typed `muddle` whilst in the wrong directory and failing to rebuild packages that might have been changed in an over-optimistic attempt to trim the amount of work we need to do.

They will probably need further tuning, and feedback is actively solicited on them. If you feel really deeply about it, you could even (shock! horror!) submit a patch.

3.11 Tips and tricks

Q. I want to specify `--my-pkg-dir=` for the place configure should find `include/` and `lib/` directories for a package?

A. Put something like:

```
MYCOMPONENTDIR=$(shell $(MUDDLE) query objdir package:mycomponent{$(MUDDLE_ROLE)}/built)
```

in your Makefile.

Q. I want to use my deployment as a live install (e.g. to link it to `/opt/where/i/want/to/install`) but redeployment keeps blowing that directory away. What do I do?

A. Use your install directory: `[build_base]/install/role` .. If you really need a deployment - because you're pulling data from multiple roles, for example - file an issue and we'll add a 'justdeploy' command (or you can do it yourself, of course).

3.12 Licencing

muddle is licenced under the MPL 1.1 .

3.13 Any other queries?

Richard Watts, <rrw@kynesim.co.uk> is the man to call. Enjoy!

CHAPTER 4

muddle and its directories

4.1 About this chapter

muddle is three or four different things, entwined:

- It is a command line tool, providing useful tools for managing build trees on Linux.
- It is a Python package, `muddled`
- It is a simple dependency tracking and analysis tool, with its state made evident via the file structure.
- It is a binding of the above to the common tasks required for managing, building and preparing deployments of Linux software from common sources to multiple architectures and platforms.

In a perfect world (perhaps the eventual "muddle 3") these would be better separated. In particular, it would be nice if it was obvious that muddle can be adapted for things other than Linux build trees.

This chapter is meant to be a longer and more discursive introduction to some of the basics of muddle, with a concentration on how the muddle build tree works as a directory tree, and with worked examples.

If you are just going to use muddle to check out and build an existing build tree, then this is probably much too much information, although the short section *An overview of the directory structure* may be useful.

If you are going to be developing new muddle builds, then this chapter should enable you to do so more effectively.

If you are going to be reading the muddle source code, or developing muddle itself, then it should explain much of the intent behind muddle's workings.

4.2 Notation in this chapter

- Directory names are consistently shown with a trailing `/`, as by `ls -F`. Since I'm talking about directories and files a lot, I think this makes it easier to tell which is which.
- The "tree" listings of directory structures are produced with the command `tree`, normally as:

```
$ tree -aF -I .svn --noreport
```

  i.e., showing files and directories that start with a dot, suffixing directory names with `/`, not showing `.svn/` directories, and not reporting on the number of files and directories. In a few places the "stub" of a `.svn/` directory (that is, just its name) may be shown - this should be obvious from context.

- When referrring to the parts of a label (`type:(domain)name{role}/tag`) it is useful to keep the punctuation that indicates that part (so, `type:`, `(domain)`, `name` (no punctuation!), `{role}` and `/tag`).

Remember that, in the current scheme of things, only `package:` labels use `{role}`, and "normal" builds do not use `(domain)`.

4.3 An overview of the directory structure

Normal practice is to keep a muddle build in its own directory. This is not a requirement, but it helps keep it more obvious that it *is* a build.

The muddle directories are as follows:

`.muddle/` contains information about the build state, and signifies that this *is* a muddle build.

`src/` contains the build description and any other checkouts.

`obj/` generated by muddle to hold the results of building packages.

`install/` generated by muddle to hold the results of "installing" packages (this will be explained later on).

`deploy/` generated by muddle to hold the results of "deploying" things in `install/`.

`versions/` generated by muddle to hold the result of `muddle stamp version`.

`domains/` present if there are sub-domains, muddle will create this directory if it is needed.

4.4 A simple example build

The muddle repository has a variety of example build descriptions, of varying complexity.

Perhaps inevitably, not all of the examples are maintained as well as they should be. If you do find problems with any of them, please raise an issue about it on the muddle issues page.

We shall use the "cpio" example.

We start with a new, empty directory:

```
$ mkdir example
$ cd example
```

and then use the muddle `init` command:

```
$ muddle init svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio builds/01.py
> Make directory /home/tibs/sw/m3/example/.muddle
Initialised build tree in /home/tibs/sw/m3/example
Repository: svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
Build description: builds/01.py


Checking out build description ..

> Make directory /home/tibs/sw/m3/example/src
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/builds builds
A    builds/01.py
Checked out revision 458.
> Make directory /home/tibs/sw/m3/example/.muddle/tags/checkout/builds
Done.
```

This leaves us with two new directories:

```
$ ls -AF
.muddle/  src/
```

Or, in more detail (with the `.svn/` directory in `src/builds/` shown as a "stub"):

```
.
|-- .muddle/
|   |-- Description
|   |-- RootRepository
|   `-- tags/
|       `-- checkout/
|           `-- builds/
|               `-- checked_out
`-- src/
    `-- builds/
        |-- 01.py
        |-- 01.pyc
        `-- .svn/
```

Whilst the name and content of the `src/builds` directory might differ, this is the normal state of a muddle build tree after the `init` command.

4.5 The `.muddle` directory

The `.muddle/` directory is, in many ways, the heart of the muddle build tree.

In the first place, it identifies a directory as the top-level of a muddle build. Indeed, the muddle command line tool looks in the current directory, and then upwards through the filesystem, to find a `.muddle/` directory. If it cannot find one, then it decides that it is not in a muddle build tree, and behaves accordingly (depending on what action it was asked to do).

The `.muddle/` directory always contains at least two files and one directory:

4.5.1 The `Description` file

```
$ cat .muddle/Description
builds/01.py
```

which should be recognisable from the `init` command. It tells muddle where its build description was checked out (in the `src/` directory, since that is where checkouts go).

You should not normally need to edit this file, but if you do, the next time you run muddle, it will believe that the build description is in the new location.

4.5.2 The `RootRepository` file

```
$ cat .muddle/RootRepository
svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
```

Again, this should be recognisable from the `init` command. This specifies the default repository, which muddle will assume is being used for checkouts unless the build description says otherwise.

You should not normally need to edit this file either, but again muddle will believe its contents after a change.

4.5.3 The `tags/` directory

The `tags/` directory contains the current state of the build - it is essentially a database of "labels satisfied", stored in the filesystem.

In the current state:

```
.muddle/tags
`-- checkout/
    `-- builds/
        `-- checked_out
```

we can see that the label `checkout:builds/checked_out` has been "built" (or has had its target satisfied). This corresponds to having checked out the `builds` checkout, which is indeed what has been done. We shall see later that it is possible to change the build state by deleting or "`touch`"ing tag files.

The `checked_out` file created by muddle does have content - it contains a timestamp:

```
$ cat .muddle/tags/checkout/builds/checked_out
2010-09-28 11:14:34
```

but this content is never actually used for anything, and if you do add a tag file, it is quite sufficient to use `touch` to create an empty file.

If we ask muddle what checkouts it knows about:

```
$ muddle query checkouts
builds
cpio_co
```

we see two checkouts. The absence of any tag files for the "cpio_co" checkout tells us that it hasn't been checked out yet.

4.5.4 Other contents

There may also be other things in the `.muddle/` directory, notably an `instructions/` directory (used for deployment instructions), and an `am_subdomain` file (described when we talk about domains), but we shall ignore these for now.

4.6 The `src/builds` directory

The other directory present is the `src/` directory, which, so far, contains a single subdirectory for the build description.

```
`src/
 `-- builds/
     |-- 01.py
     |-- 01.pyc
     `-- .svn/
```

4.7 The mechanics of the build - the build description

The build description is one or more Python files, which the muddle command runs before doing anything else.

> (A few muddle commands do not need a build description, obviously including `init`, but these are the exception.)

For various reasons, the main build description file is traditionally called `src/builds/01.py`. muddle automatically adds the directory containing the main build description (the file named in `.muddle/Description`) to the PYTHONPATH for the build description. Put more simply, the build description can `import` other Python files in the same directory, which makes it easy to split build descriptions if necessary.

This example build has a fairly simple build description, in `src/builds/01.py`:

```
#!  /usr/bin/env python
#
# An example of how to build a cpio archive as a
# deployment - e.g. for a Linux initrd.

import muddled
import muddled.pkgs.make
import muddled.deployments.cpio
import muddled.checkouts.simple

def describe_to(builder):
    # Checkout ..
    muddled.checkouts.simple.relative(builder, "cpio_co")
    muddled.pkgs.make.simple(builder, "pkg_cpio", "x86", "cpio_co")
    muddled.deployments.cpio.deploy(builder, "my_archive.cpio",
                                    {"x86": "/"},
                                    "cpio_dep", [ "x86" ])

    builder.add_default_role("x86")
    builder.by_default_deploy("cpio_dep")

# End file.
```

Briefly, muddle looks for a function called `describe_to()` in the build description, and runs it to define the build.

This particular example is quite old, and I'm not sure we'd write it quite like that any more. Regardless, it says the following things:

- there is a checkout called "cpio_co", which is checked out from the default repository (as defined in `.muddle/RootRepository`).
- the package named `pkg_cpio` in role `x86` is built from that checkout - this will, by default, be done using a `Makefile` in the checkout directory.
- when deploying the final results of the build, a CPIO archive shall be created, with the role "x86" going at the root of the filesystem in that CPIO archive. The name of this deployment is "cpio_dep".
- the default package role to build is "x86"
- the default thing to deploy is the "cpio_dep" deployment.

4.8 Some naming conventions

There are some strong naming conventions associated with muddle build trees:

- The build description lives in a checkout called `src/builds/`
- All packages build out-of-tree (that is, in the `obj/` directory, which we shall describe later), so that the `src/` directory *only* contains the checkout files as-checked-out.
- If there is a checkout containing "helper" makefiles and other associated things, used to build checkouts in other directories, it should be called `src/helpers/`.

(For instance, if one is building kbus, and downloading it directly from its google code repository each time, then it is convenient to put the muddle-specific makefile in a different checkout, possibly as `src/helpers/Makefile.kbus`)

There was an earlier convention of using the name `src/builders/` for such a directory, but this is visually confusing, and does not work well with tab-completion at the bash prompt.

- Checkout names should reflect the name of the external package being built, and generally also the version. So, for instance `screen-1.0.3`.
- Package names should be more general - if a package is built from `screen-1.0.3`, it would normally be named `screen`. This allows for a later change in the build to use (for instance) checkout `screen-1.0.4` instead.
- Deployment names should reflect the purpose of the deployment - for instance, `firmware` versus `kernel`, and so on. This is very much dependent on the purpose of the build itself.

Note: This particular build is not really following the naming convention for checkouts, packages and deployments as described above. This is because it is trying to make it very clear which name belongs to which label type (thus `co_cpio`, `pkg_cpio` and `dep_cpio`)

It is also moderately traditional to call the main Python file for a build description `01.py` (or `02.py` if it is version 2, and so on). This is not, however, a strong convention, it just makes the "main" file easy to spot.

4.9 Introspection of the dependency tree

muddle does have some ability to display the dependency tree that is generated from the build description, although it is not as user-friendly as we would like (and, ultimately, there should be graphical tools).

The simplest tool for dumping the dependency rules is the `depends` command (see `muddle help depends` for more details on what it does).

The following shows the dependency rules described directly by the build description above:

```
$ muddle depends user-short
-----
checkout:builds/changes_committed <-VcsCheckoutBuilder-- [ checkout:builds/up_to_date[T] ]
checkout:builds/changes_pushed <-VcsCheckoutBuilder-- [ checkout:builds/changes_committed ]
checkout:builds/pulled <-VcsCheckoutBuilder-- [ checkout:builds/checked_out, checkout:builds/up_
checkout:builds/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:builds/checked_out ]
checkout:cpio_co/changes_committed <-VcsCheckoutBuilder-- [ checkout:cpio_co/up_to_date[T] ]
checkout:cpio_co/changes_pushed <-VcsCheckoutBuilder-- [ checkout:cpio_co/changes_committed ]
checkout:cpio_co/pulled <-VcsCheckoutBuilder-- [ checkout:cpio_co/checked_out, checkout:cpio_co/
checkout:cpio_co/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:cpio_co/checked_out ]
deployment:cpio_dep/deployed <-CpioDeploymentBuilder-- [ package:*{x86}/postinstalled ]
package:pkg_cpio{x86}/built <-MakeBuilder-- [ package:pkg_cpio{x86}/configured ]
package:pkg_cpio{x86}/configured <-MakeBuilder-- [ package:pkg_cpio{x86}/preconfig ]
package:pkg_cpio{x86}/installed <-MakeBuilder-- [ package:pkg_cpio{x86}/built ]
package:pkg_cpio{x86}/postinstalled <-MakeBuilder-- [ package:pkg_cpio{x86}/installed ]
package:pkg_cpio{x86}/preconfig <-MakeBuilder-- [ checkout:cpio_co/checked_out ]
-----
```

Each line above shows three things:

1. a target label.

2. an arrow containing the name of the action to take to "satisfy" or "build" that label (nb: in the current trunk of muddle, this is not present).
3. a list of the labels that must be "satisfied" or "built" before that action can be performed.

It is sorted by target label, which unfortunately is not too much help, but one can see that, for instance, the `deployment:cpio_dep/deployed` label depends on `package:*{x86}/postinstalled`, where the `*` is a wildcard over all package names in the `{x86}` role.

A shorter and perhaps more helpful representation of that can be provided using the `query deps` command:

```
$ muddle query deps deployment:cpio_dep/deployed
Build order for deployment:cpio_dep/deployed ..
checkout:cpio_co/checked_out
package:pkg_cpio{x86}/preconfig
package:pkg_cpio{x86}/configured
package:pkg_cpio{x86}/built
package:pkg_cpio{x86}/installed
package:pkg_cpio{x86}/postinstalled
deployment:cpio_dep/deployed
```

This shows all of the labels that must be built before `deployment:cpio_dep/deployed`, and in what order they will be built.

4.10 Checking out the source code

Normally, after the `init` command, one would just use a "bare" muddle command to perform the rest of the checkouts, build them (as they are checked out), and do any other steps indicated by the build description. However, since I'm interested in the various stages of the build tree, I shall do all of these things one by one.

So we start by checking out the actual source for our build:

```
$ muddle checkout _all
> Building checkout:cpio_co/checked_out
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/cpio_co cpio_co
A    cpio_co/hello_world.c
A    cpio_co/Makefile
Checked out revision 458.
> Make directory /home/tibs/sw/m3/example/.muddle/tags/checkout/cpio_co
```

After doing this, we've now gained a new tag in the `.muddle/` directory:

```
`-- .muddle/
    |-- Description
    |-- RootRepository
    `-- tags/
        `-- checkout/
            |-- builds/
            |   `-- checked_out
            `-- cpio_co/
                `-- checked_out
```

indicating that we have successfully checked out the `cpio_co` checkout, and the source files for our missing checkout are now present in the `src/` directory:

```
`-- src/
    |-- builds/
    |   |-- 01.py
```

```
|   |-- 01.pyc
|   `-- .svn/
`-- cpio_co/
    |-- hello_world.c
    |-- Makefile
    `-- .svn/
```

(whilst I'm showing the `.svn/` directories here, I've removed the listing of their contents).

4.11 The mechanics of the build - the makefile

This build only has a single checkout, which produces a single package. As such it also has a single makefile, `src/cpio_co/Makefile`:

```
# Makefile for cpio_co
#
# Just dumps some compiled C into the right place.

INSTALL=install

all:
        $(CC) -o $(MUDDLE_OBJ)/hello_world hello_world.c

install:
        if [ ! -d $(MUDDLE_INSTALL)/bin ]; then mkdir $(MUDDLE_INSTALL)/bin; fi
        $(INSTALL) -m 0755 $(MUDDLE_OBJ)/hello_world $(MUDDLE_INSTALL)/bin/hello_world
        $(INSTALL) -m 0644 hello_world.c $(MUDDLE_INSTALL)/hello_world.c

config:
        @echo Nothing to do

clean:
        rm -f $(MUDDLE_OBJ)/hello_world

distclean: clean
        @echo Distclean is just a clean

# end file.
```

We saw when querying the builds dependencies (in *Introspection of the dependency tree*) that various dependency rules had `MakeBuilder` as their action. This action knows what targets in a makefile to call in order to build a particular `package:` label tag.

The target label tags (the `/xxx` at the end of a label) correspond to makefile targets as follows:

Target tag	Makefile target
/preconfig	<none>
/configured	config
/built	all
/installed	install
/postinstalled	<none>

Target tag	Makefile target
/clean	clean
/distclean	distclean

It can be useful to think of a package as "progressing" from `/preconfig` through to `/postinstalled`.

`preconfig` does not correspond to a make target, and there is no default action for it. Things that depend on a package existing (but nothing else) will depend on the `/preconfig` tagged package label.

There is no action defined by default for `/preconfig`. That doesn't stop a particular build from defining one - for instance, `ExpandingMakeBuilder` is a MakeBuilder subclass which expands a `.tgz` file into source files for the later stages to compile and install (see `muddled.pkgs.make`).

Although there is no make target associated with `/postinstalled`, it is at this stage that `pkg-config` files are rewritten (if requested), and this is the tag that should be used when another package depends on something being "fully" built.

4.12 Building a package

Whilst it is possible for a package to be built from more than one checkout, this is relatively uncommon, and it is quite usual for a package to correspond directly to a checkout. As it does in this case.

If one tries to build a package that doesn't exist, one will be told so:

```
$ muddle build cpio
Building package:cpio{x86}/postinstalled
There is no rule to build label package:cpio{x86}/postinstalled
```

It's simple to find out what packages there are:

```
$ muddle query packages
pkg_cpio
```

So let's build it:

```
$ muddle build pkg_cpio
Building package:pkg_cpio{x86}/postinstalled
> Building package:pkg_cpio{x86}/preconfig
> Make directory /home/tibs/sw/m3/example/obj/pkg_cpio/x86
> Make directory /home/tibs/sw/m3/example/install/x86
> Make directory /home/tibs/sw/m3/example/.muddle/tags/package/pkg_cpio
> Building package:pkg_cpio{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example/instal
install -m 0755 /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example/i
install -m 0644 hello_world.c /home/tibs/sw/m3/example/install/x86/hello_world.c
> Building package:pkg_cpio{x86}/postinstalled
```

We now have two new directories:

```
$ ls -AFt
install/  obj/  src/  .muddle/
```

The `src/` directory has not changed.

The `.muddle/` directory has gained some new tags in `.muddle/tags/package/pkg_cpio/`, reflecting the new state of the build:

```
.muddle/
|-- Description
|-- RootRepository
`-- tags/
    |-- checkout/
    |   |-- builds/
    |   |   `-- checked_out
    |   `-- cpio_co/
    |       `-- checked_out
    `-- package/
        `-- pkg_cpio/
            |-- x86-built
            |-- x86-configured
            |-- x86-installed
            |-- x86-postinstalled
            `-- x86-preconfig
```

The `obj/` directory contains the results of building packages. Thus we have a new directory called `obj/pkg_cpio/x86`.

```
obj/
`-- pkg_cpio/
    `-- x86/
        `-- hello_world*
```

Specifically, within `obj/` there is a directory named after each package, within which there is a directory named after each role for that package. For convenience, the makefile can use the environment variable `$(MUDDLE_OBJ)` to refer to `<root_dir>/obj/pkg_cpio/x86` (`<root_dir>` indicates the absolute path to the top-level build directory).

There would normally be more use of subdirectories such as `$(MUDDLE_OBJ)/bin` in a "real" build, and muddle provides some other environment variables to help with these.

The `install/` contains the results of installing packages. Each `{role}` gets a separate directory in `install/`:

```
install/
`-- x86/
    |-- bin/
    |   `-- hello_world*
    `-- hello_world.c
```

Again, the makefile can use the environment variable `$(MUDDLE_INSTALL)` to refer to this role-specific directory.

> As an aside, using the `MUDDLE` environment variables in this way means that if two packages (e.g., `package:a{b}/*` versus `package:c{d}/*`, or even `package:a{b}/*` versus `package:a{d}/*`) both use the same checkout, and thus the same makefile, the results of building the two packages will still end up in different, predictable directories.

It is possible to find out the entire environment muddle passes to a makefile with the `makeenv` query:

```
$ muddle query makeenv package:pkg_cpio{x86}/built
MUDDLE=/home/tibs/sw/m3/muddle3_labels/muddle/muddled/__main__.py
MUDDLE_INCLUDE_DIRS=
MUDDLE_INSTALL=/home/tibs/sw/m3/example/install/x86
MUDDLE_INSTRUCT=/home/tibs/sw/m3/muddle3_labels/muddle/muddled/__main__.py instruct pkg_cpio{x86
MUDDLE_KIND=package
MUDDLE_LABEL=package:pkg_cpio{x86}/built
MUDDLE_LD_LIBRARY_PATH=
MUDDLE_LIB_DIRS=
MUDDLE_NAME=pkg_cpio
```

```
    MUDDLE_OBJ=/home/tibs/sw/m3/example/obj/pkg_cpio/x86
    MUDDLE_OBJ_INCLUDE=/home/tibs/sw/m3/example/obj/pkg_cpio/x86/include
    MUDDLE_OBJ_LIB=/home/tibs/sw/m3/example/obj/pkg_cpio/x86/lib
    MUDDLE_OBJ_OBJ=/home/tibs/sw/m3/example/obj/pkg_cpio/x86/obj
    MUDDLE_PKGCONFIG_DIRS=
    MUDDLE_PKGCONFIG_DIRS_AS_PATH=
    MUDDLE_ROLE=x86
    MUDDLE_ROOT=/home/tibs/sw/m3/example
    MUDDLE_SRC=/home/tibs/sw/m3/example/src/cpio_co
    MUDDLE_TAG=built
    MUDDLE_TARGET_LOCATION=/
    MUDDLE_UNINSTRUCT=/home/tibs/sw/m3/muddle3_labels/muddle/muddled/__main__.py instruct pkg_cpio{x
```

Builds specifying cross-compilation toolchains and other options may have much longer environments passed down.

4.13 Deployment

Deployment is the process of copying the files for the various roles from the `install/` directories to the `deploy/` directories.

Deployment is commonly the stage that prepares the "blob" that will be put onto the final device (if one is building an embedded device), and perhaps also aggregates the tools for doing so.

In this build, we are producing a CPIO archive.

> (CPIO is a very old Unix archive format. For our purposes ("us" being muddle), its advantage is that it is possible to flag files within the archive with particular permission bits, to request creationg of device nodes, and other thing that would require superuser privileges if done "live" in the build tree.)

In this build we only have a single deployment target, so we don't need to specify its name.

```
    $ muddle deploy
    Building deployment:cpio_dep/deployed
    > Building deployment:cpio_dep/deployed
    > Make directory /home/tibs/sw/m3/example/deploy/cpio_dep
    Collecting package:*{x86}/*  for deployment to / ..
    h = ---Roots---
    / -> [ / (fs /home/tibs/sw/m3/example/install/x86) mode = 40755 uid = 7007 gid = 7007 kids = /bi
    ---Map---
    /bin/hello_world -> [ /bin/hello_world (fs /home/tibs/sw/m3/example/install/x86/bin/hello_world)
    / -> [ / (fs /home/tibs/sw/m3/example/install/x86) mode = 40755 uid = 7007 gid = 7007 kids = /bi
    /bin -> [ /bin (fs /home/tibs/sw/m3/example/install/x86/bin) mode = 40755 uid = 7007 gid = 7007
    /hello_world.c -> [ /hello_world.c (fs /home/tibs/sw/m3/example/install/x86/hello_world.c) mode
    ---

    base = /
    Scanning instructions for role x86, domain None ..
    > Writing /home/tibs/sw/m3/example/deploy/cpio_dep/my_archive.cpio ..
    > Packing / ..
    > Packing /bin ..
    > Packing /bin/hello_world ..
    > Packing /hello_world.c ..
    > Packing TRAILER!!! ..
    > Make directory /home/tibs/sw/m3/example/.muddle/tags/deployment/cpio_dep
```

That leaves us with a new `deploy/` directory:

```
$ ls -AFt
deploy/  install/  obj/  src/  .muddle/
```

containing our CPIO archive:

```
deploy/
`-- cpio_dep/
    `-- my_archive.cpio
```

and the appropriate tag has been set in the `.muddle/` directory tree:

```
.muddle/tags/deployment/
`-- cpio_dep/
    `-- deployed
```

4.14 Rebuilding things

muddle provides a convenience command to rebuild a package:

```
$ muddle rebuild pkg_cpio
Killing package:pkg_cpio{x86}/built
Clearing tags: package:pkg_cpio{x86}/built package:pkg_cpio{x86}/installed package:pkg_cpio{x86}
Building package:pkg_cpio{x86}/postinstalled
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example/instal
install -m 0755 /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example/i
install -m 0644 hello_world.c /home/tibs/sw/m3/example/install/x86/hello_world.c
> Building package:pkg_cpio{x86}/postinstalled
```

As you can see, this "kills" (deletes) the tags saying that the package has been built, installed and postinstalled, and then rebuilds the `/postinstalled` tag. This does not, however, do anything about things that *depend* on this package.

It is frequently more useful to use the `distrebuild` command, which is a conflation of the `distclean` and `build` commands:

```
$ muddle distrebuild pkg_cpio
Building: package:pkg_cpio{x86}/distclean ..
> Building package:pkg_cpio{x86}/distclean[T]
> make  -f Makefile distclean
rm -f /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world
Distclean is just a clean
Killing: package:pkg_cpio{x86}/preconfig ..
Clearing tags: package:pkg_cpio{x86}/preconfig package:pkg_cpio{x86}/configured package:pkg_cpio
Building package:pkg_cpio{x86}/postinstalled
> Building package:pkg_cpio{x86}/preconfig
> Building package:pkg_cpio{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
```

```
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example/instal
install -m 0755 /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example/i
install -m 0644 hello_world.c /home/tibs/sw/m3/example/install/x86/hello_world.c
> Building package:pkg_cpio{x86}/postinstalled
```

The `distclean` also removes the tags for any packages that depended upon this package (not very obvious in this case where we only had one, of course).

(Beware - it unsets the state of other packages, but not the deployment.)

You can also, quite legally, do the same thing by direct manipulation of the tag files in the `.muddle/` directories:

```
$ rm .muddle/tags/package/pkg_cpio/*
$ muddle build pkg_cpio
Building package:pkg_cpio{x86}/postinstalled
> Building package:pkg_cpio{x86}/preconfig
> Building package:pkg_cpio{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example/instal
install -m 0755 /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example/i
install -m 0644 hello_world.c /home/tibs/sw/m3/example/install/x86/hello_world.c
> Building package:pkg_cpio{x86}/postinstalled
```

That is, muddle doesn't care if you remove the tags directly, instead of via the command line tool.

You can even do things like:

```
$ rm .muddle/tags/checkout/builds/checked_out
$ muddle checkout _all
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/builds builds
Checked out revision 459.
```

This is quite deliberate. Whilst the muddle command line tool provides a variery of useful shorthand commands (such as "distclean", "distrebuild" and "redeploy"), it is quite possible and even sensible to exert finer control over a build by deleting (or, indeed, "touch"ing) tag files in the `.muddle` directory.

4.15 Do what I say, not what I do

Please note that although I have been explicitly requesting indiviual builds, rebuilds, deployments and so on above, this is not necessarily the normal way of using muddle. The muddle command line tool is carefully designed so that it normally does "the right thing", according to which directory you are in.

So, if one is at the top level of the build, and has checked out everything, it is more colloquial just to give the muddle command with no arguments than to be overly specific.

This is perhaps most easily shown by, well, showing it.

We can revert to the just checked out state quite easily, by just removing everything except the `.muddle/` and `src/` directories:

```
$ ls -AF
deploy/  install/  .muddle/  obj/  src/
$ rm -rf deploy/ install/ obj/
$ rm -rf .muddle/tags/package/ .muddle/tags/deployment/
```

Once we're back to the stage just after `init`, we can just do:

```
$ muddle
Building deployment:cpio_dep/deployed
> Building package:pkg_cpio{x86}/preconfig
...
> Building package:pkg_cpio{x86}/configured
...
> Building package:pkg_cpio{x86}/built
...
> Building package:pkg_cpio{x86}/installed
...
> Building package:pkg_cpio{x86}/postinstalled
> Building deployment:cpio_dep/deployed
...
> Writing /home/tibs/sw/m3/example/deploy/cpio_dep/my_archive.cpio ..
...
> Make directory /home/tibs/sw/m3/example/.muddle/tags/deployment/cpio_dep
```

(I've shortened the log above, because by now it should be all too familiar.)

Contrariwise, if we clean it all up again:

```
$ rm -rf deploy/ install/ obj/
$ rm -rf .muddle/tags/package/ .muddle/tags/deployment/
```

and go into a particular checkout directory:

```
$ pushd src/cpio_co
$ muddle
Killing package:pkg_cpio{x86}/built
Clearing tags: package:pkg_cpio{x86}/built package:pkg_cpio{x86}/installed package:pkg_cpio{x86}
Building package:pkg_cpio{x86}/*
> Building package:pkg_cpio{x86}/distclean[T]
> Make directory /home/tibs/sw/m3/example/obj/pkg_cpio/x86
> Make directory /home/tibs/sw/m3/example/install/x86
> make  -f Makefile distclean
rm -f /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world
Distclean is just a clean
> Building package:pkg_cpio{x86}/clean[T]
> make  -f Makefile clean
rm -f /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world
> Building package:pkg_cpio{x86}/preconfig
> Make directory /home/tibs/sw/m3/example/.muddle/tags/package/pkg_cpio
> Building package:pkg_cpio{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example/instal
install -m 0755 /home/tibs/sw/m3/example/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example/i
```

```
    install -m 0644 hello_world.c /home/tibs/sw/m3/example/install/x86/hello_world.c
    > Building package:pkg_cpio{x86}/postinstalled
    $ popd
```

That is, muddle only rebuilds the packages that depend on this checkout. See `muddle help build` and its friends for more information on this.

People who like to `cd` around the filesystem a lot will probably find this intuitive. Those of us who prefer to sit at the top-level and work from there (not necessarily any more sensible, of course) are more likely just to do:

```
    $ muddle distrebuild pkg_cpio
```

(yes, this has identical effect, and no, by now I'm not going to include the listing yet again).

Note: As an aside, if you want to preserve the source of a build, then (if there aren't any domains) it is sufficient to do:

```
$ tar -zcvf build.tgz .muddle/Description .muddle/RootRepository .muddle/tags/checkout src
```

However, muddle provides an alternative that will work in all cases, including when there are subdomains:

```
  $ muddle distribute --with-vcs _source_release ../save_dir
  $ cd ..
  $ tar -zcvf build.tgz save_dir

(where ``save_dir`` should be replaced by something more informative).
```

4.16 Adding a new checkout/package

Let's suppose we want to add a new package, based on a new checkout. There is more than one way to do this, but the following shows a sensible approach.

We shall create our new package by just copying one we've already got:

```
    $ cp -a src/cpio_co src/fred
```

We must remember to remove the outdated repository information:

```
    $ rm -rf src/fred/,svn/
```

And make this checkout do something different:

```
    $ mv src/fred/hello_world.c src/fred/bye_world.c
    $ sed -e s/hello_world/bye_world/g --in-place src/fred/Makefile
    $ sed -e s/Hello/Byebye/g --in-place src/fred/bye_world.c
```

muddle has no knowledge of this checkout yet, so we need to add it to the build description. We can simply edit `src/builds/01.py` to add the lines:

```
    muddled.checkouts.simple.relative(builder, "fred")
```

and:

```
    muddled.pkgs.make.simple(builder, "fred", "x86", "fred")
```

(yes, that leaves us with a package and checkout with the same name, but that's not a problem, and is closer to the more conventional usage).

That's enough to give us:

```
$ muddle query checkouts
builds
cpio_co
fred
```

and:

```
$ muddle query deps deployment:cpio_dep/deployed
Build order for deployment:cpio_dep/deployed ..
checkout:fred/checked_out
checkout:cpio_co/checked_out
package:pkg_cpio{x86}/preconfig
package:fred{x86}/preconfig
package:pkg_cpio{x86}/configured
package:fred{x86}/configured
package:fred{x86}/built
package:pkg_cpio{x86}/built
package:fred{x86}/installed
package:pkg_cpio{x86}/installed
package:pkg_cpio{x86}/postinstalled
deployment:cpio_dep/deployed
```

However, if we were to try to build:

```
$ muddle
Building deployment:cpio_dep/deployed
> Building checkout:fred/checked_out
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/fred fred
svn: URL 'http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/fred' doesn't exist
Can't build deployment:cpio_dep/deployed - Command 'svn checkout  http://muddle.googlecode.com/s
```

This makes sense, because (a) we have not told muddle that the new checkout is present in its directory structure, and (b) we have not actually checked it in to the subversion repository it wants to look in.

Since we've not yet finished developing this new package, we don't want to check it in to the repository yet (and, in particular in our case, we do not want to permanently add it to our example).

So we need to make muddle believe that it has been checked out.

As you might expect, there are two ways to do this. The one that you may guess is to manipulate the `.muddle/` directory structure directly - so we could just do:

```
$ mkdir .muddle/tags/checkout/fred
$ touch .muddle/tags/checkout/fred/checked_out
```

However, that's a bit clumsy to type, so muddle provides a convenient command for asserting a particular tag (just assuming you are comfortable with the label syntax):

```
$ muddle assert checkout:fred/checked_out
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/checkout/fred
```

and now the tag file is present:

```
$ ls .muddle/tags/checkout/fred
checked_out
```

and we can then do:

```
$ muddle
Building deployment:cpio_dep/deployed
> Building package:fred{x86}/preconfig
> Make directory /home/tibs/sw/m3/example3/obj/fred/x86
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/package/fred
> Building package:fred{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:fred{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example3/obj/fred/x86/bye_world bye_world.c
> Building package:fred{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example3/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example3/inst
install -m 0755 /home/tibs/sw/m3/example3/obj/fred/x86/bye_world /home/tibs/sw/m3/example3/insta
install -m 0644 bye_world.c /home/tibs/sw/m3/example3/install/x86/bye_world.c
> Building deployment:cpio_dep/deployed
> Make directory /home/tibs/sw/m3/example3/deploy/cpio_dep
Collecting package:*{x86}/*  for deployment to / ..
h = ---Roots---
/ -> [ / (fs /home/tibs/sw/m3/example3/install/x86) mode = 40755 uid = 7007 gid = 7007 kids = /b
---Map---
/bin/bye_world -> [ /bin/bye_world (fs /home/tibs/sw/m3/example3/install/x86/bin/bye_world) mode
/bin -> [ /bin (fs /home/tibs/sw/m3/example3/install/x86/bin) mode = 40755 uid = 7007 gid = 7007
/hello_world.c -> [ /hello_world.c (fs /home/tibs/sw/m3/example3/install/x86/hello_world.c) mode
/bin/hello_world -> [ /bin/hello_world (fs /home/tibs/sw/m3/example3/install/x86/bin/hello_world
/bye_world.c -> [ /bye_world.c (fs /home/tibs/sw/m3/example3/install/x86/bye_world.c) mode = 100
/ -> [ / (fs /home/tibs/sw/m3/example3/install/x86) mode = 40755 uid = 7007 gid = 7007 kids = /b
---

base = /
Scanning instructions for role x86, domain None ..
> Writing /home/tibs/sw/m3/example3/deploy/cpio_dep/my_archive.cpio ..
> Packing / ..
> Packing /bin ..
> Packing /bin/bye_world ..
> Packing /bin/hello_world ..
> Packing /hello_world.c ..
> Packing /bye_world.c ..
> Packing TRAILER!!! ..
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/deployment/cpio_dep
```

This just leaves actually adding the new package to revision control somewhere - in this case, the build description is saying (implicitly) that this package is stored in the same subversion repository as the rest of the example, and so one would need to add it there by the normal means (which is a topic for another time).

> (The fact that this is *much* easier to do for distributed revision control systems like bazaar, mercurial or git is in itself a good reason for using them!)

4.17 Version stamps

Version stamps allow one to produce a simple text file (actually, an `INI` file) representing the current state of a build. This can be useful for various purposes, but its main intent is to allow saving a build state so that it can be accurately recreated at a later date (for instance, so one can rebuild an earlier release to customers).

The muddle `stamp` and `unstamp` commands have documentation via muddle `help stamp` and `help unstamp`.

However, it is worth giving a quick example of creating a version stamp:

```
$ muddle stamp version
Finding all checkouts... found 2
Processing Svn checkout 'builds'
Processing Svn checkout 'cpio_co'
Creating directory /home/tibs/sw/m3/example/versions
Writing to /home/tibs/sw/m3/example/versions/_temporary.stamp
Wrote revision data to /home/tibs/sw/m3/example/versions/_temporary.stamp
File has SHA1 hash 189085d413217cb1865868b5d9916085d22e6f50
Renaming /home/tibs/sw/m3/example/versions/_temporary.stamp to /home/tibs/sw/m3/example/versions
```

As indicated above, we now have a new `versions/` directory:

```
$ ls -AFt
versions/  deploy/  install/  obj/  src/  .muddle/
```

containing the stamp file:

```
versions
`-- 01.stamp
```

Modern build descriptions are encouraged to specify a *build name* (this is as simple as adding a line of the form:

```
builder.build_name = 'ExampleBuild'
```

to the Python code). If this is given, then the version stamp file will use that build name as its filename, otherwise it defaults to the filename of the main build description file (`01` in this case).

The `versions/` directory is suitable for putting into a version control system, as one might expect, and future versions of muddle are likely to provide more support for handling this (see issue 117 for some ideas on this).

The contents of a stamp file is deliberately human readable:

```
[ROOT]
description = builds/01.py
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio

[CHECKOUT builds]
name = builds
relative = builds
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
revision = 458

[CHECKOUT cpio_co]
name = cpio_co
relative = cpio_co
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
revision = 458
```

As normal. muddle will not stop you writing or editing stamp files. This may conceivably be useful.

4.18 Domains

Domains are part of advanced muddle use, and are probably not relevant to most builds. Regardless, it is probably worth reading at least *What domains are for* from this section, as useful background, and so that you can recognise when they might be applicable.

The way domains work follows naturally from the rest of muddle, but the implementation is still being tidied up - hence the `muddle3_label` branch.

4.18.1 What domains are for

Domains allow one to include one build description inside another, in much the same way as a Python module can import another.

A simple example might be when building a system with WebKit and X11 support. We might have two builds, constructed as follows:

```
def UI_Build:
   import WebKit_build
       import X11_build
```

and:

```
def Login_build:
    import X11_build
```

Alternatively, we might have some high-level software for handling internet television, and want to build two systems on different architectures. In this case, our contrasting builds might be:

```
def IPTV_stack:
    import ARM_based_stack
```

and:

```
def IPTV_stack:
    import MIPS_based_stack
```

4.18.2 How domains work

A build includes another build (a *subdomain*) using the `include_domain()` function. This takes the same arguments as the muddle `init` command (that is, a repository specification and a build description therein), plus a name for the domain.

The following sequence of actions is then performed:

1. Create a directory called `domains/<name>/`, where `<name>` is the name of the new domain.
2. `cd` into that new directory, and perform (the equivalent of) a muddle `init` command therein, using the repository specification and build description given.
3. Read in this new build, giving a stand-alone build tree datastructure.
4. Find all of the labels in the new build tree datastructures, and *change* them to add the domain name. So, for instance, `checkout:fred/checked_out` would become `checkout:(<name>)fred/checked_out`.
5. Incorporate this amended build tree datastructure into the original ("top level") build.
6. Create a `.muddle/am_subdomain` file in the domain. This allows muddle to distinguish the *actual* (final) top-level build from any subdomains within it, which is a useful optimisation.

Note that the build description for a domain does not itself know that it is not the top-level of a muddle build. Indeed, this is an important property of domains - it means that any build description can potentially be included in any other.

For most purposes, the subdomain works just as if it were a "normal" top-level build. In particular, it uses its own `.muddle/` directory to record its build state, and writes things to its own `obj/` and `install/` directories.

Also note that, as a consequence of the way domains work (and this is a good thing), it is only possible for a build to refer to labels in subdomains, not to those in sibling or parent build trees.

To allow for more flexibility in how domains are used, there are tools in the `muddled` package for manipulating the merged label-space. For instance, if one is using subdomains that originated as two different set-top-box builds, both might provide a linux kernel, and the top-level build probably wants to amend the labels such that both subdomains use *one* of the kernel packages (broadly, saying "this label should be used instead of that label").

4.18.3 A worked example of using domains

So, we shall start in a new empty directory:

```
$ cd ..
$ mkdir example2
$ cd example2
```

and start with the same build that we used before:

```
$ muddle init svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio builds/01.py
> Make directory /home/tibs/sw/m3/example2/.muddle
Initialised build tree in /home/tibs/sw/m3/example2
Repository: svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
Build description: builds/01.py


Checking out build description ..

> Make directory /home/tibs/sw/m3/example2/src
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/builds builds
A    builds/01.py
Checked out revision 459.
> Make directory /home/tibs/sw/m3/example2/.muddle/tags/checkout/builds
Done.
```

So, just as before, we now have the following directory structure:

```
.
|-- .muddle/
|   |-- Description
|   |-- RootRepository
|   `-- tags/
|       `-- checkout/
|           `-- builds/
|               `-- checked_out
`-- src/
    `-- builds/
        |-- 01.py
        `-- 01.pyc
```

We can now edit the `src/builds/01.py` file to include a subdomain as part of the build. As described above, we need to specify how to retrieve the domain, giving the same information as for the "muddle init" command. We shall call our domain `b` (after the example we're taking it from):

```
from muddled.mechanics import include_domain
include_domain(builder,
               domain_name = "b",
               domain_repo = "svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b",
               domain_desc = "builds/01.py")
```

We also have to say how to include the result into our deployment - a simple way to do this is:

```
    import muddled.deployments.collect as collect
    collect.deploy(builder, "everything")
    collect.copy_from_role_install(builder, "everything",
                                   role = "x86",
                                   rel = "", dest = "",
                                   domain = None)
    collect.copy_from_role_install(builder, "everything",
                                   role = "x86",
                                   rel = "", dest = "usr",
                                   domain = "b")
```

and then deploy the new deployment "everything", instead of the old "cpio_dep". Note that we're not deploying to a CPIO archive this time, but just as normal files.

We should probably also change the introductory comment to:

```
    # An example of building with a subdomain
```

So the "top level" build description is now:

```
    #!  /usr/bin/env python
    #
    # An example of building with a subdomain

    import muddled
    import muddled.pkgs.make
    import muddled.deployments.cpio
    import muddled.checkouts.simple
    import muddled.deployments.collect as collect
    from muddled.mechanics import include_domain

    def describe_to(builder):
        # Checkout ..
        muddled.checkouts.simple.relative(builder, "cpio_co")
        muddled.pkgs.make.simple(builder, "pkg_cpio", "x86", "cpio_co")

        include_domain(builder,
                       domain_name = "b",
                       domain_repo = "svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b",
                       domain_desc = "builds/01.py")

        collect.deploy(builder, "everything")
        collect.copy_from_role_install(builder, "everything",
                                       role = "x86",
                                       rel = "", dest = "",
                                       domain = None)
        collect.copy_from_role_install(builder, "everything",
                                       role = "x86",
                                       rel = "", dest = "usr",
                                       domain = "b")

        builder.add_default_role("x86")
        builder.by_default_deploy("everything")

    # End file.
```

If we *now* do a "checkout _all":

```
$ muddle checkout _all
> Make directory /home/tibs/sw/m3/example3/domains/b/.muddle
Initialised build tree in /home/tibs/sw/m3/example3/domains/b
Repository: svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b
Build description: builds/01.py


Checking out build description ..

> Make directory /home/tibs/sw/m3/example3/domains/b/src
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/b/builds builds
A    builds/01.py
Checked out revision 459.
> Make directory /home/tibs/sw/m3/example3/domains/b/.muddle/tags/checkout/builds
There is no rule to build label checkout:b_co/checked_out
> Building checkout:cpio_co/checked_out
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio/cpio_co cpio_co
A    cpio_co/hello_world.c
A    cpio_co/Makefile
Checked out revision 459.
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/checkout/cpio_co
```

giving us:

```
$ ls -AF
domains/  .muddle/  src/
```

or, in more detail (omitting `.svn` directories):

```
.
|-- domains/
|   `-- b/
|       |-- .muddle/
|       |   |-- am_subdomain
|       |   |-- Description
|       |   |-- RootRepository
|       |   `-- tags/
|       |       `-- checkout/
|       |           `-- builds/
|       |               `-- checked_out
|       `-- src/
|           `-- builds/
|               |-- 01.py
|               `-- 01.pyc
|-- .muddle/
|   |-- Description
|   |-- RootRepository
|   `-- tags/
|       `-- checkout/
|           |-- builds/
|           |   `-- checked_out
|           `-- cpio_co/
|               `-- checked_out
`-- src/
    |-- builds/
    |   |-- 01.py
    |   `-- 01.pyc
    `-- cpio_co/
        |-- hello_world.c
```

```
            `-- Makefile
```

Apart from the `am_subdomain` file in its `.muddle/` directory, the `domains/b/` directory looks like a perfectly normal build.

If we then do:

```
$ muddle build _all
Building package:(b)pkg_b{x86}/postinstalled package:pkg_cpio{x86}/postinstalled
> Building checkout:(b)b_co/checked_out
> svn checkout  http://muddle.googlecode.com/svn/trunk/muddle/examples/b/b_co b_co
A    b_co/hello_world.c
A    b_co/Makefile
Checked out revision 459.
> Make directory /home/tibs/sw/m3/example3/domains/b/.muddle/tags/checkout/b_co
> Building package:(b)pkg_b{x86}/preconfig
> Make directory /home/tibs/sw/m3/example3/domains/b/obj/pkg_b/x86
> Make directory /home/tibs/sw/m3/example3/domains/b/install/x86
> Make directory /home/tibs/sw/m3/example3/domains/b/.muddle/tags/package/pkg_b
> Building package:(b)pkg_b{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:(b)pkg_b{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example3/domains/b/obj/pkg_b/x86/hello_world hello_world.c
> Building package:(b)pkg_b{x86}/installed
> make  -f Makefile install
install -m 0755 /home/tibs/sw/m3/example3/domains/b/obj/pkg_b/x86/hello_world /home/tibs/sw/m3/e
> Building package:(b)pkg_b{x86}/postinstalled
> Building package:pkg_cpio{x86}/preconfig
> Make directory /home/tibs/sw/m3/example3/obj/pkg_cpio/x86
> Make directory /home/tibs/sw/m3/example3/install/x86
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/package/pkg_cpio
> Building package:pkg_cpio{x86}/configured
> make  -f Makefile config
Nothing to do
> Building package:pkg_cpio{x86}/built
> make  -f Makefile
cc -o /home/tibs/sw/m3/example3/obj/pkg_cpio/x86/hello_world hello_world.c
> Building package:pkg_cpio{x86}/installed
> make  -f Makefile install
if [ ! -d /home/tibs/sw/m3/example3/install/x86/bin ]; then mkdir /home/tibs/sw/m3/example3/inst
install -m 0755 /home/tibs/sw/m3/example3/obj/pkg_cpio/x86/hello_world /home/tibs/sw/m3/example3
install -m 0644 hello_world.c /home/tibs/sw/m3/example3/install/x86/hello_world.c
> Building package:pkg_cpio{x86}/postinstalled
```

we end up with the subdomain built in its directory structure:

```
domains/
`-- b/
    |-- install/
    |   `-- x86/
    |       `-- hello_world*
    |-- .muddle/
    |   |-- am_subdomain
    |   |-- Description
    |   |-- RootRepository
    |   `-- tags/
    |       |-- checkout/
```

```
        |       |   |-- b_co/
        |       |   |   `-- checked_out
        |       |   `-- builds/
        |       |       `-- checked_out
        |       `-- package/
        |           `-- pkg_b/
        |               |-- x86-built
        |               |-- x86-configured
        |               |-- x86-installed
        |               |-- x86-postinstalled
        |               `-- x86-preconfig
        |-- obj/
        |   `-- pkg_b/
        |       `-- x86/
        |           `-- hello_world*
        `-- src/
            |-- b_co/
            |   |-- hello_world.c
            |   `-- Makefile
            `-- builds/
                |-- 01.py
                `-- 01.pyc
```

and the top-level build in its:

```
    `-- install/
    |   `-- x86/
    |       |-- bin/
    |       |   `-- hello_world*
    |       `-- hello_world.c
    |-- .muddle/
    |   |-- Description
    |   |-- RootRepository
    |   `-- tags/
    |       |-- checkout/
    |       |   |-- builds/
    |       |   |   `-- checked_out
    |       |   `-- cpio_co/
    |       |       `-- checked_out
    |       `-- package/
    |           `-- pkg_cpio/
    |               |-- x86-built
    |               |-- x86-configured
    |               |-- x86-installed
    |               |-- x86-postinstalled
    |               `-- x86-preconfig
    |-- obj/
    |   `-- pkg_cpio/
    |       `-- x86/
    |           `-- hello_world*
    `-- src/
        |-- builds/
        |   |-- 01.py
        |   `-- 01.pyc
        `-- cpio_co/
            |-- hello_world.c
            `-- Makefile
```

If we ask after packages, we have both sets:

```
$ muddle query packages
pkg_b
pkg_cpio
```

(this should arguably report the domain of each package name as well - there should probably be an option to select this).

If we ask what domains we have, we get:

```
$ muddle query domains

b
```

(the current printout of domains includes the "empty" or top-level domain in its listing, which is slightly unobvious, and will probably be fixed at some time).

Our dependency rules are now extended to include those from the subdomain as well:

```
$ muddle depend user-short
-----
checkout:builds/changes_committed <-VcsCheckoutBuilder-- [ checkout:builds/up_to_date[T] ]
checkout:builds/changes_pushed <-VcsCheckoutBuilder-- [ checkout:builds/changes_committed ]
checkout:builds/pulled <-VcsCheckoutBuilder-- [ checkout:builds/checked_out, checkout:builds/up_
checkout:builds/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:builds/checked_out ]
checkout:cpio_co/changes_committed <-VcsCheckoutBuilder-- [ checkout:cpio_co/up_to_date[T] ]
checkout:cpio_co/changes_pushed <-VcsCheckoutBuilder-- [ checkout:cpio_co/changes_committed ]
checkout:cpio_co/pulled <-VcsCheckoutBuilder-- [ checkout:cpio_co/checked_out, checkout:cpio_co/
checkout:cpio_co/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:cpio_co/checked_out ]
checkout:(b)b_co/changes_committed <-VcsCheckoutBuilder-- [ checkout:(b)b_co/up_to_date[T] ]
checkout:(b)b_co/changes_pushed <-VcsCheckoutBuilder-- [ checkout:(b)b_co/changes_committed ]
checkout:(b)b_co/pulled <-VcsCheckoutBuilder-- [ checkout:(b)b_co/checked_out, checkout:(b)b_co/
checkout:(b)b_co/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:(b)b_co/checked_out ]
checkout:(b)builds/changes_committed <-VcsCheckoutBuilder-- [ checkout:(b)builds/up_to_date[T] ]
checkout:(b)builds/changes_pushed <-VcsCheckoutBuilder-- [ checkout:(b)builds/changes_committed
checkout:(b)builds/pulled <-VcsCheckoutBuilder-- [ checkout:(b)builds/checked_out, checkout:(b)b
checkout:(b)builds/up_to_date[T] <-VcsCheckoutBuilder-- [ checkout:(b)builds/checked_out ]
deployment:everything/deployed <-CollectDeploymentBuilder-- [ package:*{x86}/postinstalled, pack
package:pkg_cpio{x86}/built <-MakeBuilder-- [ package:pkg_cpio{x86}/configured ]
package:pkg_cpio{x86}/configured <-MakeBuilder-- [ package:pkg_cpio{x86}/preconfig ]
package:pkg_cpio{x86}/installed <-MakeBuilder-- [ package:pkg_cpio{x86}/built ]
package:pkg_cpio{x86}/postinstalled <-MakeBuilder-- [ package:pkg_cpio{x86}/installed ]
package:pkg_cpio{x86}/preconfig <-MakeBuilder-- [ checkout:cpio_co/checked_out ]
package:(b)pkg_b{x86}/built <-MakeBuilder-- [ package:(b)pkg_b{x86}/configured ]
package:(b)pkg_b{x86}/configured <-MakeBuilder-- [ package:(b)pkg_b{x86}/preconfig ]
package:(b)pkg_b{x86}/installed <-MakeBuilder-- [ package:(b)pkg_b{x86}/built ]
package:(b)pkg_b{x86}/postinstalled <-MakeBuilder-- [ package:(b)pkg_b{x86}/installed ]
package:(b)pkg_b{x86}/preconfig <-MakeBuilder-- [ checkout:(b)b_co/checked_out ]
-----
```

and our deployment has the expected build order:

```
$ muddle query deps deployment:everything/deployed
Build order for deployment:everything/deployed ..
checkout:cpio_co/checked_out
checkout:(b)b_co/checked_out
package:(b)pkg_b{x86}/preconfig
package:pkg_cpio{x86}/preconfig
package:(b)pkg_b{x86}/configured
package:pkg_cpio{x86}/configured
package:(b)pkg_b{x86}/built
```

```
package:pkg_cpio{x86}/built
package:(b)pkg_b{x86}/installed
package:pkg_cpio{x86}/installed
package:(b)pkg_b{x86}/postinstalled
package:pkg_cpio{x86}/postinstalled
deployment:everything/deployed
```

Which means that when we deploy:

```
$ muddle deploy
Building deployment:everything/deployed
> Building deployment:everything/deployed
> Make directory /home/tibs/sw/m3/example3/deploy/everything
Copying /home/tibs/sw/m3/example3/install/x86/ to /home/tibs/sw/m3/example3/deploy/everything/
Copying /home/tibs/sw/m3/example3/domains/b/install/x86/ to /home/tibs/sw/m3/example3/deploy/eve
> Make directory /home/tibs/sw/m3/example3/.muddle/tags/deployment/everything
```

we end up with everything deployed in the `deploy/` directory at the top-level:

```
deploy/
`-- everything/
    |-- bin/
    |   `-- hello_world*
    |-- hello_world.c
    `-- usr/
        `-- hello_world*
```

`bin/hello_world` comes from the top-level build, and `usr/hello_world` from the subdomain. The `hello_world.c` is also from the top-level build (the example Makefile for the `cpio` package copies the source file into the `install/` directory).

4.18.4 Version stamps and domains

Version stamps work with domains as well, although one may only generate them for the top-level in a particular build tree.

Thus we can try:

```
$ muddle stamp version
Finding all checkouts... found 4
Processing Svn checkout 'builds'
builds: 'svnversion' reports checkout has revision '459M'
Processing Svn checkout 'cpio_co'
Processing Svn checkout '(b)b_co'
Processing Svn checkout '(b)builds'
Found domains: set([DomainTuple(name='b', repository='svn+http://muddle.googlecode.com/svn/trunk

Unable to work out revision ids for all the checkouts
- although we did work out 3 of 4
Problems were:
* builds: 'svnversion' reports checkout has revision '459M'
Problems prevent writing version stamp file
```

OK, that makes sense, since we had indeed edited that checkout, and had not commited the changes. Luckily, it is sufficient for our purposes to save the state without that edit:

```
$ muddle stamp save -force
Forcing original revision ids when necessary
```

```
Finding all checkouts... found 4
Processing Svn checkout 'builds'
builds: 'svnversion' reports checkout has revision '459M'
Processing Svn checkout 'cpio_co'
Processing Svn checkout '(b)b_co'
Processing Svn checkout '(b)builds'
Found domains: set([DomainTuple(name='b', repository='svn+http://muddle.googlecode.com/svn/trunk

Unable to work out revision ids for all the checkouts
- although we did work out 3 of 4
Problems were:
* builds: 'svnversion' reports checkout has revision '459M'
Writing to working.stamp
Wrote revision data to working.stamp
File has SHA1 hash 6425284f002c81c9849f2f23add025a710207509
Renaming working.stamp to 6425284f002c81c9849f2f23add025a710207509.partial
```

and that at least shows us that the stamp file *does* record details of any subdomains:

```
[ROOT]
description = builds/01.py
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio

[DOMAIN b]
description = builds/01.py
name = b
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b

[CHECKOUT (b)b_co]
domain = b
name = b_co
relative = b_co
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b
revision = 459

[CHECKOUT (b)builds]
domain = b
name = builds
relative = builds
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/b
revision = 459

[CHECKOUT cpio_co]
name = cpio_co
relative = cpio_co
repository = svn+http://muddle.googlecode.com/svn/trunk/muddle/examples/cpio
revision = 459

[PROBLEMS]
problem1 = builds: 'svnversion' reports checkout has revision '459M'
```

CHAPTER 5

Muddle workflows - or how to use muddle in real life

This section is intended to give some examples of how one actually uses muddle in normal programming life.

If you are using weld, see the chapter on working with weld.

5.1 Working with version control systems

or, "Should I use ''muddle push/pull'' or ''git push/pull''?"

Each checkout uses a particular VCS (above we're assuming it's git), but muddle also provides some VCS commands, notably `muddle push`, `muddle pull` and `muddle merge`.

So, when one could use the VCS-specific command, or the muddle command, which should one use?

My normal practise is to use:

- `muddle pull` - to update a working directory from the repository
- `muddle push` - to push changes once they're done

and otherwise use the appropriate VCS commands to manipulate local state.

The reason for using the muddle commands for pull/push is that the build description keeps track of where the far repository is, and this *can* change - we've had occasion to do that whilst working with various build trees. Secondly, different VCS's have different degrees of guessing correctly (in our usage) that the push and pull are to be for the same place, and muddle already knows that.

For any other operations, specific VCS commands are finer grained and generally more useful.

So, if I was editing the build description (`01.py` in this case), then I would typically:

1. Edit `src/builds/01.py`.
2. Test that `muddle help` still works - this is a good general test that the build description is at least still valid Python code, and doesn't contain circular dependencies or anything else pathological.
3. Test that everything (or some representative sample) still builds OK
4. Use git to commit `01.py`, with an appropriate commit message
5. Use muddle to push it
6. Use gitk to check I've done what I think I have

Similarly, if I were editing source code in a checkout, I would:

1. `cd` or `pushd` into the checkout directory

2. Perform the edit.
3. Use `muddle` or `muddle distrebuild` to rebuild it, and if that fails, do (2) again (and so on).
4. Use git (or whatever the appropriate VCS is) to commit the changes.
5. Use `muddle push` to push them.

(And as to `muddle merge` - I'd use the VCS specific command in all cases, and it is possible that `muddle merge` may go away at some point in the future.)

5.2 Creating a new checkout, and adding it to (for example) git

When developing a new system, it is periodically necessary to add a new checkout to the system as a whole.

For the purpose of this illustration, I shall assume that the build is using git to store its repositories, and the muddle build was intialised using the command:

```
$ muddle init git+ssh://git@git.kynesim.co.uk/opt/projects/Fred builds/01.py
```

(note that this is *not* a real command line, there is no such server).

That is, we are expecting to connect to the server with SSH, all interaction is via the user `git`, and project repositories are to be found as subdirectories of `/opt/projects/Fred` on the server (typically named to reflect the layout of checkouts in the build trees `src/` directory).

Each user will normally have an SSH key allowing them access to the server machine, as user `git` (that is, their public key will be in the `git` users `.ssh/authorized_keys` file).

So, to add a new checkout, we start by creating the repository on the server.

Logon to the server machine as `git`, something like:

```
$ ssh git@git.kynesim.co.uk
```

Then `cd` to the project root:

```
$ cd /opt/projects/Fred
```

We will assume we are adding the `libpng` library, which will live in `src/libs`, and we thus want to put it into a `libs` subdirectory on the server. We're actually adding version 1.4.3 of the library, so we put that information into the checkout name (in fact, this will typically be how the directory would be named after retrieving it from its original home). This allows our build tree to change the version of a checkout in a clean fashion, later on.

So, create the new directory:

```
git@git.kynesim.co.uk$ mkdir -p libs/libpng-1.4.3
```

and tell `git` to set it up as a bare repository:

```
git@git.kynesim.co.uk$ cd libs/libpng-1.4.3
git@git.kynesim.co.uk$ git init --bare
```

You can then log out of the server (I surely don't need to show that).

Back in the development tree, we can now add the new directory - in this instance we're unpacking an archive we'd already downloaded:

```
$ cd src/libs
$ tar -zxvf ~/Downloads/libpng-1.4.3.tar.gz
$ cd libpng-1.4.3
```

It's best to add the files to git *exactly as received* before doing anything else:

```
$ git init
$ git add *
$ git commit
```

and then in the editor, add a commit message something like:

```
New checkout libpng-1.4.3

Retrieved from http://www.libpng.org/pub/png/libpng.html, specifically
http://prdownloads.sourceforge.net/libpng/libpng-1.4.3.tar.gz?download
```

Of course, if this is a new and original checkout, there won't be any content, and this stage will be omitted.

Then we will need a muddle Makefile - it's best to add this as a separate transaction, to make it clear it is not part of the original code:

```
$ touch Makefile.muddle
$ git add Makefile.muddle
$ git commit Makefile.muddle -m 'A library needs a muddle Makefile'
```

The content of the muddle Makefile can be sorted out later.

The new library also needs to be added to the build description, or muddle won't know about it. So, in this case, we edit src/builds/01.py and add:

```
# This is the version of the PNG library that works with our other code
make.twolevel(builder, name='libpng', roles=[graphics_role],
              co_dir='libs', co_name='libpng-1.4.3')
```

in an appropriate place.

Note that the package name does not include the version number - again, this makes it easier if we change the version of `libpng` we are using.

We can now use:

```
$ muddle query checkouts
```

to make sure the checkout has been succesfullly added to the build description. If it has, then `libpng-1.4.3` should appear in the list that prints out.

Finally, we tell muddle that we have already checked this library out:

```
$ muddle import libpng-1.4.3
```

and we can then push the contents of our new checkout to the server:

```
$ muddle push libpng-1.4.3
```

The remaining tasks are:

1. Add some dependencies on the library to the build description (there must be something that depends on it, or we'd not have included it).
2. Write the actual muddle Makefile for it.
3. Try building it with `muddle build libpng`.

5.3 How I use terminal windows with muddle

In a GUI environment (e.g., Gnome or whatever), I personally tend to have several terminal windows open:

- One at the top level of the build tree.

  This is where I issue commands like `muddle pull _all`, or `muddle query`, or even `visdep`.

- One for each checkout I'm working on, with the terminal `cd`-ed into the appropriate source directory.

  In these terminals, I can do `muddle` or `muddle distrebuild` or whatever without needing to specify the checkout or package I'm working on. This also facilitates using the appropriate VCS commands directly, as discussed above, or doing `muddle push` when I believe I've reached an appropriate stage.

5.4 Making a clean binary deployment/release

There are two approaches to this.

5.4.1 Using a clean build tree

```
$ cd <where the build trees live>
$ mkdir <new build tree directory>
$ cd <new build tree directory>
$ muddle init <repo> <desc>
$ muddle
```

or possible `muddle deploy _all` or whatever is appropriate.

`<where the build trees live>` is the directory that `<new build tree directory>` is going to be created in - in my work environment, I segregate these by project, but it may be as simple as a directory called `~/work`. The important thing is that each build tree is a separate directory, with an appropriate name (perhaps by date).

`<repo>` and `<desc>` are normally known for a particular project (they may be documented on a wiki page), but are also to be found in:

- `.muddle/RootRepository`
- `.muddle/Description`

in a previous build tree.

This approach is always safe, but can take much more time (since the entire build tree needs to be checked out again).

5.4.2 Using an updated and "very clean" build tree

```
$ muddle pull _all
$ muddle veryclean
$ muddle
```

or possible `muddle deploy _all` or whatever is appropriate.

This approach is quicker, as it does not involve checking everything out again.

The `muddle veryclean` is to ensure that the `obj`, `install` and `deploy` directories are cleanly removed. Over time, during development, it is common for unused software and files to build up in theses directories - checkouts change the files they produce, or produce differently named files - and the `muddle distclean` operation does

what the `distclean:` target in a muddle Makefile says it should do, which may not clean things as thoroughly as one might wish.

The disadvantage of this approach is that, for some projects, there can be problems with not building entirely from scratch. In particular:

- If any of the packages actually do build or alter stuff in their checkout directory, then the build tree may not be as clean as one thought (this is why we strongly recommend out-of-tree building for all packages).
- *In older versions of muddle*: If there are any shallow checkouts (typically specified in the build description as follows:

```
pkg.set_checkout_vcs_option(builder, Label('checkout', 'kernel-source'),
                            shallow_checkout=True)
```

  where the `Label` part may, of course, differ), then these will not be updated by `muddle pull`, so it is possible to miss important changes.

CHAPTER 6

Muddle and weld - how to use muddle with weld

Muddle's normal workflow involves having one checkout per package. But sometimes, particularly in complex systems with limited sharing, you will want to store all your software in a single git repository so that bug fixes that cross packages are easier to track.

Under these circumstances you can use a tool called weld, written by some of the same people who wrote muddle - http://code.google.com/p/weld .

Weld is essentially an automated tool for managing vendor branches in git.

Muddle provides some limited facilities to allow the git repositories created by weld (known as welds) to be built using muddle.

If you are interested in using weld, now is a good time to go and read the documentation that comes with weld.

It does this by registering a special version control system called weld, which is always up to date. In future, muddle may well provide the ability to generate weld specifications from its build description or vice versa but for now, the build description and the weld specification are separate entities.

To use muddle with weld:

1. Check out the weld as usual using git, into say, `mydir`.
2. Create a .muddle directory using `cd mydir; muddle init weld+ssh://some/origin builds/01.py` - assuming that your builds are in `mydir/src/builds` and that you want to use build description `01.py`
3. Use muddle as normal

You will be unable to use the muddle revision control commands (`muddle push`, `muddle pull` etc.); this is in keeping with the weld idiom that you should use git commands to accomplish version control in welds and just use weld for vendor branch management.

CHAPTER 7

The muddle command line

This is very much a work-in-progress

7.1 How to find out about the command line

Muddle itself provides help on how to use it as a command line tool.

Specifically, `muddle help` is a useful place to start.

7.2 Special command line arguments

Muddle reserves "label names" starting with an underscore for itself (so you should never create a checkout, package or deployment name with an underscore).

The list of "special" names may be added to at any time, but at the time of writing is:

- `_all`
- `_all_checkouts`, `_all_packages`, `_all_deployments`
- `_default_roles`
- `_default_deployments`
- `_just_pulled`
- `_release`

Many muddle commands support using these as if they were standard labels, and will then expand them to the implied labels. Those commands which do support them all say so in their help text.

`muddle help labels` gives a brief explanation of each of these.

7.2.1 `_all`

This expands to all of the target labels defined in the build description *of the appropriate type* for the command.

A typical use of `_all` would be:

```
$ muddle pull _all
```

where it means all checkouts.

7.2.2 _all_checkouts, all_packages, all_deployments

These expand to all the target labels defined in the build description of the specific type.

So the previous example could also be specified (more explicitly) as:

```
$ muddle pull _all_checkouts
```

7.2.3 _default_roles

This expands to the package labels for all of the default roles, as given in the build description. Specifically, `package:*{<role>}/postinstalled` for each such `<role>`. You can find out what the default roles are with `muddle query default-roles`.

`default_roles` is mainly provided to allow users to understand and reproduce the effect of `muddle` at the top level of the build tree, which is equivalent to:

```
$ muddle buildlabel _default_deployments _default_roles
```

7.2.4 _default_deployments

This expands to the deployment labels for each of the default deployments, as given in the build description. You can find out what the default deployments are with `muddle query default-deployments`.

`default_deployments` is mainly provided to allow users to understand and reproduce the effect of `muddle` at the top level of the build tree, which is equivalent to:

```
$ muddle buildlabel _default_deployments _default_roles
```

7.2.5 _just_pulled

This expands to the checkouts that were (actually) pulled by the last `muddle pull` or `muddle merge` command. The current value is always stored in the file `.muddle/_just_pulled`. If that file doesn't exist, then nothing has yet been pulled/merged in the current build tree.

Note: The labels in the _just_pulled file are stored one per line, in label-sort order (so the order is predictable).

`muddle pull` and `muddle merge` both:

1. clear the `_just_pulled` file,
2. then do the pulling or merging,
3. and then when they have finished, update the `_just_pulled` file.

Note: Just because a checkout has been updated, it is not *necessarily* the case that it needs rebuilding. For instance, documentation changes might not be relevant to the muddle build, and changes to unused source code are obviously not relevant.

A typical use of `_just_pulled` would be:

```
$ muddle pull _all
$ muddle distrebuild _just_pulled
```

7.2.6 _release

This expands to the targets specified in the build description with `builder.add_to_release_build()`. It's definition and use is described in the chapter on "The muddle release mechanism".

CHAPTER 8

Repositories

The definition of the Repository class, and various other useful functions, are in `muddled.repository.py`. Use `muddle doc muddled.repository` for more information.

8.1 Checkouts and repositories

Muddle separates the concerns of the *checkout*, the local copy of some source code, and the *repository*, the non-local place where the (or an) original of the source code is stored, and from whence the checkout was cloned or checked out.

Note: Earlier versions of muddle (before v2.3) did not make this separation as clearly. Things are better now...

In most build descriptions, this separation is actually de-emphasised. Particularly, most build descriptions work with semi-local repositories - that is, rather than referring to the original (somewhere on the internet) repositories for software such as Linux, busybox, zlib, etc., a local set of repositories is maintained, containing copies of the original repositories. This has the advantage that instantiating a new build is not dependent on a long list of remote network sites being up and available.

Thus the typical muddle build does its:

```
muddle init <vcs>+<base-repo> builds/01.py
```

which expects to find the build description in a repository of type <vcs> (e.g., git) located at `<base-repo>/builds`. Other checkouts are then likely to be retrieved from repositories whose type is the same, and whose repository URLs also start with `<base-repo>`.

So, for instance, the convenience call:

```
from muddled.pkgs import make

def describe_to(builder):
  ...
  make.medium(builder, "first_pkg", ['x86'], "first_co")
```

does quite a lot:

1. It says that `package:first_pkg{x86}` is built from `checkout:first_co`, using (by default) a `Makefile.muddle` to be found in that checkouts source directory.
2. It says that `checkout:first_co` has its source in `src/first_co` (because no subdirectory of `src/` has been specified).

3. It says that `checkout:first_co` is of type `<vcs>` (i.e., defaulting to the same VCS as used for the build description), and is cloned from `<base-repo>/first_co` (again, with defaults taken from the build description).

The writer of the build description could instead have been more explicit about much of that:

```
from muddled.depend import checkout
from muddled.pkgs import make
from muddled.repository import get_checkout_repo, add_upstream_repo
from muddled.version_control import checkout_from_repo

def describe_to(builder):
  ...
  build_desc_repo = builder.build_desc_repo
  co_repo = build_desc_repo.copy_with_changes('first_co')

  co_label = checkout('first_co')
  checkout_from_repo(builder, co_label, co_repo)
  muddled.pkgs.make.simple(builder, 'package1', 'x86', co_label.name)
```

although there's not normally any particular gain in doing so.

8.2 Upstream repositories

8.2.1 What upstream repositories are and why we need them

Consider a library (such as zlib). We can generally find a repository that contains the current version of the library, and tell muddle to use that to retrieve the package.

But it is often more convenient to take our own copy - it means that we have direct control over the exact version (yes, we could use a revision id to do that with a remote repository), it means we can add a `Makefile.muddle` directly to the source without worrying about accidentally sending it back (although we could use a branch or even a separate "helper" checkout for that), and it also means we can make our own patches as needed (again, branching would work).

The main thing, though, is that we're insulated against that far repository being down just when we need it (and the more checkouts we're dealing with, the more likely this is - even Google Projects occasionally goes down).

However, what if we do make a useful patch to the library, and want to submit it back to the original repository? Or if we want to update our near repository because the far repository has changed? This is where the concept of an "upstream" repository comes in - it is a farther away (in some sense) repository that some or part of our near repository can synchronise with.

Linux development uses this concept all of the time, and it is typical there to have multiple upstreams.

So, the idea is that we have a "near" repository, which is the one we clone our checkout from, and push our own changes to. And we may also have one or more upstream repositories, which we can optionally pull from or, perhaps, push to.

8.2.2 Declaring upstream repositories

This is done in the build description.

For instance:

```
from muddled.depend import checkout
from muddled.pkgs import make
from muddled.repository import Repository, get_checkout_repo, add_upstream_repo
```

```
def describe_to(builder):
  ...
  # Declare package "first_pkg" that is build from checkout "first_co",
  # which is retrieved from a repository related to the build description
  # repository
  make.medium(builder, "first_pkg", [role], "first_co")

  # We get the *actual* repository with:
  co_repo = get_checkout_repo(builder, checkout("first_co"))

  # And we can then add some upstreams
  repo1 = co_repo.copy_with_changes('repo1.1')
  # We are not allowed to push to this second repository
  repo2 = Repository.from_url('git', 'http://example.com/repos/first_co',
                              push=False)
  add_upstream_repo(builder, co_repo, repo1, ['rhubarb', 'wombat'])
  add_upstream_repo(builder, co_repo, repo2, 'rhubarb')
```

or, replacing the `make.medium` with lower level calls, we could have done:

```
from muddled.depend import checkout
from muddled.pkgs import make
from muddled.repository import Repository, get_checkout_repo, add_upstream_repo
from muddled.version_control import checkout_from_repo

def describe_to(builder):
  ...
  # This shows that our repository is explicitly related to the
  # build description repository...
  co_repo = builder.build_desc_repo.copy_with_changes('first_co')

  co_label = checkout('first_co')
  checkout_from_repo(builder, co_label, co_repo)
  muddled.pkgs.make.simple(builder, 'package1', role, co_label.name)

  # And we can then add some upstreams
  # - this bit is just the same as the previous example
  repo1 = co_repo.copy_with_changes('repo1.1')
  # We are not allowed to push to this second repository
  repo2 = Repository.from_url('git', 'http://example.com/repos/first_co',
                              push=False)
  add_upstream_repo(builder, co_repo, repo1, ['rhubarb', 'wombat'])
  add_upstream_repo(builder, co_repo, repo2, 'rhubarb')
```

Note:

1. Upstream names must be formed of A-Z, a-z, 0-9, underscore (_) and hyphen (-). This is mainly to ensure interoperability with VCS like git, where we want upstream names to also work as remote names - see *Upstream repositories and git* below.
2. Each upstream repository must have at least one name.
3. For your convenience, `add_upstream_repo` allows its final argument to be a single upstream name, or a list of such.

8.3 The commands

There are several muddle commands related to upstream repositories:

- `muddle push-upstream`
- `muddle pull-upstream`
- `muddle query upstream-repos`

See the `muddle help` text on each for detailed information.

Note that the current versions of all these commands assume wide terminals to display their output. This may possibly change in the future.

8.3.1 muddle query upstream-repos

This commnand reports either all the upstream repositories known, or those for a particular checkout. For instance:

```
$ muddle query upstream-repos
> Upstream repositories ..
Repository('git', 'http:/example.com/repo/main', 'repo1') used by checkout:(subdomain1)co_repo1/*, ch
    Repository('git', 'http:/example.com/repo/main', 'repo1.1')  rhubarb, wombat
    Repository('git', 'http:/example.com/repo/main', 'repo1.2', push=False)  insignificance, wombat
    Repository('git', 'http:/example.com/repo/main', 'repo1.3', pull=False)  platypus, rhubarb
```

The description of the near repository is followed by a list of those checkouts using it.

The description of each upstream repository is followed by a list of the corresponding upstream names.

Note that some of the repositories listed may not be pushed to (`push=False`) and some may not be pulled from (`pull=False`).

Note: It is quite possible to (a) set an upstream repository for a repository that is not used by any checkout, and (b) set an upstream repository for an upstream repository. Muddle will make no use of either of these cases (although it is *possible* that some future version of muddle might allow upstreams of upstreams).

The `-u` or `-url` switch may be used to view repositories as their URL, rather than as their constructor:

```
$ m3 query upstream-repos -u
> Upstream repositories ..
http:/example.com/repo/main/repo1 used by checkout:(subdomain1)co_repo1/*, checkout:co_repo1/*
    http:/example.com/repo/main/repo1.1  rhubarb, wombat
    http:/example.com/repo/main/repo1.2  insignificance, wombat
    http:/example.com/repo/main/repo1.3  platypus, rhubarb
```

This does not, however, show the value of any `push` or `pull` flags, nor would it should branch or revision information.

You can also ask for upstream repositories that relate only to a single checkout, for instance:

```
$ muddle query upstream-repos co_repo1
Repository('git', 'http:/example.com/repo/main', 'repo1') used by checkout:co_repo1/checked_out
    Repository('git', 'http:/example.com/repo/main', 'repo1.1')  rhubarb, wombat
    Repository('git', 'http:/example.com/repo/main', 'repo1.2', push=False)  insignificance, wombat
    Repository('git', 'http:/example.com/repo/main', 'repo1.3', pull=False)  platypus, rhubarb
```

In this instance, the first line of output only lists the requested checkout, although we know from the previous example that this is not the only checkout using that repository.

8.3.2 muddle pull-upstream

Pulling from upstream repositories is done with a different command than pulling from the "near" repository, partly because it is likely to be a less common operation.

The command line is broadly:

```
muddle pull-upstream [ <checkout> ... ] -u[pstream] <name> ...
```

where `<checkout>` is a label fragment in the normal way, and `<name>` is one of the names registered for an upstream repository.

So:

```
$ muddle pull-upstream package:android{x86} -u upstream-android
```

says to pull from the upstream repositories for all the checkouts used by the `android{x86}` package, using those upstream repositories with name `upstream-android`.

Similarly:

```
$ muddle pull-upstream builds co_repo1 -upstream wombat rhubarb
```

asks to pull from any upstreams called `rhubarb` or `wombat` into checkouts `builds` and `co_repo1`.

The command works by looking at each checkout in turn, and:

1. finding the "near" repository for the checkout
2. findings its upstreams (if any)
3. determining which of those have any of the names requested
4. pulling from each of those in turn

If there is no appropriate upstream repository for a particular checkout, then this is reported, but is not an error. If there is more than one upstream repository for a checkout, the order of pulling from them is not defined - if this matters, then it is up to the user to do the process by hand.

Unlike the normal `pull` command, there is no `-stop` switch. If any of the constituent `pull` operations fails, then the whole `pull-upstream` command will fail at that point.

As with other "action" commands, `muddle -n pull-upstream` can usefully be used to find out what it would do, without actually doing it. However, it does not check whether a repository allows pull.

8.3.3 muddle push-upstream

Pushing to upstream repositories is done with a different command than pushing to the "near" repository, partly because it is likely to be a less common operation.

The command line is broadly:

```
muddle push-upstream [ <checkout> ... ] -u[pstream] <name> ...
```

where `<checkout>` is a label fragment in the normal way, and `<name>` is one of the names registered for an upstream repository.

So:

```
$ muddle push-upstream package:android{x86} -u upstream-android
```

says to push to the upstream repositories for all the checkouts used by the `android{x86}` package, using those upstream repositories with name `upstream-android`.

Similarly:

```
$ muddle push-upstream builds co_repo1 -upstream wombat rhubarb
```

asks to push to any upstreams called `rhubarb` or `wombat` from checkouts `builds` and `co_repo1`.

The command works by looking at each checkout in turn, and:

1. finding the "near" repository for the checkout
2. findings its upstreams (if any)
3. determining which of those have any of the names requested
4. pushing to each of those in turn

If there is no appropriate upstream repository for a particular checkout, then this is reported, but is not an error. If there is more than one upstream repository for a checkout, the order of pushing to them is not defined - if this matters, then it is up to the user to do the process by hand.

Unlike the normal `push` command, there is no `-stop` switch. If any of the constituent `push` operations fails, then the whole `push-upstream` command will fail at that point.

As with other "action" commands, `muddle -n push-upstream` can usefully be used to find out what it would do, without actually doing it. However, it does not check whether a repository allows push.

8.3.4 Upstream repositories and git

Git already has its own support for "upstreams", via remotes. Muddle attempts to work with this in a reasonably approriate manner.

When doing a `muddle push-upstream`, muddle will actually do:

```
git config remote.<name>.url <repo>
git push <name> <branch>
```

That is, it will attempt to setup a remote for each upstream repository, and then push to it using that remote name, where `<repo>` is the appropriate upstream repository, and `<name>` is its upstream name. As normal, `<branch>` defaults to `master`.

If there is more than one upstream name for a particular repository, the first (when the names are sorted in C order) will be used.

If more than one repository has the same `<name>` (according to that rule), then this means that the last repository pushed to (with that name) will be remembered with that remote name.

Similarly, when doing a `muddle pull-upstream`, muddle will actually do:

```
git remote rm <name>
git remote add <name> <repo>
git fetch <name>
git merge --ff-only remotes/<name>/<branch>
```

for each upstream repository (actually, it first checks to see if the remote name is already defined, and if it is not, omits the `git remote rm` command).

If you want to use this interaction (between muddle upstream repositories and git remotes) fully, then it is probably a good idea to ensure that each upstream for a given checkout has a single `<name>` that is unique for that checkout - this will allow the git remotes and the upstreams to be named the same.

8.3.5 Upstream repositories and subdomains

Mostly, it hopefully Does The Right Thing.

However, if we have a checkout in the main domain that uses a particular repository, and a checkout in a subdomain that uses the same repository, but adds an upstream to it that is not also added in the main domain, then muddle will give up with an error message. The thinking is that if the same repository is used in the main domain and a subdomain, but has different upstreams in the two, then a user might not notice this, and might end up accidentally pushing somewhere unexpected. The solution is that the repository in the main domain must have a superset of the upstreams that are used in any subdomains.

CHAPTER 9

Instruction files

See:

- `muddle help instruct`
- `muddle help query inst-files`
- `muddle help query inst-details`

for some basic help on instruction files, until I have time to write a better introduction. For the moment, this section is mainly here to document what goes *into* instruction files.

9.1 Where instruction files get put

Instruction files are stored in the `.muddle` directory, specifically as either:

- `.muddle/instructions/<package-name>/<role>.xml` or
- `.muddle/instructions/<package-name>/_default.xml`

The latter corresponds to instructions for role `{*}` for package `<package-name>`.

9.2 The content of instruction files

Instruction files are XML files. Here is a simple example:

```
<?xml version="1.0"?>
<instructions>
  <chown>
    <filespec>
      <root>/bin</root>
      <spec>hello_world</spec>
    </filespec>
    <user>root</user>
    <group>root</group>
  </chown>
  <chmod>
    <filespec>
      <root>/bin</root>
      <spec>hello_world</spec>
    </filespec>
    <mode>ugo+rx</mode>
```

```
  </chmod>
</instructions>
```

and here is an example with more comments:

```
<?xml version="1.0"?>

<!-- Filesystem for a Linux with busybox - the fiddly bits -->

<instructions>

  <!-- There's something to be said for making all files be owned by
       root (it makes the system look tidier), but on the other hand
       it involves changing *all* files -->
  <!--
  <chown>
    <filespec>
      <root>/rootfs</root>
      <spec>.*</spec>
      <all-under />
    </filespec>
    <user>0</user>
    <group>0</group>
  </chown>
  -->

  <!-- Certain things *must* be set executable -->
  <chmod>
    <filespec>
    <root>/rootfs/etc/init.d</root>
      <spec>rcS</spec>
    </filespec>
    <mode>0755</mode>
  </chmod>

  <!-- Everyone needs access to /tmp -->
  <chmod>
    <filespec>
      <root>/rootfs</root>
      <spec>tmp</spec>
    </filespec>
    <mode>01777</mode>
  </chmod>

  <!-- Traditionally, this is the only device node we *need* -->
  <mknod>
    <name>rootfs/dev/console</name>
    <uid>0</uid>
    <gid>0</gid>
    <type>char</type>
    <major>5</major>
    <minor>1</minor>
    <mode>0600</mode>
  </mknod>

</instructions>
```

9.2.1 Summary

The file is of the form:

```
<?xml version="1.0"?>
<instructions priority=100>
 <instr-name>
   <stuff .. />
 </instr-name>
</instructions>
```

where `<instr-name>` is a valid instruction specification (see below).

The `priority` attribute on `<instruction>` is optional. If it is present, then it is used by deployments to decide what order to apply instructions in. Higher priority values cause the instructions to be applied later (so instructions with `priority=10` will be applied before those with `priority=20`).

9.2.2 Standard instructions

There are currently three standard instructions available.

These are *descriptions* of the actual instruction to be applied. It is up to the deployment tool that uses the instructions to decide what to actually do. For instance, `filedep` will run `chown`, `chmod` or `mknod` directly, whilst `cpiofile` will simply emit instructions to the CPIO file it is constructing to do the appropriate thing when the CPIO file is "unpacked".

chown

This causes the `chown` program to be run, to set the ownership (user and group) for a file. For instance:

```
<chown>
  <filespec>
    <root>/bin</root>
    <spec>hello_world</spec>
  </filespec>
  <user>root</user>
  <group>root</group>
</chown>
```

Internal tags are:

- `<filespec>` which contains:
  - `<root>` - the location of the file on the target filesystem, or the location relative to `${MUDDLE_INSTALL}` (these are substantially the same).
  - `<spec>` - its filename
- `<user>` - the name of the user that should own it. This is optional. If not specified, the value will not be changed.
- `<group>` - the name of the group it should be in. This is optional. If not specified, the value will not be changed.

chmod

This causes the `chmod` program to be run, to set the permissions for a file. For instance:

```
<chmod>
  <filespec>
    <root>/bin</root>
    <spec>hello_world</spec>
  </filespec>
  <mode>ugo+rx</mode>
</chmod>
```

Internal tags are:

- <filespec> which contains:
  - <root> - the location of the file on the target filesystem, or the location relative to ${MUDDLE_INSTALL} (these are substantially the same).
  - <spec> - its filename
- <mode> - the required permissions, specified in a manner that chmod will understand.

mknod

This causes mkdnod to be run to create a device node. For instance:

```
<mknod>
 <name>/lib/udev/devices/console</name>
 <uid>0</uid>
 <gid>0</gid>
 <type>char</type>
 <major>5</major>
 <minor>1</minor>
 <mode>0600</mode>
</mknod>
```

Internal tags are:

- <name> - the path of the device node to create.

  In early versions of muddle, this was an absolute path, and if a leading / was used, then an attempt to write to the local filesystem was likely.

  In current versions of muddle, leading / characters will be removed, rendering this relative to the target filesystem.
- <uid> - the user id to use for it
- <gid> - the group id to use for it
- <type> - the type of device node. char means c, a charecter device, block means b, a block device.
- <major> and <minor> - the major and minor device numbers. These may be in decimal, in hexadecimal (starting with 0x) or octal (starting with 0 but not x).
- <mode> is the permissions to be used for the device. These must be a "umask" - i.e., an octal value specifying the permissions. See "man chmod" for help on this.

CHAPTER 10

Project lifecycle support: how to manage maintenance branches

10.1 Summary

The project lifecycle commands allow for creating a set of maintenance branches on every checkout in a build tree.

Let us assume that we have a build tree with build name "Widget", and that release version 1.0 of the Widget product has just been delivered. We want to continue working towards version 2 of the product in the "normal" build tree, but we may also need to make maintenance releases based on version 1.0 of the code.

Muddle allows us to branch all of the checkouts in the build tree, so that we can do the v1.0 maintenance work on its own branch. Thus:

```
$ muddle branch-tree Widget-v1.0-maintenance
```

will perform various checks, and then create a new branch called `Widget-v1.0-maintenance` in each checkout.

Note: It is worth naming maintenance branches carefully so that they are unlikely to clash with any existing branchs in any of the checkouts. Using the build name, the version number, and some additional descriptive text (here, just "maintenance") seems sensible.

Note: `muddle branch-tree -check` will just do the checks, and `muddle branch-tree -force` will just do the branching, without any checks. See *Coping with problems in muddle branch-tree* below

We then need to tell the (branched) build description that all its checkouts should "follow" its branch (we'll see why further below). So we add the line:

```
$ builder.follow_build_desc_branch = True
```

to the build description (normally in `src/builds/01.py`).

We now need to commit and push all of these changes. Unfortunately, we don't yet have a way of doing a "muddle commit" that specifies a commit message, so the best we can do (assuming everything is using git) is:

```
$ muddle runin _all_checkouts 'git commit -a -m "Create maintenance branch"'
```

and then:

```
$ muddle push _all
```

Since the checkouts are on the branch requested by the build description (because we set the "follow" flag), this will push the new branch to the remote repositories, even though it didn't exist before.

At which stage, our remotes all have the branch, and our checkouts are still on it. That's probably a good time to record a stamp file:

```
$ muddle stamp version
$ muddle stamp push
```

Branching the build tree doesn't branch the "versions/" directory, but because we have set "follow" in the build description, the stamp file created will be called:

```
versions/<build_name>.<branch_name>.stamp
```

instead of the normal `versions/<build_name>.stamp` - so in this case:

```
versions/ExampleBuild.Widget-v1.0-maintenance.stamp
```

We can now change to a new, empty directory, and make a new copy of our maintenance build tree:

```
$ cd ..
$ mkdir Widget-v1.0-maintenance
$ cd Widget-v1.0-maintenance

$ muddle init -branch Widget-v1.0-maintenance <repo> builds/01.py
```

(where `<repo>` is the remote repository).

This gets the build description and automatically checks out the requested branch. If we look in the `.muddle/` directory, we can see that there is a new file:

```
.muddled/DescriptionBranch
```

which contains the name of the branch used at "muddle init" (it will only be present if "-branch" was used, and it is never altered by muddle itself).

We can now do:

```
$ muddle checkout _all
```

and, because the build description has "follow_build_desc_branch" set, all the checkouts will also be checked out with that branch.

Note: If we had not set `builder.follow_build_desc_branch = True`, then the other checkouts would just have their "normal" branch, as defined by the build description. It is perfectly legitimate to use `muddle init -branch` to retrieve a branch of the build description for other purposes, not to do with whole build tree branching.

You can use:

```
$ muddle query checkout-branches
```

to see the details of the branches being used, and why. For instance:

```
$ m3 query checkout-branches
--------  ----------------------  ----------------------  ----------------------
Checkout  Current branch          Original branch         Branch to follow
--------  ----------------------  ----------------------  ----------------------
builds    Widget-v0.1-maintenance  Widget-v0.1-maintenance  <it's own>
co1       Widget-v0.1-maintenance  master                  Widget-v0.1-maintenance
```

"Original branch" is the branch as requested by the build description. Normally that will be "master" - i.e., the default branch - but sometimes a build description specifies a particular branch for a checkout.

If a build description *does* specify a particular branch (or revision) for a checkout, or if a checkout is not using git, then `muddle branch-tree` will not be able to branch it for you, and you will have to decide what to do about that checkout in this context, a maintenance tree.

10.2 A more detailed look

10.2.1 Why do we need this?

Let us assume that we have been developing a build tree, and we reach the point of releasing version 1 of the product that it builds. We can do the normal steps of generating stamp files (`muddle stamp`), perhaps producing source releases (`muddle distribute`), and quite likely producing an actual release tarball (`muddle release`), but then we want to continue development towards the next version of the product.

The problem is that we almost certainly also want to be able to go back and produce later "bugfix" releases based on version 1, and ideally we would like muddle to make this easy for us.

One rather cumbersome way to do this would be make a branch of our build description describing, explicitly, what is in release 1. So we could look at the stamp file we produced for our release (of course we made a stamp file), and for each checkout, explicitly specify its revision in the build description. If we committed that (on its branch), then we could later build our tree again, making sure we're using the branch of the build description. If we needed to fix a bug in a particular package, we could then branch that package, determine the new revision, update the build description, and so on.

This would be extremely tedious, and definitely error prone.

A slightly (but only slightly) better approach would still specify all of the revisions "by hand", but when a package was changed for a bugfix, we would perofrm the change on an appropriate branch, and amend the build description for that package to specify this new branch instead of the revision.

Which is really not that much better.

So instead we provide a mechanism to branch all checkouts in a build tree, and to use the same branch (i.e., branch name) for the build description and the checkouts.

Note: This assumes that all the checkouts support this mechanism. At time of writing, muddle supports this for git. If most checkouts use git, and a few use a VCS that is not supported for this purpose, then one of the "solutions" above may be adopted for the less able VCS checkouts.

10.2.2 How we do it

This means we would:

1. Decide we're going to make release 1. If our build is called "Widget", we might thus decide to use a branch called "Widget-v1.0-maintenance".

   (It seems sensible to put the name of our build tree into the branch name as a way of avoiding clashes with any previous branches in the individual checkouts. It is quite likely that some checkout of source code from elsewhere might have already have a branch called "v1.0", for instance. Whilst muddle can warn if this is the case, it is ultimately up to the user to choose an appropriate branch name.)

2. We then use:

```
$ muddle branch-tree Widget-v1.0-maintenance
```

   to make various checks, and then branch all the checkouts.

3. We then edit the build description (which is now on the new branch) and add a line to the `describe_to()` function:

```
builder.follow_build_desc_branch = True
```

   This tells muddle that checkouts should use the same branch as the build description. Of course, if the build description explicitly specifies a revision or branch for a checkout, then that takes precedence, so please remember to check.

   A build description with this value set is described as having set "following", and the build tree "follows" the build description.

   Note: Muddle tries to be a little bit helpful by also recognising the (mistaken):

```
builder.follows_build_desc_branch = True
```

   and treating that as an error - otherwise the nature of Python would allow this, with no effect.

   ---

4. We then need to commit everything. We really need a:

```
$ muddle commit _all -m <message>
```

   but we don't have one yet, but we can do:

```
$ muddle runin _all_checkouts 'git commit -a -m "Create maintenance branch"'
```

   if we are careful with the quoting. Of course, if any of our checkouts are not using git, that's going to search up through the directories looking for a `.git` directory, which can sometimes lead to surprising effects, so be a little bit careful.

5. We can then do:

```
$ muddle push _all
```

   to push the build description and all the branched checkouts.

6. As one might expect, it's then generally a good idea to take a stamp:

```
$ muddle stamp version
$ muddle stamp push
```

   Since our build description sets "following", the version stamp file will be called:

```
versions/<build-name>.<build-desc-branch>.stamp
```

   The "versions/" directory is *not* branched by `muddle branch-tree`, since it is meant to contain all the version stamps for the project. Thus naming the stamp file with the branch name (which should indicate the purpose of the branch) is a sensible solution.

If anyone does a `muddle init` of the build tree, they will still get the normal ("master") branch, which we have not altered, and can `muddle checkout` and continue mainstream development just as normal.

However, if we want to do a bugfix on version 1, we can do:

```
$ muddle init -b Widget-v1.0-maintenance <repo> src/builds
$ muddle checkout _all
```

where `<repo>` is the same repository spec as used in the original `muddle init` (and remembered in `.muddle.RootRepository`).

This will checkout the build description on the named branch, and then, because the build description sets "follow", will checkout the same branch of each checkout.

We can use:

```
$ muddle query checkout-branches
```

to show that the correct branches are checked out (see `muddle help query checkout-branches` for what the columns output by that command mean).

We can also use `muddle query build-desc-branch` to see if we are "following":

```
$ muddle query build-desc-branch
Build description checkout:builds/* is on branch Widget-v1.0-maintenance
  This WILL be used as the default branch for other checkouts
```

One last useful command, introduced as part of the support for branched builds, is `muddle sync`. This is used to put the branch of a checkout (or checkouts) back to what the build description says it should be. This can be useful if you've been switching branches to look at alternative implementations of a checkout. How it actually decides what to do is slightly complex - see `muddle help sync` for details.

10.2.3 A worked example

This is basically following an example in the `tests/test_lifecycle.py` test script.

Imagine that we have done:

```
$ muddle init git+<repo> builds/01.py
```

and the build description `01.py` looks like:

```
def describe_to(builder):
    builder.build_name = 'ExampleBuild'
    add_package(builder, 'package1', 'x86', co_name='co1')
    # ... more normal packages ...
    add_package(builder, 'package2', 'x86', co_name='co.branch1',
                branch='branch1')
    add_package(builder, 'package3', 'x86', co_name='co.fred',
                revision='fred')
    add_package(builder, 'package4', 'x86', co_name='co.branch1.fred',
                branch='branch1', revision='fred')
    add_bzr_package(builder, 'package5', 'x86', co_name='co.bzr')
```

where `add_package` and `add_bzr_package` do appropriate things, and the necessary `import` statements have been omitted.

We have several checkouts:

- "co1" is a "normal" checkout - we haven't specified an explicit branch or revision, and are expecting to work with HEAD of master. We would normally expect to have more of these, as indicated by the comment.
- "co.branch1" specifies an explicit branch.
- "co.fred" specifies an explicit revision - in this case, a tag name.
- "co.branch1.fred" specifies a branch and a revision. When cloning it out, muddle will first checkout the branch, and then go to the revision if it is not already there (i.e., if it is not HEAD of the branch).
- Finally, "co.bzr" is a checkout using Bazaar, taken from some other repository.

We'll assume that the user has done:

```
$ muddle checkout _all
```

and probably built everything, to check that the build tree works.

We can try branching the tree, but this will fail as follows:

```
$ muddle branch-tree Widget-v1.0-maintenance

Unable to branch-tree to Widget-v1.0-maintenance, because:
  checkout:co.branch1.fred/checked_out explicitly specifies revision "fred" in the build description
  checkout:co.branch1/checked_out explicitly specifies branch "branch1" in the build description
  checkout:co.bzr/checked_out uses bzr, which does not support lightweight branching
  checkout:co.fred/checked_out explicitly specifies revision "fred" in the build description
```

all of which make sense, if one looks at the build description.

As discussed in *Coping with problems in muddle branch-tree* (below), our best option in this particular case is to force a branch, aiming to fix things afterwards:

```
$ muddle branch-tree -force Widget-v1.0-maintenance
> git remote set-branches --add origin Widget-v1.0-maintenance
checkout:co.branch1.fred/checked_out explicitly specifies revision "fred" in the build description
checkout:co.bzr/checked_out uses bzr, which does not support lightweight branching
> git remote set-branches --add origin Widget-v1.0-maintenance
checkout:co.branch1/checked_out explicitly specifies branch "branch1" in the build description
checkout:co.fred/checked_out explicitly specifies revision "fred" in the build description
Successfully created  branch Widget-v1.0-maintenance in 2 out of 6 checkouts
Successfully selected branch Widget-v1.0-maintenance in 2 out of 6 checkouts
Unable to branch the following:
  checkout:co.branch1.fred/checked_out (specific revision fred)
  checkout:co.bzr/checked_out (VCS %s not supported)
  checkout:co.branch1/checked_out (specific branch branch1)
  checkout:co.fred/checked_out (specific revision fred)

If you want the tree branching to be persistent, remember to edit
the branched build description,
  /home/tibs/sw/m3/tests/transient/example/src/builds/01.py
and add:

  builder.follow_build_desc_branch = True

to the describe_to() function, and check it in/push it.
```

That reiterates the problems we already knew we had, and reminds us to amend the build description.

So, we edit the build description, doing the following:

1. Add the `builder.follow_build_desc_branch = True` line to the end.
2. Amend the Bazaar checkout, "co.bzr", to set the "no_follow" option (for a real project, we'd also change the repository it refers to, but muddle can't tell the diference, so we shall ignore that for now).

The build description now looks like:

```
def describe_to(builder):
    builder.build_name = 'ExampleBuild'
    add_package(builder, 'package1', 'x86', co_name='co1')
    # ... more normal packages ...
    add_package(builder, 'package2', 'x86', co_name='co.branch1',
                branch='branch1')
    add_package(builder, 'package3', 'x86', co_name='co.fred',
```

```
                revision='fred')
    add_package(builder, 'package4', 'x86', co_name='co.branch1.fred',
                branch='branch1', revision='fred')
    add_bzr_package(builder, 'package5', 'x86', co_name='co.bzr',
                    no_follow=True)
    builder.follow_build_desc_branch = True
```

where `add_bzr_package` is defined as:

```
def add_bzr_package(builder, pkg_name, role, co_name=None, revision=None, no_follow=False):
    base_repo = builder.build_desc_repo
    if co_name is None:
        co_name = pkg_name
    repo = Repository('bzr', base_repo.base_url, co_name, revision=revision)
    checkout_from_repo(builder, checkout(co_name), repo)
    muddled.pkgs.make.simple(builder, pkg_name, role, co_name)
    if no_follow:
        builder.db.set_checkout_vcs_option(checkout(co_name), 'no_follow', True)
```

We can check that branches are being handled as expected:

```
$ muddle query checkout-branches
---------------  ----------------------  ---------------  ------------------------
Checkout         Current branch          Original branch  Branch to follow
---------------  ----------------------  ---------------  ------------------------
builds           Widget-v1.0-maintenance <none>           <it's own>
co.branch1       branch1                 branch1          <none>
co.branch1.fred  <none>                  branch1          <none>
co.bzr           <not supported>         ...              ...
co.fred          <none>                  <none>           <none>
co1              Widget-v1.0-maintenance <none>           Widget-v1.0-maintenance
```

The build description is on the right branch, so is our "normal" checkout, "co1". We can also see that both of those are now following the build description branch, as we wanted.

As to the other checkouts:

- "co.branch1" still names a specific branch in the build description, which is thus its "original" and "current" branch. It is not "following" because of that.
- "co.branch1.fred" still names a specific revision and branch in the build description. The "current" branch being `<none>` means that it is on a detached HEAD. Again, it is not "following".
- "co.fred" still names (just) a specific revision, and thus it too has a "current" branch of `<none>`, and is not "following".
- "co.bzr" is using Bazaar, and muddle does not support branching it, so it says that it does not support it.

We can now commit - I'm only going to commit the two checkouts we've actually branched, not least because trying to do `git commit` in "co.bzr" will not generally end well:

```
$ muddle runin co1 'git commit -a -m "Create maintenance branch"'
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/co1
# On branch Widget-v1.0-maintenance
nothing to commit (working directory clean)
$ muddle runin builds 'git commit -a -m "Create maintenance branch"'
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/builds
[Widget-v1.0-maintenance 726189c] Create maintenance branch
 1 file changed, 34 insertions(+), 2 deletions(-)
```

(obviously I could have used `muddle commit` directly here, which would have opened an editor for me to type the commit comments).

And we can now push:

```
$ muddle push _all
> Building checkout:builds/changes_pushed[T]
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/builds
> git push origin HEAD
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 794 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///stuff/tibs/sw/m3/tests/transient/01.init.branch.repo/src/builds
 * [new branch]      HEAD -> Widget-v1.0-maintenance
> Building checkout:co.branch1/changes_pushed[T]
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/co.branch1
> git push origin HEAD
Everything up-to-date
> Building checkout:co.branch1.fred/changes_pushed[T]
Checkout co.branch1.fred in directory src/co.branch1.fred
is on branch None, but should be on branch branch1.
Use "muddle query checkout-branches" for more information.
Refusing to push. Use git directly if that is what you really meant
> Building checkout:co.bzr/changes_pushed[T]
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/co.bzr
> bzr push file:///stuff/tibs/sw/m3/tests/transient/01.init.branch.repo/src/co.bzr
No new revisions or tags to push.
> Building checkout:co.fred/changes_pushed[T]
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/co.fred
Failure pushing checkout:co.fred/changes_pushed[T] in src/co.fred:
This checkout is in "detached HEAD" state, it is not
on any branch, and thus "muddle push" is not alllowed.
If you really want to push, first choose a branch,
e.g., "git checkout -b <new-branch-name>"
> Building checkout:co1/changes_pushed[T]
++ pushd to /home/tibs/sw/m3/tests/transient/example/src/co1
> git push origin HEAD
Total 0 (delta 0), reused 0 (delta 0)
To file:///stuff/tibs/sw/m3/tests/transient/01.init.branch.repo/src/co1
 * [new branch]      HEAD -> Widget-v1.0-maintenance

The following problems occurred:

Checkout co.branch1.fred in directory src/co.branch1.fred
is on branch None, but should be on branch branch1.
Use "muddle query checkout-branches" for more information.
Refusing to push. Use git directly if that is what you really meant

Failure pushing checkout:co.fred/changes_pushed[T] in src/co.fred:
This checkout is in "detached HEAD" state, it is not
on any branch, and thus "muddle push" is not alllowed.
If you really want to push, first choose a branch,
e.g., "git checkout -b <new-branch-name>"
```

We can see the new branch being pushed to the "builds" and "co1" repositories. As expected, neither "co.branch1.fred" nor "co.fred" could be pushed.

If we make a version stamp:

```
$ muddle stamp version
```

we end up with an appropriately named file:

```
$ ls versions
ExampleBuild.Widget-v1.0-maintenance.stamp
```

which we should remember to push:

```
$ muddle stamp push
```

Since we've pushed our changes, we can create a new build using them:

```
$ cd ..
$ mkdir example2
$ cd example2
$ muddle init -branch Widget-v1.0-maintenance git+<repo> builds/01.py
```

Now, when we do:

```
$ muddle checkout _all
> Building checkout:co.branch1/checked_out
...
> Building checkout:co.branch1.fred/checked_out
...
> Building checkout:co.bzr/checked_out
...
> git checkout fred
...
> Building checkout:co1/checked_out
++ pushd to /home/tibs/sw/m3/tests/transient/example2/src
> git clone -b Widget-v1.0-maintenance file:///stuff/tibs/sw/m3/tests/transient/01.init.branch.repo/s
Cloning into 'co1'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.
> Make directory /home/tibs/sw/m3/tests/transient/example2/.muddle/tags/checkout/co1
```

we can see that "co1" is being checked out with the correct branch (I've left out the text for the other checkouts, but we'd have seen "co.branch1.fred" and "co.fred" ending up in detached HEAD states).

Thus:

```
$ muddle query checkout-branches
--------------  ----------------------  ----------------------  ----------------------
Checkout        Current branch          Original branch         Branch to follow
--------------  ----------------------  ----------------------  ----------------------
builds          Widget-v1.0-maintenance Widget-v1.0-maintenance <it's own>
co.branch1      branch1                 branch1                 <none>
co.branch1.fred <none>                  branch1                 <none>
co.bzr          <not supported>         ...                     ...
co.fred         <can't tell>            <none>                  <none>
co1             <can't tell>            <none>                  Widget-v1.0-maintenance
```

Note that this is *nearly* identical to the output in the original build tree, except that now the "Original branch" for the build description is the branch we requested on the `muddle init` command line.

Note: If we hadn't added the "no_follow" option to the Bazaar checkout, we would have had errors of the form:

```
The build description wants checkouts to follow branch 'Widget-v1.0-maintnance',
but checkout co.bzr uses VCS Bazaar for which we do not support branching.
The build description should specify a revision for checkout co.bzr,
or specify the 'no_follow' option.
```

whenever we tried to do any checkout-related operations.

10.2.4 Coping with problems in muddle branch-tree

`muddle branch-tree` moves all of the checkouts in the current build tree to a specified branch, if possible.

It does this in two phases:

1. Check that there are no problems.
2. Perform the branching.

The `-c` or `-check` switch makes it just do the first, and the `-f` or `-force` switch makes it just do the second.

The problems that might occur are as follows:

- A checkout uses a VCS that muddle does not know how to branch. Broadly, this means "anything but git".

  There are two sensible solutions for this. First "force" a branch of the build tree, and then edit the build description in one of the two following ways:

  1. Specify a particular revision for the checkout. This is sensible if the particular checkout is not going to be developed anymore on this new branch.
  2. Specify a new "branch" for the checkout (that is, a new repository location), and mark the checkout as having the "no_follow" option. This last tells muddle that it is OK not to try to make the checkout "follow" the build description. Muddle needs to be told this explicitly because it has no way of telling that this (new) Bazaar repository is not the same as the original.

  Note: The "no_follow" option is supported for both Bazaar and Subversion.

- The build description explicitly specifies a particular revision and/or branch for a checkout.

  If a specific revision is named, then this can generally be left as it is. Care must, of course, be taken if it is ever changed, but the specific revision was presumably named because it was not expected to change.

  If a specific revision and branch are named, this is essentially the same case.

  If a specific branch is named, but without a revision, then there are two possible solutions.

  In both cases, first "force" a branch of the build tree, and then edit the build description, to either:

  - add a specific revision (the current revision on the named branch seems most likely), or
  - branch the checkout, by hand, to the same branch that is being "followed" by everything else. Then either amend the build description to name that branch instead of the original branch, or remove the explicit branch entirely. The latter has the disadvantage that it is no longer clear that this checkout was being "special cased" with an explicit branch.

- A checkout is shallow - that is, it does not contain all of its history.

The best solution for this is probably to find the current revision id of the shallow checkout using `muddle query checkout-id`, and amend the build description to specify that exact revision. This can then be changed to later revision ids as necessary.

- A checkout already has a branch of the name you wanted to use.

  Either that's a true clash, and you need to come up with a new name for this branch, or it's because the tree was already branched earlier, and you are using `branch-tree` as a convenient way to move all the checkouts in a non-branched tree to the (already extant) branch.

10.2.5 Can I use `muddle branch-tree` to go to an existing branch?

As implied by the last item above (in *Coping with problems in muddle branch-tree*), this is perfectly possible - using the `-force` switch will cause it to overcome its worries that the branch already exists.

However, you're probably better off doing:

```
$ cd src/builds  # or wherever the top-level build description is
$ git checkout branch.to.follow
$ muddle sync _all
```

If the build description on branch "branch.to.follow" has the "follows" flag set, then this will produce the same result in a simpler fashion.

10.2.6 Muddle commands in branched build trees

In the following, I'm assuming that the build description sets "following", and is on a branch, and that the checkouts being discussed are using git (because muddle doesn't support branching in other version control systems).

In a branched build tree, the build description sets "following". A checkouts branch is determined as follows:

1. If a specific revision and a specific branch are specified in the build description, then muddle "goes to" that branch and then, if necessary changes to the specified revision. If the revision is not HEAD of the branch, this will leave the checkout on a detached HEAD (see http://alblue.bandlem.com/2011/08/git-tip-of-week-detached-heads.html for a useful explanation of this).
2. If a specific revision alone is given in the build description, then that will be used. This will result in a detached HEAD.
3. If a specific branch alone is given in the build description, then that branch will be used.
4. Otherwise, the build description branch will be used.

Note: This is identical to how things normally work, except that in a non-"follow" build tree, step 4 chooses "master" instead of the build description branch.

There is one caveat: if the version control system being used for the checkout is not one that muddle knows how to branch (i.e., at the moment, is not git), then revisions are followed, but branches are not.

You can use `muddle query checkout-branches` or read the build description to see what applies for each checkout.

So, looking at specific commands, and remembering these descriptions are for checkouts using git:

- `muddle checkout` will clone the appropriate branch of the checkout, as described above.
- `muddle pull` and `muddle merge` will first go to the appropriate branch, and then do their work.

- `muddle push` will push the current branch of the checkout, creating that branch at the far end if necessary. It does not check if this is the "appropriate" branch or not.
- `muddle pull-upstream` and `muddle push-upstream` both act as their "normal" versions, but beware that the branch you are on may not exist in the upstream repositories.
- `muddle sync` sets the checkout to the "appropriate" branch. `muddle sync -verbose` tells you how it is deciding this, and `muddle sync -show` tells you but doesn't actually do it. These can be useful when trying to decide why one is not on the branch one expected.
- `muddle stamp version` will use a stamp name that incorporates the build description branch name.
- `muddle stamp release` does not itself know about branched trees. However, the user should take care to specify a sensible version number for the release stamp when creating it. The `-next` switch is unlikely to be useful in this case.

10.2.7 Using muddle sync

Muddle build descriptions implicitly or explicitly describe a branch for each checkout. When branched build trees are in use, that can be even more implicit than normal. Particularly in a branched build tree, it is quite possible to have changed into another branch in one or more checkouts (perhaps to revert to "master" or some other maintenance branch, for comparison). It therefore becomes useful to have a muddle comamnd that puts the build tree back to the "expected" branches.

That's what `muddle sync` is for.

The rules of what it does are necessarily a little complex. They are summarised in `muddle help sync`, or in the docstring for the `sync()` method on the VersionControlHandler class (try `muddle doc sync`).

10.2.8 Useful query commands

muddle query build-desc-branch

Reports the branch of the build description, and whether it is being used as the (default) branch for other checkouts. For instance:

```
$ muddle query build-desc-branch
Build description checkout:builds/* is on branch v1-maintenance
  This WILL be used as the default branch for other checkouts
```

muddle query checkout-id

Reports the VCS revision id (or equivalent) for a checkout.

For instance:

```
$ muddle query checkout-id builds
4c209d3e1ed449d1a5721552671d0506fdcb3de0
$ muddle query checkout-id '(subdomain2)co_repo1.1'
bbf0d5ef5000a88c3d9e6f52283af51f71ad6d1e
$ muddle query checkout-id co.bzr
tibs@kynesim.co.uk-20130212165003-notzt8cbn2n7vi2e
```

These are the same ids that are used in stamp files.

muddle query checkout-vcs

Report on the VCS (version control system) being used for each checkout, and also any VCS options set thereon.

For instance:

```
$ muddle query checkout-vcs
> Checkout version control systems ..
checkout:builds/*          -> VCS git
checkout:co.branch1/*      -> VCS git
checkout:co.branch1.fred/* -> VCS git
checkout:co.bzr/*          -> VCS bzr {'no_follow': True}
checkout:co.fred/*         -> VCS git
checkout:co1/*             -> VCS git
```

muddle query checkout-branches

Report on the branches associated with each checkout:

- the current branch
- the branch that the build description requests (or, for the build description itself, the branch requested at `muddle init`)
- the branch to follow (if the build description has requested "follow")

For instance:

```
$ muddle query checkout-branches
----------------------  --------------  ---------------  ----------------
Checkout                Current branch  Original branch  Branch to follow
----------------------  --------------  ---------------  ----------------
builds                  v1-maintenance  v1-maintenance   <it's own>
co_repo1                v1-maintenance  <none>           v1-maintenance
(subdomain1)builds      v1-maintenance  <none>           v1-maintenance
(subdomain1)co_repo1    v1-maintenance  <none>           v1-maintenance
(subdomain2)builds      v1-maintenance  <none>           v1-maintenance
(subdomain2)co_repo1.1  v1-maintenance  <none>           v1-maintenance
```

The help text (`muddle help query checkout-branches`) has some more examples, and more explanation of the content of the columns.

muddle query checkout-repos

For instance:

```
$ muddle query checkout-repos
> Checkout repositories ..
checkout:builds/*          -> Repository('git', '<repo>', 'builds', branch='branch.follow')
checkout:co.branch1/*      -> Repository('git', '<repo>', 'co.branch1', branch='branch1')
checkout:co.branch1.fred/* -> Repository('git', '<repo>', 'co.branch1.fred', revision='fred', branch=
checkout:co.bzr/*          -> Repository('bzr', '<repo>', 'co.bzr')
checkout:co.fred/*         -> Repository('git', '<repo>', 'co.fred', revision='fred')
checkout:co1/*             -> Repository('git', '<repo>', 'co1')
```

(I've replaced the actual URL of the repository with `<repo>` to save space across the page), or, just showing the URL and not VCS, branch or revision:

```
$ muddle query checkout-repos -url
> Checkout repositories ..
checkout:builds/*          -> file:///.../repo/src/builds
checkout:co.branch1/*      -> file:///.../repo/src/co.branch1
checkout:co.branch1.fred/* -> file:///.../repo/src/co.branch1.fred
checkout:co.bzr/*          -> file:///.../repo/src/co.bzr
checkout:co.fred/*         -> file:///.../repo/src/co.fred
checkout:co1/*             -> file:///.../repo/src/co1
```

(again, I've truncated the middle of the repository URLs to save space).

muddle sync -show

Whilst not strictly a query command, `muddle sync -show` can be useful for showing what muddle believes about a checkout and its branches:

```
$ muddle sync -show _all
Synchronising for checkout:builds/changes_committed
  This is the build description
  The build description has "follow" set
  Following ourselves - so we're already there
Synchronising for checkout:co.branch1/changes_committed
  We have a specific branch in the build description, "branch1"
  We do NOT want to follow the build description
  We want branch "branch1"
Synchronising for checkout:co.branch1.fred/changes_committed
  We have a specific revision in the build description, "fred"
  We do NOT want to follow the build description
  We want revision "fred", branch "branch1"
Synchronising for checkout:co.bzr/changes_committed
  Checkout is marked "no_follow"
  We do NOT want to follow the build description
  We want branch "master"
Synchronising for checkout:co.fred/changes_committed
  We have a specific revision in the build description, "fred"
  We do NOT want to follow the build description
  We want revision "fred"
Synchronising for checkout:co1/changes_committed
  The build description has "follow" set
  The build description is checkout:builds/*
  We want to follow the build description onto branch "branch.follow"
```

10.2.9 What happens if there are sub-domains?

Subdomains always follow the top level build description.

Setting the "follow" flag in a subdomain's build description will have no effect. It is only the top-level build description that can be followed.

10.2.10 What happens with upstream repositories?

A local repository can be linked to several upstream repositories. Checkouts that use that local repository can then do push-upstream and/or pull-upstream commands to those upstreams.

`muddle push-upstream` and `muddle pull-upstream` always use refer to the repository as defined in the build description. Thus they continue to work in the same manner for a "following" build and a normal build.

However, "in the same manner" does mean that pushing a branched checkout will push the branch, and pulling to a branched checkout will try to pull that branch, so whether these commands succeed depends on what branches exist in the remote repositories.

CHAPTER 11

Distributions, licenses and `muddle distribute`

11.1 What we want to do

We want to be able to:

- Distribute build trees to other users, without their needing to clone stuff from the build's repositories.
- Indicate the licensing for checkouts, and thus
- Distribute partial build trees, especially partial build trees whose content is determined by that licensing.
- In particular, distribute partial build trees that satisfy GPL compliance rules, but are still maximally useful *as* build trees.

11.2 The `muddle distribute` command

The `muddle distribute` command is intended to help with the following use cases:

1. I want to distribute all of my source code as an archive (typically, a `.tgz` (gzipped TAR file) or `.zip` file). How do I extract the relevant directories?
2. I want to prepare a binary release for passing to someone.
3. I want to prepare a distribution where some packages are distributed as source code, some as binary, and some not at all. This should be doable by:
   (a) specifying checkouts and packages explicitly, or
   (b) specifying checkout licenses, and selecting based on licenses.

That third case was actually the driving force of the development of muddle distribution support, as we wanted to be able to produce a build tree containing source code for any open-source packages, binary blobs for proprietary packages whose source code is private, and omitting entirely packages that are **private** (i.e., typilcally for proprietary reasons).

The desired intent is that, whatever means (1 through 3) is used, the result should be a functioning muddle build tree. Clearly this might not always be possible for some of the more extreme values of case 3 (if too much of the build tree is not distributed because it is **private**), but even in such cases it should ideally be possible to write the build description(s) in such a manner that something useful can be built.

> (For instance, GPL v2 says "For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable." (I hope that counts as fair quoting!). At the individual checkout/package level, the Makefiles are simple to distribute. Quite often the "scripts used to ..." for

the entire system, though, are indeed those used to build the system, but don't actually function outside the original context in which they were used. We'd like to make it possible, in this sort of situation, to distribute muddle build trees that actually work and produce someting useful.)

Warning: The `muddle distribute` command is new, and the details of its operation, and especially of the `-no-muddle-makefile` and `-copy-vcs` switches, may change.

Note: The `muddle distribute _deployment` command is bundled as part of `muddle distribute`, but works rather differently, and does not distribute any part of the source tree. See *Deployment distribution* below.

11.3 What is a distribution?

A distribution is two things:

1. a copy of all or part of the muddle build tree (the result of distributing)
2. a description of what to copy, for this purpose

The `muddle distribute` command uses an instance of (2) to produce (1).

Distributions are identified by a name, and each is also tagged with which license categories it can distribute.

The name is just a string. As normal, we reserve names starting with underscore (_) for muddle itself to define. These are termed "standard distributions", and new standard distributions may be introduced as time goes on, so avoid defining new distributions with names starting with an underscore.

Standard distributions include:

- `_source_release` - discussed in *Case 1: Source distribution*
- `_binary_release` - discussed in *Case 2: Binary distribution*
- `_for_gpl` - discussed in *Case 3: The other standard distributions*
- `_all_open` - ditto
- `_by_license` - ditto

License categories are the broad classification of license type, and are discussed below.

How does it work? ***(you can safely ignore this bit)***

More-or-less as you might expect, with labels and tags and actions.
For those checkouts or packages that are to be distributed, a rule is added to "build" label `checkout:<name>/distributed` from `checkout:<name>/checked_out` using the DistributeCheckout action, or `package:<name>{<role>)/distributed` from `package:<name>{<role>}/postinstalled` using the DistributePackage action.

Note: *Technically* doing a DistributePackage of just the `obj` directories doesn't require that the package(s) be `/postinstalled`, as `/built` would suffice. I don't think that's worth worrying about at the moment, especially as `muddle` always builds through to `/postinstalled` if at all posible (i.e., that's what `muddle build` does, and see what it says in `muddle help build` about depending on `/postinstalled` being the norm).

For source and binary releases (cases 1 and 2) the `muddle distribute` command itself organises this. For case 3, appropriate code in the build description causes the rules to be added, or sets the conditions that allow `muddle distribute` to do it.
You won't, however, see any `/distribute` tags in the `.muddle` tags directories, as the `/distribute` tag is treated as "transient" - i.e., it doesn't get recorded on disk. This avoids problems if one (for instance) does a source distribution with VCS to one directory, and then without to another, followed by a partial distribution to yet another - it would not be clear what the `/distribute` tags meant, and it would always be necessary to clear them before each `muddle distribute`.

11.4 Source distribution

In a simple build tree, where there are no subdomains, a `.tgz` file can be produced with a command like:

```
$ tar -zcvf archive.tgz build/src build/.muddle build/versions
```

where `build` is the directory containing the build tree.

If VCS directories (`.git/` and the like) are not wanted, then something like:

```
$ tar -zcvf --exclude-vcs archive.tgz build/src build/.muddle build/versions
```

may suffice.

However:

1. Both of those copies over the entirety of the `.muddle` directory, not just the parts needed (so one would typically get tags set for packages indicating that they have been built, which will not be true of the archived build tree).
2. If there are subdomains, the command would need to be rather more complex. And if you're doing something weird, like having a VCS directory (perhaps `.git`) for the tree as stored in the local repositories, but also a historical VCS directory (perhaps `.bzr`) that came with the checkout when it was imported, then it all gets rather difficult.

So instead the standard distribution "_source_release" is provided. The command:

```
$ pushd build
$ muddle distribute -with-version _source_release ../release_1.0
$ popd
$ tar -zcvf release_1.0.tgz release_1.0
```

will copy the sources to a directory called `release_1.0`.

Specifically, it will copy:

- each checkout directory, including the build description, in all domains
- the `versions` directory, if it exists
- the necessary parts of the `.muddle` directory:
  - the `Description`, `RootRepository` and `VersionsRepository` files
  - the `tags/checkout` directory and the tags for the copied checkouts

It does not copy VCS "special" files (as determined by the build description, `.git`, `.gitignore`, etc.) for the checkouts, or for the `versions` directory.

If the `versions` directory is not wanted, then the `-with-versions` switch can be left off.

If the VCS "special" files in the checkouts (and `versions` directory) *are* wanted, then the `-with-vcs` switch should be specified:

```
$ muddle distribute -with-version -with-vcs _source_release ../release_1.0.vcs
```

In either case, because the necessary parts of the `.muddle` directory have been copied, it should be possible to build the build tree in the normal manner:

```
$ cd release_1.0
$ muddle
```

11.5 Binary distribution

Sometimes someone wants a binary release. A binary release is taken to be the "end product" of doing a `muddle build`, and is thus the content of the `install` directories, plus the build descriptions and muddle Makefiles (so the recipient hopefully has enough context to `muddle deploy` if necessary).

For this purpose, the standard distribution "_binary_release" is provided. The command:

```
$ muddle distribute _binary_release ../release_1.0-bin
```

performs a binary release. It copies:

- the `install/<role>` directory for each package (this is, of course, the same directory for all packages in the same role)
- the build description (i.e., its checkout) for the top-level and for any subdomains that have distributed packages (and any "intermediate" subdomains - i.e., if we distribute `package:(a(b))name/*` then we will distribute the build description for subdomain "a(b)", but also for "a").
- the necessary parts of the `.muddle` directory:
  - the `Description`, `RootRepository` and `VersionsRepository` files
  - the `tags/package` directory, and the tags for each package and its role therein
  - the `tags/checkout` directory and the tags for the checkouts used by the distributed packages
- the muddle Makefile (alone in its source checkout directory) for each package.

  For instance, if `package:zlib{x86}/*` is built from `checkout:zlib-3.0/checked_out`, using muddle Makefile `Makefile.muddle` in `src/libs/zlib-3.0`, then the user will receive a directory called `src/libs/zlib-3.0` that just contains `Makefile.muddle`.

  Note that this is *not* always enough to allow deployment, and sometimes other support files will need to be (epxlicitly) added via the build description. See *Adding extra checkout files in a binary distribution* below.

As you might guess, if the `-with-version` switch is given, then it will also distribute the `versions` directory, if it exists.

Also, if `-with-vcs` is specified, the VCS "special" files for the build descriptions and the `versions` directory (if distributed) will also be included.

11.6 Deployment distribution

Sometimes, a customer just wants the content of the `deploy/` directory. This can happen because they are not set up to deal with muddle, or because they really want to have just the results of a build, quickly.

In some situations, it would then be sufficient just to zip up the `deploy/` directory directly, and give that to the customer, but it is better practice to also include a stamp file indicating the status of the source tree from which the deployment was generated, and also some indication of what command was used to build the distribution.

Thus `muddle distribute _deployment` is provided as a special case. It does not work with the same mechanisms as the other distribute commands (which, in particular, means that none of the switches are relevant). Instead:

```
muddle distribute _deployment <target_directory> <labels>
```

acts more or less as if the user had done:

- `muddle deploy <labels>`
- `mkdir -p <target_directory>`
- `cp -a deploy <target_directory>`
- `muddle stamp save <target_directory>/`\`muddle query name\`.stamp
- `cat "muddle distribute _deployment <target_directory> <labels>" > <target_directory>/MANIFEST.txt`

This results in a `<target_directory>` which contains:

1. A copy of the `deploy` directory
2. A version stamp
3. A MANIFEST.txt file, reproducing the command line used

which the command user can then zip or tar up as appropriate.

Note: It is recommended that the user ensures that the `deploy` directory is created from scratch (so it doesn't have any previous build artefacts left in it). The recommended way is either to make a new `muddle init` in a clean directory, or to do a `muddle veryclean`, before using this command.

So, for instance:

```
$ muddle veryclean
$ muddle distribute _deployment ../deploy-20130902 _all
$ cd ..
$ tar -zcf deploy-20130902.tgz deploy-20130902
```

11.7 Filtering the labels distributed

Sometimes it is not desirable to distribute all of the build tree. For instance, only one deployment might be appropriate to distribute, or only a subset of packages and checkouts.

If the `muddle distribute` command line ends with one or more labels, then they will be used to determing a "filter" for the distribution.

If a deployment label is given, then all packages in that deployment will be added to the filter, and also all checkouts used directly by each of those packages.

If a package label is given, then that package and all checkouts used directly by it will be added to the filter.

Finally, if a checkout label is given, then that checkout will be added to the filter.

In all cases, the actual label tag is ignored.

If a filter (i.e., a set of package and checkout labels) has been specified, then after the labels for the distribution, taken from the entire build tree, have been determined, the will be compared to the filter set, and only those labels which occur in the filter set will be kept in the distribution. Note that this filtering is done before build descriptions are added.

So, for instance:

```
$ muddle distribute _binary_release ../release_fred-bin deployment:fred
```

will only consider packages and checkouts in deployment `fred` for distribution.

11.8 Basic setting up of Licenses

Any piece of source code has a license, whether implicit or explicit. Muddle does not try to capture the whole essence of what a "license" means, but simply allows the association of a checkout with a license name and some basic properties.

> *""" "DISCLAIMER: The allocation of licenses to particular checkouts within a build description is purely for programmatic purposes for use with the "muddle" build tool, and is not necessarily intended to reflect the actual licensing state of a particular checkout - for that, please consult the checkout source code directly." """*

11.8.1 The licenses module

Stuff to do with licensing is conveniently packaged in the `muddled.licenses` module, so one might typically do:

```
from muddled.licenses import set_license
```

(and so on for any other items needed from it), or just:

```
import muddled.licenses
```

or:

```
import muddled.licenses as licenses
```

The muddle documentation command can be used to get docstrings from the source code - for instance:

```
muddle doc licenses
muddle doc licenses.LicenseLGPL
muddle doc licenses.set_license
```

11.8.2 License categories

For this purpose, licenses come in four broad categories:

- **gpl**. This category includes all of the FSF GPL licenses.

  For our purposes, they are distinguished by two properties:

  1. There is a requirement to be able to produce the source code on request. That "source code" includes the checkout itself and also the build infrastructure needed to build it (although there is no requirement that this must necessarily work outside the build-as-a-whole).
  2. Some (but not all) GPL licenses "propagate" to other checkouts.

     This means that if package:A is built from GPL checkout:A, and package:B links with package:A, then checkout:B may also need to be distributed.

     There are two considerations here, though:

     (a) Some GPL-licensed checkouts will have licenses with exemptions that specifically allow certain sorts of linking, without causing this GPL "propagation". The best known example is probably the C libraries, which one may explicitly link against, and whose header files can be freely included.

     To allow for this, when defining a GPL license, muddle allows the license to be marked with or without such exemption. There are several examples in the standard licenses - for instance, `GPL-2.0-with-GCC-exception`.

     (b) When muddle is told that package:B depends on package:A, there may be several reasons for this, not all of which trigger GPL "propagation".

     It may be that package:B depends on a file having been created by package:A, and that file itself is not GPL. Or the checkout for package:A may be LGPL, and package:B is linking against package:A dynamically (which is allowed), and not statically.

     A function is thus provided to allow a build description to say that this is in fact the case.

     In either case, note that muddle is not trying to describe the fine details of how GPL licenses work, and nor is the above meant to be an authoratitive explanation. It is still up to the individual writing the build description to use the facilities available to do what is required by law.

- **open-source**. This category includes all open-source licenses which are not GPL-like - i.e., they do not *require* source code distribution, nor do they propagate.
- **prop-source**. Proprietary source. This is source code that is proprietary, but still distributed as source. Examples might include `/etc` files, configuration files, and Python scripts.
- **binary**. This category is used for checkouts (and thus packages using them) that are only to be distributed as binary.

  Note that this will still (normally) distribute the appropriate muddle Makefile, and possibly other user-nominated files.

- **private**. This category is used for checkouts (and thus packages using them) that should not, in general, be distributed. Some provision is also made for not distributing the build description for such packages, as well.

There is also a "meta" type, **not licensed**. Any checkout that has not had its license specified is regarded as **not licensed**. In many cases, such checkouts will be treated as if they had **open-source** licenses.

11.8.3 Standard licenses

Muddle predefines some of the more common licenses, and gives them useful short names (such as "gpl3", "bsd-new" and "mpl"). A list of the standard licenses can be printed out with:

```
$ muddle query licenses
```

For instance:

```
$ muddle query licenses
Standard licenses are:

GPL-2.0                          LicenseGPL('GPL', version='v2.0 only')
GPL-2.0+                         LicenseGPL('GPL', version='v2.0 or later')
GPL-2.0-linux                    LicenseGPL('GPL', version='v2.0', with_exception=True)
GPL-2.0-with-GCC-exception       LicenseGPL('GPL with GCC Runtime Library exception', version='v2.0',
GPL-2.0-with-autoconf-exception  LicenseGPL('GPL with Autoconf exception', version='v2.0', with_excep
GPL-2.0-with-bison-exception     LicenseGPL('GPL with Bison exception', version='v2.0', with_exceptio
GPL-2.0-with-classpath-exception LicenseGPL('GPL with Classpath exception', version='v2.0', with_exce
GPL-2.0-with-font-exception      LicenseGPL('GPL with Font exception', version='v2.0', with_exception
GPL-3.0                          LicenseGPL('GPL', version='v3.0 only')
GPL-3.0+                         LicenseGPL('GPL', version='v3.0 or later')
GPL-3.0-with-GCC-exception       LicenseGPL('GPL with GCC Runtime Library exception', version='v3.0',
GPL-3.0-with-autoconf-exception  LicenseGPL('GPL with Autoconf exception', version='v3.0', with_excep
GPL-3.0-with-bison-exception     LicenseGPL('GPL with Bison exception', version='v3.0', with_exceptio
GPL-3.0-with-classpath-exception LicenseGPL('GPL with Classpath exception', version='v3.0', with_exce
GPL-3.0-with-font-exception      LicenseGPL('GPL with Font exception', version='v3.0', with_exception
LGPL-2.0                         LicenseLGPL('Lesser GPL', version='v2.0 only')
LGPL-2.0+                        LicenseLGPL('Lesser GPL', version='v2.0 or later')
LGPL-2.1                         LicenseLGPL('Lesser GPL', version='v2.1 only')
LGPL-2.1+                        LicenseLGPL('Lesser GPL', version='v2.1 or later')
LGPL-3.0                         LicenseLGPL('Lesser GPL', version='v3.0 only')
LGPL-3.0+                        LicenseLGPL('Lesser GPL', version='v3.0 or later')

APL-1.0                          LicenseOpen('Adaptive Public License', version='1.0')
Apache-2.0                       LicenseOpen('Apache', version='2.0')
Artistic-2.0                     LicenseOpen('Artistic License', version='2.0')
BSD-2-Clause                     LicenseOpen('BSD 2-clause "Simplified" or "FreeBSD"')
BSD-3-Clause                     LicenseOpen('BSD 3-clause "New" or "Revised" license')
BSD-4-Clause                     LicenseOpen('BSD 4-clause "Original" license ("with advertising")')
BSL-1.0                          LicenseOpen('Boost Software License', version='1.0')
CDDL-1.0                         LicenseOpen('Common Development and Distribution License')
EPL-1.0                          LicenseOpen('Eclipse Public License', version='1.0')
IPA                              LicenseOpen('IPA Font License')
Libpng                           LicenseOpen('libpng license')
MIT                              LicenseOpen('MIT License')
MPL-1.1                          LicenseOpen('Mozilla Public License', version='1.1')
MPL-2.0                          LicenseOpen('Mozilla Public License', version='2.0')
OFL-1.1                          LicenseOpen('Open Font License', version='1.1')
OSL-3.0                          LicenseOpen('Open Software License', version='3.0')
Python-2.0                       LicenseOpen('Python License', version='2.0')
QPL-1.0                          LicenseOpen('Q Public License', version='1.0')
UKOGL                            LicenseOpen('UK Open Government License')
Zlib                             LicenseOpen('zlib license')

Proprietary                      LicenseProprietarySource('Proprietary Source')

Private                          LicensePrivate('Private')
CODE NIGHTMARE GREEN             LicensePrivate('Code Nightmare Green')
```

The list of standard licenses has been inspired by some of the content at the Open Source Initiative, specifically by the lists at http://opensource.org/licenses/alphabetical and http://opensource.org/licenses/category.

The key for each license is its SPDX short-form identifier, as described at http://www.spdx.org/licenses/.

11.8.4 Licensing a checkout

There are several ways to assign a license to a checkout.

The first and simplest is directly:

```
set_license(builder, co_label, license)
```

In this call, `license` is either the short name for a standard license, or an instance of a License subclass (see *Creating a new license* below).

`co_label` is the label for the checkout to which this license should apply.

Note: For convenience, we also allow the *name* of a checkout to be used in the call of `set_license`. This seems worth it because this is the function that is used most often in setting up licenses, and constructing labels just for this purpose can be a chore. However, remember that using a checkout name does not allow specifying a subdomain (for those who are using subdomains).

It may be simpler to specify a license for more than one checkout at the same time. For this, you can use:

```
set_license_for_names(builder, co_names, license)
```

`license` is the same as before, but `co_names` is a sequence of checkout *names*. This is normally all that's needed in a build description, but note that it does mean that the checkouts cannot be in a different domain.

So, for instance:

```
set_license(builder, checkout('zlib'), 'Zlib')

set_license_for_names(builder, ['docs', 'specs'],
                      LicenseOpen('Creative Commons Attribution 2.0'
                                  ' UK: England & Wales License.')
```

Some licenses, notably the BSD licenses, require that a file containing the license be distributed, even in binary distributions. In this case, the checkout license is set with:

```
set_license(builder, co_label, license, license_file)
```

where `license_file` is the name of the file (its path relative to the checkout directory). For instance:

```
set_license(builder, 'strace-4.5.20, 'BSD-3-Clause', license_file='COPYRIGHT')
```

This implicitly adds the license file `COPYRIGHT` as a required file for all distributions of checkout `strace-4.5.20`.

Note: You may feel that you need to indicate the appropriate license file for each GPL licensed checkout as well. The GPL licenses require clear indication of the license being used in a binary distribution, and for many purposes the indication in the build description may be enough. However, for absolute safety you might want to force distribution of the license file, which is normally in a file called "COPYING".

(Clearly this is not a problem with a source distribution.)

11.8.5 Creating a new license

There are license classes corresponding to each of the license categories. They are all subclasses of `License`, but `License` itself is not intended to be used directly.

- `LicenseGPL(name, version='v3.1', with_exception=False)`

  This creates a GPL license, version `v3.1`.

  If `with_exception` is True, then the license does not automatically "propagate" to other packages/checkouts that depend on it.

- `LicenseLGPL(name, with_exception=False)`

  This creates an LGPL license. If `with_exception` is True, then the license does not automatically "propagate" to other packages/checkouts that depend on it.

  The default, however, is still False because muddle cannot tell whether "depends on" means linking to dynamically (which would be OK), statically (which would cause "propagation"), or not at all (which would also be OK).

- `LicenseOpen(name)`
- `LicenseProprietarySource(name)`
- `LicenseBinary(name)`
- `LicensePrivate(name)`

All `License` subclass instances have the following values and methods:

- `category` is a string containing the category, one of **gpl**, **open-source**, **binary** or **private**.
- `version` is None or a string containing the license version.

  All licenses can be given a version, using the `version=<string>` argument when creating the instance.

  If no version is specified for a license, then it has no specific version, and the version will not be shown when it is printed.

  Most, but not all, of the standard licenses have versions.

- `is_gpl()` returns True if this is license has category **gpl**.
- `is_lgpl()` returns True if this license has category **gpl** and is LGPL. This is true for the `LicenseLGPL` class.
- `is_open()` returns True if this license has category **open-source** or **gpl**.
- `is_open_not_gpl()` returns True if this license has category **open-source** (but not if it has category **gpl**)
- `is_proprietary_source()` returns True if this license has category **prop-source**.
- `is_binary()` returns True if this license has category **binary**
- `is_private()` returns True if this license has category **private**
- `propagates()` returns True if the license "propagates" to other checkouts. This will be True for licenses in category **gpl** which have `with_exception` set to False.
- `distribute_as_source()` returns True for licenses that relate to source code distribution - so **gpl**, **open-source** and **prop-source**.
- `copy_with_version(version)` returns a copy of the license object, but with its version set to the given string.

So, for instance:

```
top_secret = LicensePrivate('Top Secret', version='3.14')
only_works_on_my_machine = LicensePrivate('Local')
CC-ASA = LicenseOpen('Creative Commons Attribution-ShareAlike'
                     ' Unported License', version='3.0')
```

Note that `str` of a License gives back the license name and version (if any):

```
>>> str(top_secret)
'Top Secret 3.14'
```

whilst `repr` will give back how it was created:

```
>>> repr(top_secret)
"LicensePrivate('Top Secret', version='3.14')"
```

The standard licenses are stored in a dictionary:

```
licenses.standard_licenses
```

Please do *not* alter the contents of this dictionary.

Obviously the standard licenses do not include all versions of even the common licenses, and sometimes a checkout is licensed with an older license. A variant version of an existing standard license can be produced using the `copy_with_version` method - for instance:

```
>>> from muddled.licenses import standard_licenses
>>> mpl11 = standard_licenses['MPL-1.1']
>>> mpl10 = mpl11.copy_with_version('1.0')
>>> print mpl10
Mozilla Public License 1.0
```

11.9 The other standard distributions

We've already met the standard distributions that do not take account of checkout licenses, "_source_release" and "_binary_release". There are also some standard distributions to help with common cases where licenses should be observed. These are:

- "_for_gpl"
- "_all_open"
- "_by_license"

Warning: None of these distribute **private** licensed checkouts. This means that if there are such checkouts in the build tree, and the build description is meant to *work* when distributed, then some case must be taken in its construction - see *Keeping parts of the build description private* below.

11.9.1 GPL source distribution

The command:

```
$ muddle distribute _for_gpl ../gpl_release_1.0
```

is meant to help with creating a GPL-compliant source code release.

It distributes

- all checkouts that have a GPL license (i.e., in category GPL)
- all checkouts to which a GPL license has "propagated" (and which have not explicitly said they are not affected - see *Avoiding unnecessary GPL "propagation"* below)
- all open source checkouts on which one of the GPL licensed checkouts depend

Actually: if the package built from the GPL checkout depends directly on any non-GPL open source checkouts, or if it depends directly on any packages that are built from non-GPL open source checkouts, then those open source checkouts will also be included.

If any of the selected dependant checkouts are not open-source, then muddle will give a warning, and will not distribute the offending checkouts. Occasionally, such a warning may be spurious, as muddle does not know why a package depends on a checkout, and the writer of the build description may have done something non-standard. The correct behaviour, though, is still not to distribute the offending checkout.

- any necessary build descriptions

It will fail if GPL "propagation" affects checkouts that have declared **binary** or **private** licenses.

11.9.2 Open source distribution

The command:

```
$ muddle distribute _all_open ../open_release_1.0
```

is meant to help with creating a release of all the open-source checkouts in a build tree. As such, it distributes everything that "_for_gpl" does, plus any checkouts that have licenses in the **open-source** category.

Since it includes the "_for_gpl" distribution, it will also fail if GPL "propagation" affects checkouts that have declared **binary** or **private** licenses.

11.9.3 By license source/binary distribution

The command:

```
$ muddle distribute _by_license ../mixed_release_1.0
```

attempts to distribute as much of the build tree as it can in a license-appropriate manner. Thus it attempts to distribute:

- source code for **gpl**, **open-source** and **prop-source** licensed checkouts
- binaries for **binary** licensed checkouts
- nothing at all for **private** licensed checkouts

It starts by including the source code content of an "_all_open" distribution (and can fail for the same reason).

For checkouts with a **binary** license, it determines which packages are built (directly) from them, and then which role those packages are in. It then distributes the entirety of the `install/<role>` directory for each such role.

(See *Not distributing too many binaries* below for how to write a build description that does not distribute binaries you were not expecting.)

11.10 Avoiding unnecessary GPL "propagation"

As we've said, muddle has to assume that "depends on" means "links to" (or the equivalent) when calculating GPL "propagation". Since this is not always so, we need a way of telling muddle when it is not.

There are three cases:

1. The GPL license being used is marked as "with exemption" or "with exception", in which case there is no "propagation".

2. The checkout is never actually linked against (whatever action is needed to "propagate" the license is not done). An example would be busybox, which only provides executables (programs). Sometimes, even a checkout which provides libraries might not actually be linked against by any other checkouts (perhaps it was included in the build for another reason).
3. Particular packages/checkouts do depend on the checkout, but not in a way that is significant for GPL "propagation".

If we know that nothing "builds against" a particular checkout (whatever that means in the relevant context), then we can declare this with:

```
set_nothing_builds_against(builder, co_label)
```

where `co_label` is the checkout that no-one builds against. It is sometimes worth adding a comment to explain what "builds against" means for this particular checkout.

For convenience, this can also be stated in the call of `set_license` - for instance:

```
set_license(builder, 'busybox-1.19.3', 'GPL', not_built_against=True)
```

If we know that a particular label depends upon our checkout, but does not "build against" it, then we can use:

```
set_license_not_affected_by(builder, this_label, co_label)
```

This says that the checkout with label `co_label` does not in fact affect the license of `this_label`, despite implicit GPL "propagation". `this_label` may be a package or a checkout.

So, for instance, we might have:

```
muddled.pkgs.make.medium(builder, 'libProp', ['x86'], 'libProp-2.3')
muddled.pkgs.make.medium(builder, 'program', ['x86'], 'program-4.9',
                         deps=['libProp'])

set_license(builder, checkout('libProp-2.3'), 'LGPL-3.0')
set_license(builder, checkout('program-4.9'), 'CODE NIGHTMARE GREEN')

# 'program' links dynamically to 'libProp', and thus doesn't need
# to be distributed as source
set_license_not_affected_by(builder, package('program', 'x86'), checkout('libProp-2.3'))
```

This function does not check that:

- `this_label` actually depends on `co_label`
- `co_label` is GPL licensed
- `co_label` is GPL licensed with `with_exception` set to False (i.e., that it is a "propagating" license)

but none of these will hurt if incorrect.

Note: If checkout A depends on B depends on C which depends on a GPL checkout D, saying that B is "not built against" D doesn't say anything about A or C, so you will have to address A and C with separate function calls, if appropriate.

11.11 Build descriptions

11.11.1 Build description licensing

Build descriptions are a little awkward in licensing terms, as all distributions wish to include them.

There are three obvious ways to handle build description licenses:

1. Do not declare any license in the muddle build description. This is, of course, not the same as not actually having a license for the build description, it just means not telling muddle what it is (muddle has no way of reading a license statement in the comments or docstring of the build description, or held in a LICENSE file, and so on).

   This is perhaps the simplest option, as **not-licensed** checkouts are allowed in any distribution.

2. Declare a **gpl** or **open-source** license. This is obviously only a solution if that form of licenses is actually applicable to the particular build description.

   Again, such a build description is clearly allowed in any distribution.

3. Declare a **prop-source** license. Muddle assumes that proprietary source is not to be distributed in `_for_gpl` or `_all_open` distributions, but in the case of a build description, will just output a warning and carry on:

```
WARNING: DISTRIBUTING BUILD DESCRIPTION DESPITE LICENSE CLASH
  Checkout checkout:builds/distributed is not allowed in distribution "_for_gpl"
  Checkout has license "Wombat & co. licensed, may be distributed as source",
    which is "prop-source", distribution allows "gpl", "open-source"
END OF WARNING
```

If you tell muddle that your build description has a **binary** or **private** license, then it will not be possible to do a `_for_gpl` or `_all_open` distribution.

There is also another solution, though:

11.11.2 Distribution specific build descriptions

In some cases, the full build description is just not appropriate.

A typical case is when doing a `_for_gpl` distribution of a large and complex build tree, where much of it is not GPL. If the full build description is distributed, the recipient can only `muddle build` those checkouts that have actually been distributed, and these are likely to be buried in an unobvious way in the rest of the build description.

The appropriate solution in this case is to provide a replacement (simpler) build description.

Let us consider doing `muddle distribute _for_gpl` of a build tree that has its normal build description in `src/builds/01.py`.

To provide a build description just for that distribution, add a file called `_distribution/_for_gpl.py` in the build description checkout - so the full path is:

```
src/builds/_distribution/_for_gpl.py
```

If `muddle distribute <name>` sees a build description file called `_distribution/<name>.py`, that is, in directory `_distribution` and named after the distribution (don't forget the `.py`), then instead of distributing the "normal" contents of the checkout, it will instead just distribute that file.

So let us assume we have a build description checkout that contains:

```
.git/
01.py
01.pyc
kernel.py
```

```
userspace.py
_distribution/
    _for_gpl.py
```

then `muddle distribute _for_gpl` will produce a target build description checkout that looks like:

```
01.py
```

where `01.py` contains the text from `_distribution/_for_gpl.py`.

Note: When such a substitution is performed, the VCS information for the build description checkout is never copied, since it would be misleading at best. Similarly, "private" files (see *Keeping parts of the build description private*) are not relevant, as the normal build description is not being copied.

In comparison, `muddle distribute _source_release` would copy all of the `.py` files from the directory (and from `_distribution` as well), and might or might not copy the VCS, because there is no file called `_source_release.py` in `_distribution/`).

Note: It is not an error if the `_distribution` directory does not exist - it only needs to be created if you need it.

11.11.3 Keeping parts of the build description private

There are two reasons to keep part of the build description private:

- describing how to build private checkouts/packages may itself be something that should not be advertised, and
- once the build tree has been distributed *without* private checkouts, any part of the build description that relies on them won't work.

The solution to this is to construct the build description so that **private** checkouts and packages are kept separate:

1. separate in role
2. separate in build description

The first is necessary so that a binary distribution won't try to distribute a role that includes **binary** and **private** files (which would cause a clash, and thus won't work).

The second essentially means organising the build description so that the **private** parts are in a separate file, coded to a particular API.

This is best illustrated with an example.

Assuming that our build description lives in `src/builds`, then we put our private description into a separate Python file, `src/builds/private.py`:

```
# How to build our private code

import muddled.deployments.collect as collect
import muddled.pkgs.make as make

from muddled.depend import checkout
from muddled.licenses import LicensePrivate, set_license

def describe_private(builder, *args, **kwargs):
    """This describes how to do the private part of our build.
```

```
    We require it to be called as:

        describe_private(builder, deployment='<deployment-name>')
    """
    deployment = kwargs['deployment']

    make.medium(builder, 'secret-thing', ['x86-private'], 'secretThing-0.1')

    set_license(builder, checkout('secretThing-0.1'),
                LicensePrivate('Do Not Distribute'))

    collect.copy_from_role_install(builder, deployment, role='x86-private',
                                   rel='', dest='')
```

and then in the "normal" build description (typically `src/builds/01.py`), import the function from that file:

```
from private import describe_private
```

and call it in the body of the build description `describe_to` - for instance:

```
# We also have some private stuff, described elsewhere
describe_private(builder, deployment=deployment)
```

and also tell muddle that `private.py` is not to be distributed as-is:

```
# So that "elsewhere" is private - i.e., private.py
# and we should never distribute it in non-private distributions
for name in get_distributions_not_for(builder, ['private']):
    set_private_build_files(builder, name, ['private.py'])
```

This last tells muddle that when `muddle distribute` is called to distribute something that does not distribute **private** licensed checkouts, the content of `private.py` is to be replaced by a "stub" file, which just contains:

```
def describe_private(builder, *args, **kwargs):
    pass
```

This means that the build description will continue to work, but when it calls the `describe_private()` function, nothing will happen, and no information on the original content of `private.py` is leaked.

In summary, the *name* of the private file (or files) does not matter, but their API must be a single function with signature:

```
describe_private(builder, *args, **kwargs)
```

11.12 Refining binary distributions

11.12.1 Adding extra checkout files in a binary distribution

When doing binary distributions, "muddle distribute" will generally include the necessary muddle Makefiles, so that the data in `install/<role>/` can be processed to give a deployment in the normal manner.

Sometimes, however, the muddle Makefile is not sufficient - typically because it calls out to the "original" Makefile or files from a checkout.

Thus there is a way to say that extra files should be included, for a given checkout, in a particular distribution:

```
distribute_checkout_files(builder, name, label, source_files)
```

In this:

- `name` is the name of a distribution, or a wildcard string matching one or more distributions (see *Distribution names and wildcards* below).
- `label` is a checkout label. The label tag is ignored.
- `source_files` is a list of extra source files, relative to the checkout directory.

So, for instance:

```
muddled.pkgs.make.medium(builder, 'binapp', ['x86'], 'binapp-1.2')
distribute_package(builder, 'marmalade', package('binapp', 'x86'))
distribute_checkout(builder, 'marmalade', checkout('binapp-1.2'),
                    ['Makefile', 'src/Makefile', 'src/rules'])
```

causes the distribution "marmalade" to include the extra files as named.

In fact, sometimes one wants to assert this for a whole range of distributions, and so it is perhaps more likely that one woud write:

```
for name in get_distributions_for(['binary']):
  distribute_checkout(builder, name, checkout('binapp-1.2'),
                      ['Makefile', 'src/Makefile', 'src/rules'])
```

which loops over all the distributions that distribute anything with a **binary** license, and set them to distribute the extra files.

11.12.2 Not distributing too many binaries

Binary distributions distribute the content of `install/<role>` directories. Muddle has no way of telling which packages have actually put content into a particular `install/<role>` directory - just because a package is in role `<role>` does not necessarily mean that it does so.

So when a particular package is distributed as binary, and causes its `install/<role>` directory to be distributed, it may "drag along" unwanted other content, from other packages in the same role.

This can only really be avoided by writing a build description which does not use the same role for incompatible (licensing) purposes.

When first creating a new distribution, it is always worth looking hard at the content of any `install` directories that are being distributed, to check for unexpected files.

The commands:

```
$ muddle query role-licenses
$ muddle -n distribute <name> ../fred
```

may be useful in working out what is going on.

Note: It is arguable that a role that contains binaries from **binary** and **open-source** or **binary** and **gpl** checkouts should be regarded as a clash in the same way that **binary** and **private** is. This may be implemented in a future version of muddle.

11.13 Creating your own distributions

It is also possible to create new distributions, by specifying them in the build description.

11.13.1 The distribution module

Stuff to do with distributions is conveniently packaged in the `muddled.distribe` module, so one might typically do:

```
from muddled.distribute import name_distribution
```

(and so on for any other items needed from it), or just:

```
import muddled.distribute
```

or:

```
import muddled.distribute as distribute
```

The muddle documentation command can be used to get docstrings from the source code:

```
muddle doc distribute
muddle doc distribute.name_distribution
```

11.13.2 Name a distribution first

Before you can use a new distribution, it must be "named":

```
name_distribution(builder, name)
name_distribution(builder, name, categories)
```

`name` is the name for the new distribution. It should not start with an underscore, as such names are reserved for the standard distributions (which muddle has already defined, before reading the build description).

If `categories` is given, then it is a sequence of license categories (so, taken from **gpl**, **open-source**, **binary** and **private**). Only checkouts with those categories will be allowed to be distributed with this new distribution.

> **Warning:** When naming the license categories that a distribution uses, **gpl** and **open-source** are distinct and separate - there is no assumption that **open-source** includes **gpl**. In this instance **open-source** means "open but not GPL".

If `categories` is not given (or is None), then all license categories are allowed.

All non-standard distributions must be named before they can be used. It is not an error to name a distribution more than once, but the categories must match each time it is named.

Note: If a subdomain names distribution "Fred", and then the parent domain refers to domain "Fred" after including that subdomain, that will work, as the naming of "Fred" (in the subdomain) did come before its use (in the parent domain). However, we don't recommend writing this sort of code in a build description - it would generally be better to re-name the distribution in the parent domain as well, so anyone reading the description can easily tell how it is defined.

11.13.3 Distribution names and wildcards

The functions that add checkouts, packages and checkout files to distributions can all take a shell-style wildcard instead of a specific distribution name. They will then add the relevant entity to all distributions with names that match that wildcard.

Only distributions that have already been named will be considered in "expanding" the wildcard.

Wildcarding is done with `fnmatchcase` from the Python `fnmatch` module, so:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

11.13.4 Distribute a checkout

```
distribute_checkout(builder, name, label, copy_vcs=False)
```

This is used to say that a particular checkout (identified by `label`) is to be part of the distribution called `name`. If `name` is wildcarded (see *Distribution names and wildcards*), then the checkout will be added to each distribution that matches the wildcard.

If `copy_vcs` is true, then the checkouts VCS "special" files should be distributed. The default is not to do so.

Note that:

1. the label name may also be a wildcard ('*'), in which case all matching checkouts will be distributed.
2. the label tag is ignored.

The function creates a rule saying that `checkout:<name>/distributed` is build from `checkout:<name>/checked_out` using the DistributeCheckout action.

Adding a checkout to the same distribution more than once has no special effect, except that it is the last call that sets the value of `copy_vcs` that will be used.

Note: All checkouts in the build description are implicitly part of distribution `_source_release`. The `muddle distribute` command itself calls `distribute_checkout` to add them to this distribution, after the build description has been read. Thus there is never any point in explicitly adding a checkout to `_source_release` in the build description itself, as that will be ignored.

11.13.5 Distribute specific files from a checkout

```
distribute_checkout_files(builder, name, label, source_files)
```

This is used to say that the files in sequence `source_files`, named relative to the source directory of checkout `label`, should be distributed as part of distribution `name`.

If `name` is wildcarded (see *Distribution names and wildcards*, then the files will be added to each distribution that matches the wildcard.

Note that the label name may *not* be wildcarded, but the label tag is still ignored.

This is the function that is used internally to add muddle Makefiles for packages. It can also be used directly to specify that other files must also be distributed.

Multiple calls with the same `builder`, `name` and `label`, but different `source_files`, will just add the new files to the same distribution.

Calling `distribute_checkout_files` after calling `distribute_checkout` for the same distribution and checkout has no effect - the latter call has already selected all files.

Calling `distribute_checkout` after calling `distribute_checkout_files` for the same distribution and checkout means that the calls of the latter are essentially ignored, because `distribute_checkout` is choosing all files.

11.13.6 Build description checkouts

Build description checkouts are treated specially.

The `muddle distribute` command adds the necessary rules itself, by calling `distribute_build_description`, and saying that each `checkout:<build-desc>/distributed` is built from `checkout:<build-desc>/checked_out` using the DistributeBuildDescription action (which is actually very similar to the DistributeCheckout action, of course).

There is never any reason to call `distribute_build_description` directly, as `muddle distribute` will always override it.

Similarly, there is never a reason to call `distribute_checkout()` on a build description checkout, as `muddle distribute` will always override it, too.

11.13.7 Distribute a package

```
distribute_package(builder, name, label, obj=False, install=True,
                   with_muddle_makefile=True)
```

This is used to say that a particular package (identified by `label`) is to be part of the distribution called `name`. If `name` is wildcarded (see *Distribution names and wildcards*, then the package will be added to each distribution that matches the wildcard.

Note that:

1. the label name and role may also be wildcards ('*'), in which case all matching packages will be distributed.
2. the label tag is ignored.

If `obj` is true, then the `obj/<package-name>/<role>` directory for `label` will be distributed.

If `install` is true, then the `install/<role>` directory for `label` will be distributed.

If `with_muddle_makefile` is true, then the muddle Makefile for this package will also be distributed. For instance, if `package:zlib{x86}/*` is built from `checkout:zlib-3.0/checked_out`, using muddle Makefile `Makefile.muddle` in `src/libs/zlib-3.0`, then the user will receive a directory called `src/libs/zlib-3.0` that just contains `Makefile.muddle`. Note that this is *not* always enough to allow deployment, and sometimes other support files will need to be (epxlicitly) added via the build description.

If both `obj` and `install` are false, nothing much is going to be done for this package. The current implementation doesn't grumble about that.

The function creates a rule saying that `package:<name>{<role>)/distributed` is built from `package:<name>{<role>}/postinstalled` using the DistributePackage action.

Adding a package to the same distribution more than once has no special effect, except that it is the last call that sets the value of `obj`, `install` and `with_muddle_makefile` that will be used for that distribution.

Note: All packages in the build description are implicitly part of distribution `_binary_release`. The `muddle distribute` command itself calls `distribute_package` to add them to this distribution, after the build description has been read. Thus there is never any point in explicitly adding a package to `_binary_release` in the build description itself, as that will be ignored.

11.14 Finding out about distributions

A variety of useful functions are provided by the `distributions` module.

You can use:

```
get_distribution_names(builder)
```

to return the names of all the distributions that are currently defined, or:

```
get_distribution_names_for(builder, categories)
```

to return the names of any distributions that have *all* of the license categories in `categories`. So, for instance:

```
for name in get_distributions_for(['gpl', 'open-source']):
    distribute_checkout_files(builder, name, co_label, 'Makefile')
```

Contrariwise:

```
get_distribution_names_not_for(builder, categories)
```

returns the names of any distributions that do not have *any* of the license categories in `categories`. So, for instance:

```
for name in get_distributions_not_for(['private']):
    set_private_build_files(builder, name, ['private_app.py'])
```

The function:

```
get_distributions_by_category(builder)
```

returns a dictionary with category names as keys, and sets of distributions names as values - so it allows one to look up which distributions can distribute particular license categories.

Finally:

```
get_used_distribution_names(builder)
```

returns the names of all the distributions that are "in use", i.e., referenced by some action in the dependency tree. This will not normally return any of the "standard" distributions, because they are set up by "muddle distribute" after the build description has been read.

11.15 Muddle commands

- `muddle distribute` handles distribution.
- `muddle help distribute` gives the help text for `muddle distribute`, which summarises some of this document.
- `muddle query distributions` lists the names of the distributions that exist, either because they are defined in the build description, or because they are predefined.

- `muddle query licenses` prints out the standard licenses
- `muddle query checkout-licenses` prints the licenses for the checkouts in the dependency tree, and other related information it has deduced. This is deliberately verbose, giving as much information as possible.
- `muddle query role-licenses` prints the licenses used in each role, and whether there are any obvious clashes.

11.16 More license stuff

The `licenses` module provides more functions, mostly aimed at the muddle developer. See their docstrings for details on how to use them.

A tuple containing all of the license categories is provided:

```
ALL_LICENSE_CATEGORIES
```

There are some useful query functions:

```
get_gpl_checkouts(builder)
get_implicit_gpl_checkouts(builder)
get_open_checkouts(builder)
get_open_not_gpl_checkouts(builder)
get_prop_source_checkouts(builder)
get_binary_checkouts(builder)
get_private_checkouts(builder)
get_not_licensed_checkouts(builder)
```

for finding out which checkouts have particular categories of licenses.

You can find out if a checkout has a license using:

```
builder.db.checkout_has_license(co_label)
```

You can retrieve the particular license using:

```
get_license(builder, co_label)
```

which will return None if the checkout does not have a license. If you'd prefer an exception in that case, you can call it as:

```
get_license(builder, co_label, absent_is_None=False)
```

You can find out if a checkout has a license that is in a particular category (or categorie) with:

```
checkout_license_allowed(builder, co_label, categories)
```

Note: Some of the functions in the `licenses` module are just wrappers around calls of `builder.db` methods. Specifically:

- `set_license` is partly a wrapper for `builder.db.set_checkout_license`, but it (`set_license`) may also store the license file information, which the low-level `set_checkout_license` method does not do.
- `get_license` is a wrapper for `builder.db.get_checkout_license` (but beware that the latter swaps the default for `absent_is_None`, so that it defaults to raising an exception if there is no license)
- `set_license_not_affected_by` is a wrapper for `builder.db.set_license_not_affected_by`

There is also a boolean query:

```
builder.db.checkout_has_license(co_label)
```

which doesn't currently have a wrapper function.

11.17 Questions

11.17.1 What happens if we rebuild in a binary distribution?

Experimenting shows:

```
$ m3 distrebuild package:main_pkg{x86}
Building: package:main_pkg{x86}/distclean ..
> Building package:main_pkg{x86}/distclean[T]
Can't build package:main_pkg{x86}/distclean: Missing source directory
  package:main_pkg{x86}/distclean[T] depends on checkout:main_co/*
  Directory /home/tibs/sw/m3/tests/transient/binary/src/main_co does not exist
```

but, unfortunately:

```
$ m3 rebuild main_pkg
Killing package:main_pkg{x86}/built
Clearing tags for package:main_pkg{x86}/built
  package:main_pkg{x86}/built
  package:main_pkg{x86}/installed
  package:main_pkg{x86}/postinstalled
Building package:main_pkg{x86}/postinstalled
> Building package:main_pkg{x86}/built
Can't build package:main_pkg{x86}/postinstalled - Missing source directory
  package:main_pkg{x86}/built depends on checkout:main_co/*
  Directory /home/tibs/sw/m3/tests/transient/binary/src/main_co does not exist
```

so we've now lost our "built" tags. Oh well. At least it told us.

CHAPTER 12

The muddle release mechanism

12.1 Summary

The muddle release mechanism is intended to be useful in preparing some form of meaningful binary release for a customer. It is more specific than can be obtained using `muddle distribute`, and also has more mechanism associated with it.

The normal pattern of usage is expected to be:

1. Produce a build that needs releasing to the customer.
2. Define what is to be released, in the build description (write a `release_from()` method).
3. Verify the build tree is up-to-date and builds the entities to be released:

```
$ cd build-dir
$ muddle pull _all
$ muddle veryclean
$ muddle build _release
```

4. Produce a release file, and ideally archive it:

```
$ muddle stamp release simple v1.0
$ pushd versions
$ git commit simple_v1.0.release -m 'Release file for simple v1.0'
$ muddle stamp push
```

   (a more detailed commit message would doubtless be a good thing.)

5. Test the release mechanism in the current build tree (this doesn't *prove* the build works, because it doesn't check the whole tree out anew and try building it from scratch, so it mustn't be used as the final release):

```
$ muddle release -test versions/simple_v1.0.release
```

   If that shows problems, repeat from an earlier stage.

5. In order to produce an actual release, create a clean working directory and prepare the release there, using the same release file:

```
$ cd ..
$ mkdir release-dir
$ cd release-dir
$ muddle release ../build-dir/versions/simple_v1.0.release
```

6. Send the customer the resultant archive file, which will be called something like:

```
    simple_v1.0_40c60888d187c4e639820e77bd9532d007f74f92.tgz
```

12.2 Muddle release files

Muddle releases are defined in release files, which are muddle stamp files (specifically, a stamp version files) with extra fields (at the start):

```
[RELEASE]
version = 1.2
name = simple
archive = tar
compression = gzip
```

See the DESCRIPTION section of `muddle doc version_stamp` for details on the content of stamp files.

Release files are created using `muddle release`, and put into the "versions/" directory. `muddle stamp release` will automatically add them to the version control system in that directory, just as `muddle stamp version` does for its stamp files. They can then be committed, and then pushed with `muddle stamp push`.

12.2.1 Explicit version numbering

The command:

```
$ muddle stamp release Project99 1.2
```

creates a release stamp file called:

```
versions/Project99_1.2.release
```

which looks like a normal version 2 stamp file with an extra section after the `[STAMP]` section:

```
[RELEASE]
name = Project99
version = 1.2
archive = tar
compression = gzip
```

12.2.2 Guessing the next version number

Since release files are named consistently, and are version controlled in the "versions/" directory, muddle should be able to make a sensible guess at the next version number to use.

Thus:

```
$ muddle stamp release Project99 -next
```

will look through the "versions/" directory for `Project99` release files, which are called:

```
Project99_v<major>.<minor>.release
```

where `<major>` and `<minor>` are release numbers.

Note: Leading zeroes are ignored, so `1.01` is the same as `1.1`, and indeed `001.001` is also the same as `1.1`.

The current "guessing" mechanism only supports two-part version numbers.

So if the "versions/" directory contains:

```
Project88.v28.1.release
Project99_v1.1.release
Project99_v13.2.release
Project99_v22.3.release
Project99_v27.1.release
Project99_v28.release
```

then the next version for `Project99` will be `27.2`.

The `Project88` file will be ignored, as it has the wrong name, and the `Project99_v28.release` file is ignored because its release number is not of the form `<major>.<minor>`.

12.2.3 Archive and compression switches

You can use the `-archive` and `-compression` switches to specify the release archive mechanism and release compression - for instance:

```
$ muddle stamp release Project99 1.2.3 -archive tar -compression bzip2
```

12.2.4 Template release files

It is also possible to create a "template" release file, with unspecified release name and version:

```
$ muddle stamp release -template
```

which creates a file called:

```
versions/this-is-not-a-file-name.release
```

with the `[RELEASE]` section set to:

```
[RELEASE]
name = <REPLACE THIS>
version = <REPLACE THIS>
archive = tar
compression = gzip
```

The user is expected to rename the file and edit the `<REPLACE THIS>` values to something sensible. The `muddle release` command will not accept release files where these fields have not been edited.

Note: Both release name and release version follow the same rules:

- the first character must be an ASCI alphanumeric ('A'-'Z', 'a'-'z', '0'-'9')
- any following characters must be ASCII alphanumeric, '-', '_' or '.'

The release archive must currently be "tar".

The release compression must currently be one of "gzip" or "bzip2". The default is "gzip".

12.2.5 More information

See `muddle help stamp release` and `muddle help stamp push` for more information.

12.3 The `_release` build target

This is a build target specifically for use in `muddle build _release`, indicating what is to be built for a release.

Its meaning is defined in the build description, using `builder.add_to_release_build()`.

Items can be added to the `_release` entity as labels, individually:

```
builder.add_to_release_build(package('fred', 'x86')
```

or as sequences:

```
builder.add_to_release_build([package('fred', 'x86'),
                              Label.from_string('package:(infrastructure)jim{*}/*')])
```

Wildcarded labels can be used, if necessary.

The "special" values that start with an underscore may also be used, for instance:

```
builder.add_to_release_build('_default_deployments')
```

Warning: The values `_all`, `release` and `just_pulled` may not be used:

- The interpretation of `_all` depends on the muddle command being used - use `_all_checkouts`, `_all_packages` or `_all_deployments` instead.
- Using `_release` inside `_release` would, of course, just be asking for trouble.
- And `just_pulled` is too transient to be useful as a release target.

The `_release` value is interpreted and expanded lazily, when the muddle command line is interpreted. This means one is allowed to specify:

```
builder.add_to_release_build('_default_deployments')
```

in the build description before the deployments have all been described, which is useful.

Warning: This also means that `add_to_release_build()` itself cannot complain if you add a non-existent label. Such warnings will have to wait until `_release` is actually used in a "muddle" command.

`muddle query release` can be used to find out what `_release` is set to (see `muddle help query release` for details).

Warning: If your build includes subdomains, then you need to know that only calls to `add_to_release_build()` in the top-level build will be effective. Calls in subdomain build descriptions *will be ignored*. It is up to the top-level build to explicitly include anything it wishes to release from the subdomain.
This decision is made to avoid confusion over what is meant by adding special names (such as `_all_checkouts`) to the release specification in a subdomain. It is possible that a future version of muddle might instead expand a subdomain's `_release` value as the subdomain was "promoted", but I am not sure that this is worth the added complication.
There is also a strong argument that only the top-level build description can have full awareness of what should actually be released.

12.4 The `release_from` function in the build description

The `_release` special name describes what is to be built for a release, but another mechanism is needed to tell muddle what to do with the results.

The `release_from()` function in the build description is responsible for copying files into the release directory (which will become the release tarball). For instance:

```
import shutil

def release_from(builder, release_dir):

    f0 = package('first_pkg', 'x86')
    m0 = package('main_pkg', 'x86')
    s1 = package('second_pkg', 'x86', domain='subdomain1')

    install_path = builder.package_install_path

    # Copy our executables, with permissions intact
    shutil.copy(os.path.join(install_path(f0), 'first'), release_dir)
    shutil.copy(os.path.join(install_path(m0), 'main0'), release_dir)
    shutil.copy(os.path.join(install_path(s1), 'second'), release_dir)
```

The normal build description will have been loaded before this function is called, so it is safe to assume that `builder` has its normal content.

Also, muddle will have added the build description's checkout directory to the Python path before calling it (so other Python files therein can be imported), just as it does before calling `describe_to`.

Muddle will already have copied the release file into the release directory, but otherwise it will be empty.

The following methods may be particularly useful:

- `builder.checkout_path`
- `builder.package_obj_path`
- `builder.package_install_path`
- `builder.deploy_path`

Use `muddle doc <method>` to confirm what they do.

Note: `muddle bootstrap` puts an empty `release_from` function into its template build description.

Warning: If your build includes subdomains, then you need to know that only the `release_from()` function in the top-level build will be executed. Any `release_from()` functions in subdomain build descriptions will not be called by muddle itself.

12.5 The `muddle release` command

The `muddle release` command actually makes a release.

For example:

```
$ muddle release project99-1.2.release
```

This:

1. Checks that the current directory is empty, and refuses to proceed if it is not.

   We always recommend doing `muddle init` or `muddle bootstrap` in an empty directory, but muddle insists that `muddle release` must be done in an empty directory.

2. Does `muddle unstamp <release-file>`,

3. Copies the release file to `.muddle/Release`, and the release specification to `.muddle/ReleaseSpec`. The existence of the former indicates that this is a release build tree, and "normal" muddle will refuse to build in it.

   The `ReleaseSpec` contains the basic information describing a release (the name, version, archive and compression mechanisms, and the SHA1 hash of the release file).

4. Sets some extra environment variables, which can be used in the normal manner in muddle Makefiles:

   - `MUDDLE_RELEASE_NAME` is the release name, from the release file.
   - `MUDDLE_RELEASE_VERSION` is the release version, from the release file.
   - `MUDDLE_RELEASE_HASH` is the SHA1 hash of the release file

   "Normal" muddle will also create those environment variables, but they will be set to `(unset)`.

5. Does `mudddle build _release`.

6. Creates the release directory, which will be called `<release-name>_<release-version>_<release-sha1>`. It copies the release file therein.

7. Calls the `release_from(builder, release_dir)` function in the build description, which is responsible for copying across whatever else needs to be put into the release directory.

   (Obviously it is an error if the build description does not have such a function.)

8. Creates a compressed tarball of the release directory, using the compression mechanism specified in the release file. It will have the same basename as the release directory.

   The existence of this latter file indicates that this is a release build tree, and some muddle commands will thus refuse to work in it (notably, anything to do with pushing to or pulling from a VCS).

   (Of course, the user can delete the file, but if they do then that's their responsibility.)

The `-test` variant (`muddle release -test`) omits the first two stages, and may thus be done in the working build tree. It still copies the `Release` and `ReleaseSpec` files into the `.muddle` directory, and thus marks the build as a release build - this can be undone by deleting `.muddle/Release`.

Warning: Do not release a test release. Always prepare a *proper* release created in an empty directory. Test releases cannot be trusted as the build tree may still contain artefacts from earlier development, and a clean build may or may not work.

12.6 Other useful commands

Whether a build is a release build, the content of the release specification, and what `_release` is defined as, can be found using:

```
$ muddle query release
```

You can find out exactly what labels will be built in the normal manner, using:

```
$ muddle -n build _release
```

Since a release file is an extended stamp file, it is perfectly legitimate to use it as such:

```
$ muddle unstamp simple_v1.0.release
```

This will tell you that it is ignoring the `RELEASE` section, which it has no use for.

12.7 Version include files

If you want to produce a version.h file, so that C or C++ programs can report the build version, then you can use `muddle subst` and the release specific environment variables.

For instance, create a template `version.h.in` file for "project99":

```
#ifndef PROJECT99_VERSION_FILE
#define PROJECT99_VERSION_FILE
#define BUILD_VERSION "${MUDDLE_RELEASE_NAME}: $(MUDDLE_RELEASE_VERSION}"
#endif
```

(of course, a string constant might be more appropriate, depending on the language and situation).

In the `Makefile.muddle`, then, for instance, alter:

```
.PHONY: config
 config:
```

to be:

```
.PHONY: config
 config: $(MUDDLE_OBJ_INCLUDE)/version.h
```

and add a new rule:

```
$(MUDDLE_OBJ_INCLUDE)/version.h: version.h.in
    -mkdir -p $(MUDDLE_OBJ_INCLUDE)
    $(MUDDLE) subst $< $@
```

Remember that the `MUDDLE_RELEASE_` environment variables will be set to `(unset)` when the build is not a release build.

CHAPTER 13

Jottings

In an ideal world, I'd have time to write all the things about muddle that need writing down. This doesn't seem to be that world.

This section of the documentation is where I shall try to put useful notes - more or less what is typically called "Frequently Asked Questions", except that that seems to rather a high standard to aim for.

13.1 bash: <toolchain>/bin/arm-none-linux-gnueabi-gcc: No such file or directory

This isn't really a muddle issue, but it's very confusing when it happens.

What this normally means is that your are running 64-bit operating system (e.g., Ubuntu), but the toolchain has been built for 32-bit. The solution for Debian derived systems is normally to install `ia32-libs`:

```
sudo apt-get install ia32-libs
```

which provide compatibility.

It might be nice to put this into an `aptget` clause in the build description, but I'm not sure what happens if you attempt to install `ia32-libs` on a 32-bit OS.

13.2 ./autogen.sh: line 3: autoreconf: command not found

And this one means you need to install `autoconf` (for the GNU autotools):

```
sudo apt-get install autoconf
```

13.3 So what should I apt-get install?

Clearly one normally needs `build-essentials`, which provides C and C++ compilers, GNU make and suchlike.

Future versions of muddle are likely to take advantage of `rsync`, if its installed, to speed up copying of files.

Building the kernel is likely to want you to install `lzma`, which is unobvious.

If you insist on using `udev` (busybox's `mdev` is smaller and simpler to use, but you might not have a choice), then you'll probably *have* to install `docbook-xsl`, even if you try your hardest to stop it building any documentation.

It's not uncommon for FSF packages to insist of `texinfo`, even if you're trying not to build documentation. You may also need to install `autotool` and `libtool`.

13.4 bash or dash?

Current Ubuntu systems (is it all Debian derived systems?) default to `/bin/sh` being `dash`, rather than `bash`.

`dash` is a Posix compliant shell. It's much smaller than bash, and arguably simpler to understand.

`bash` is a monster, with inaccurate documentation, but it is the norm on many Linux distributions (and on Mac OS X).

Many software packages assume bash in their build scripts and Makefiles. FSF packages do so explicitly. It is also quite easy to accidentally require bash when writing shell scripts, as we're not normally conscious of where its extensions start. Unfortunatley, many of those same shell scripts start with `#! /bin/sh` when what they *mean* is `#! /bin/bash`.

> (A relatively common example of this is when scripts use `echo -e`, which is a valid switch for bash's built in `echo`, but not that in dash. Of course, another solution for this specific problem would be for the script to use the system `echo`, in `/bin/echo`, which probably does support `-e`.)

Unfortunately this means that building on Linux typically requires one to replace `sh` meaning dash with `sh` meaning bash.

To check, do:

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 2012-01-04 16:52 /bin/sh -> bash
```

...you can see I'd already done the conversion.

There are two "obvious" ways of fixing this:

1. Reconfigure dash:

   ```
   $ sudo dpkg-reconfigure dash
   ```

   and answer "no" when asked if it should install dash as `/bin/sh`.

2. Do it by hand (but please think about the commands before typing them, in case I got them wrong...):

   ```
   $ cd /bin
   $ sudo ln -sf bash sh
   $ ls -l sh
   ```

There's an argument to say that, if some of the Makefiles or build scripts in the system really do care, it is a good idea to test for this in the build description, and give an "up front" error, rather than a confusing A similar sort of thing to check that `/bin/sh` is a link to `bash` may also be worth doing - something like:

```
if os.path.realpath('/bin/sh') == '/bin/dash':
    raise GiveUp('/bin/sh is linked to dash, not bash - this will not work for some scripts')
```

or you might prefer to test on `!= '/bin/bash'`. Neither is particularly elegant.

13.5 Why didn't my `deploy` directory change?

The `deploy` directory (or, in other words, deployments) are treated differently than the rest of the build infrastructure.

When you do `muddle build` of a package, it generally works something like:

- take the sources from the appropriate checkouts in the `src` directory
- build in the appropriate `obj` directory
- copy useful stuff to the appropriate `install` directory

`muddle rebuild`, `muddle distrebuild` and so on all do variations on this sequence. None of them touch the `deploy` directory and its contents.

The commands that *do* are:

- the commands in the "deployment" categorie (see `muddle help categories`):
  - `muddle deploy` and `muddle redeploy`
  - `muddle cleandeploy`
- `muddle` at the top level, when there are default deployments defined in the build description.

  This last is because `muddle` at the top level is equivalent to `muddle buildlabel _default_deployments _default_roles` - that is, it will build the default deployments (but not any that are not default deployments).

If one is just building a build tree that one was given, then the build description should (hopefully!) have been written so that `muddle` at the top level does the right thing, producing the deployments that a normal user wants.

If one is developing within a build tree, however, redeploying is expected to be a rarer thing - one is normally doing the build/fix/build cycle until everything works, and all the depedencies appear to be correct, and *then* one redeploys. In this context, the `deploy` directory contains "the stuff to be deployed on the target board" (or whatever the target might be), and one doesn't expect to generate this until everything (more or less) appears to be building correctly.

13.6 How do I pull in a "meta" checkout?

Sometimes one has a "meta" checkout, which contains information that isn't actually built, but one always wants present. A typical example is a `src/docs/` directory, containing documentation.

The simplest way to make this always present is to use a fragment like the following:

```
checkouts.simple.relative(builder, co_name='docs')

null_pkg = null_package(builder, name='meta', role='meta')
package_depends_on_checkout(builder.ruleset,
                            pkg_name='meta', role_name='meta',
                            co_name='docs')

# And add it to our default roles
builder.add_default_role('meta')
```

in your build description.

This says:

- we have a simple checkout called `docs`

- we have a NULL package, called `meta`, in role `{meta}`. A NULL package has "empty" rules for how to build it, so it will never actually do anything.
- we make this NULL package depend on the checkout (and if we had other "meta" checkouts, we could make it depend on each of those as well, so you only need one `package:meta{meta}`)
- and we add the `{meta}` role as one of our default roles. It will thus be built by a bare `muddle` command at the top of the build tree, which is normally how people build the whole tree.

Of course, just declaring the checkout means that `muddle checkout _all` would pull it out, but pulling it in via a default role makes it even more likely that the `docs` directory will get instantiated.

13.7 Build out-of-tree. Please.

We recommend going to lengths to do all builds out-of-tree.

That is, don't build in the checkout directory, build in the obj/ directory for your package.

There are two main reasons for doing this:

1. The traditional reason for building out of tree is that it allows one to build for several different configurations, and this is especially important in a muddle build, where one may build a single package in more than one role. Indeed, it's not uncommon to build a package for more than one architecture.
2. The other reason is clarity and simplicity. It's very useful to have all build artefacts sequestered from the original code. And moreover, this works best when using version control systems as well. If the build scatters files around in the checkout source directory, you're going to have to amend the *.gitignore*' file (or equivelant) so the VCS doesn't think there are un-checked-in files around. And `muddle push` is deliberately cautious about pushing in such circumstances.
3. The final reason is that you can recursively delete the `obj/` directory and *know* you've got rid of build artefacts. Which is uncommonly useful.

Many packages are quite capable of building out-of-tree (and many, for instance the Linux kernel, actively recommend it themselves).

However, if you have to build a checkout that does not support it (sadly, KBUS has been a problem here), then the simple and obvious solution is to use `$(MUDDLE) copywithout` and just copy the whole source tree (or those bits needed) into the obj/ tree.

So, for instance, from a Makefile.muddle for `zlib`:

```
config:
        -rm -rf $(MUDDLE_OBJ_OBJ)
        -mkdir -p $(MUDDLE_OBJ_OBJ)
        $(MUDDLE) copywithout $(MUDDLE_SRC) $(MUDDLE_OBJ_OBJ) .git
        (cd $(MUDDLE_OBJ_OBJ); ./configure --shared)
```

13.8 How do I get back to a clean checkout state?

The following strongly assumes all your builds are out-of-tree (see above).

If you understand how muddle works (see the chapter on "muddle and its directories"), then it should be obvious that you want to do:

```
$ rm -rf obj/ install/ deploy/
$ rm -rf .muddle/tags/package .muddle/tags/deployment
```

(for some value of "obvious"). Unfortunately, it is very easy to get this wrong (I've accidentally deleted my `src/` directory before now), and also this doesn't take account of any subdomains in a build tree.

The solution is the muddle command:

```
$ muddle veryclean
```

which does exactly those operations in a simple build, but also recurses through subdomains if necessary (see `muddle help veryclean` for details).

This approach is to be preferred to:

```
muddle distclean _all
muddle cleandeploy
```

because the first actually only obeys the "distclean" target in each Makefile.muddle, which won't necessarily do what (or as much) as you want.

13.9 How do I build my kernel?

Muddle has a special bit of infrastructure advertised for building the Linux kernel. It seemed like a good idea at the time, but it's overcomplex and doesn't work well for cross-compilation. So we now recommend something like the following:

1. Keep your kernel source tree as a GIT checkout. If you need to do any patches, do them directly into that checkout (don't, for instance, keep a directory of patches and try to apply them. We've tried this, it doesn't scale, and heh, it's actually what GIT automates for you. We tried it so you didn't need to.). Use branches to describe what you're doing as well. In other words, treat your local version of the kernel source tree just like normal kernel developers do.
2. The following fragment of a build description may be a useful guideline. We're assuming cross-compiling for ARM here.

   We start with some useful headers and constants:

```
import muddled.pkgs.make as make
import muddled.pkg as pkg

from muddled.utils import LabelType, LabelTag

# The name of the kernel checkout, and which branch we're using therein
# (note that this is likely the only branch that will get cloned)
LINUX_CHECKOUT_BRANCH = 'linux-3.2.0'

# Kernel config - obviously this needs to name an existing file!
TARGET_KERNEL_CONFIG = 'arch/arm/configs/some_evm_defconfig'
```

   and then later on actually define the kernel:

```
make.twolevel(builder, 'kernel', roles=['base'], co_dir='base',
              co_name='kernel', branch=LINUX_CHECKOUT_BRANCH)
```

   the following can be a nice trick, if most people are just going to be building the kernel, and not changing its sources:

```
# If we don't have KERNEL_DEEP_CLONES set in our environment, then we want
# a shallow clone - this *much* reduces the time it takes to clone the
# kernel sources, but then means we can't "muddle push" from it.
#
```

```
    # Users can set KERNEL_DEEP_CLONES by, for instance, doing:
    #
    #    export KERNEL_DEEP_CLONES=yes
    #
    # in their .bashrc.
    #
    if 'KERNEL_DEEP_CLONES' not in os.environ:
        pkg.set_checkout_vcs_option(builder, Label(LabelType.Checkout, 'kernel'),
                                    shallow_checkout=True)
```

If you're using `u-boot`, then something like the following may also be needed. I'm not bothering to show the description for `u-boot` itself.

```
    # Building a kernel image suitable for use with u-boot depends on the
    # mkimage program, provided by u-boot, so our kernel build depends on it
    rule = depend.depend_one(None,
                    Label(LabelType.Checkout, 'kernel', 'base', LabelTag.Configured),
                    Label(LabelType.Package, 'u-boot', 'boot', LabelTag.Installed))
    builder.ruleset.add(rule)

    # We also want to tell the kernel build what architecture and
    # configuration file it should use.
    pkg.append_env_for_package(builder, 'kernel', ['base'],
                               'TARGET_CPU', TARGET_ARCH)
    pkg.append_env_for_package(builder, 'kernel', ['base'],
                               'TARGET_KERNEL_CONFIG', TARGET_KERNEL_CONFIG)
```

3. If other packages need to know the kernel label, then give it to them "by hand" - for instance, for a notional package `driverXX`:

```
    pkg.append_env_for_package(builder, 'driverXX', ['drivers'],
                               'KERNEL_PACKAGE_LABEL',
                               str(Label(LabelType.Package, 'kernel', 'base',
                                         LabelTag.PostInstalled)))
```

4. Finally, a Makefile.muddle that looks something like:

```
    # Build a kernel.
    #
    # We put a couple of useful scripts in the obj dir:
    #
    #  do_menuconfig - does a 'make menuconfig' with all the right options.
    #  do_install_new_config - copies the current config back to the source
    #  directory.

    KERNEL_SRC=$(MUDDLE_SRC)
    KERNEL_BUILD=$(MUDDLE_OBJ_OBJ)
    MAKE_CMD=$(MAKE) -C $(KERNEL_SRC) O=$(KERNEL_BUILD) ARCH=arm \
            CROSS_COMPILE=$($(TARGET_CPU)_PFX)

    UBOOT=$(shell $(MUDDLE) query dir package:u-boot{boot}/*)
    UBOOT_TOOLS=$(UBOOT)/u-boot/u-boot-tools

    all:
            PATH=$$PATH:$(UBOOT_TOOLS) $(MAKE_CMD) uImage
            $(MAKE_CMD) INSTALL_MOD_PATH=$(MUDDLE_INSTALL) modules

    install:
            mkdir -p $(MUDDLE_INSTALL)/boot
```

```
            install -m 0744 $(KERNEL_BUILD)/arch/$(KERNEL_ARCH)/boot/uImage $(MUDDLE_INSTALL)/boot/u
            $(MAKE_CMD) INSTALL_MOD_PATH=$(MUDDLE_INSTALL) modules_install
            $(MAKE_CMD) INSTALL_HDR_PATH=$(MUDDLE_INSTALL) headers_install
            $(MAKE_CMD) INSTALL_FW_PATH=$(MUDDLE_INSTALL)/firmware firmware_install

    config:
            -mkdir -p $(KERNEL_BUILD)
            echo '$(MAKE_CMD) menuconfig' > $(KERNEL_BUILD)/do_menuconfig
            chmod +x $(KERNEL_BUILD)/do_menuconfig
            echo 'cp $(MUDDLE_OBJ_OBJ)/.config $(KERNEL_SRC)/$(TARGET_KERNEL_CONFIG)' >$(KERNEL_BUIL
            chmod +x $(KERNEL_BUILD)/do_install_new_config
            install -m 0644 $(KERNEL_SRC)/$(TARGET_KERNEL_CONFIG) $(MUDDLE_OBJ_OBJ)/.config
            $(MAKE_CMD) oldconfig

    clean:
            $(MAKE_CMD) clean

    distclean:
            rm -rf $(MUDDLE_OBJ)
```

(if you're very luck I've got that right - this is a generalisation of some more specialised makefiles, so I'll not be surprised if it has a bug or so).

13.10 How do I change my kernel configuation?

This depends on whether one means temporarily or permanently...

Typically, we recommend building the kernel out-of-tree.

Thus, given a kernel package called `package:kernel{base}`, and building for architecture `arm`, one can do:

```
cd obj/kernel/base/obj
make menuconfig ARCH=arm
```

(or `gconfig` or whatever one prefers as the interface - menuconfig is really the worst choice)

This will create an updated `.config` in that directory, and doing `muddle rebuild kernel{base}` should then rebuild using that configuration.

Beware that `muddle distrebuld kernel{base}` would do a *reconfigure* first, which will obey the `configure` target in your Makefile.muddle, which is probably copying in the (unchanged) .config from somewhere else...

And the two scripts `do_menuconfig` and `do_install_new_config` propagated by the Makefile.muddle in the previous section may also be of use.

13.11 How do I update a shallow checkout?

Shallow checkouts (or clones) were discussed in *How do I build my kernel?*.

In older versions of muddle, `muddle pull` refused to update shallow checkouts, so if a change occurs, it was necessary to retrieve it by hand.

Newer versions of muddle will allow `muddle pull` (and `muddle merge`), but still forbid `muddle push` (because git won't allow it).

If you do need to update a shallow checkout, it is simplest to re-checkout as a "full" checkout, and then carry on as normal. For instance, edit the build description to remove the "shallow" flag for the checkout, and then (assuming we're working on `src/linux/kernel`):

```
$ cd src/linux
$ mv kernel kernel.save      # remove the directory (but save it just in case)
$ muddle uncheckout kernel   # to tell muddle the checkout has gone away
$ muddle checkout kernel     # to check it out again
$ rm -rf kernel.save         # we no longer need the saved directory
```

13.12 How can I make sure the correct toolchain is available?

Let us assume you have:

```
ARM_TOOLCHAIN = "/opt/toolchains/arm-2010q2"
```

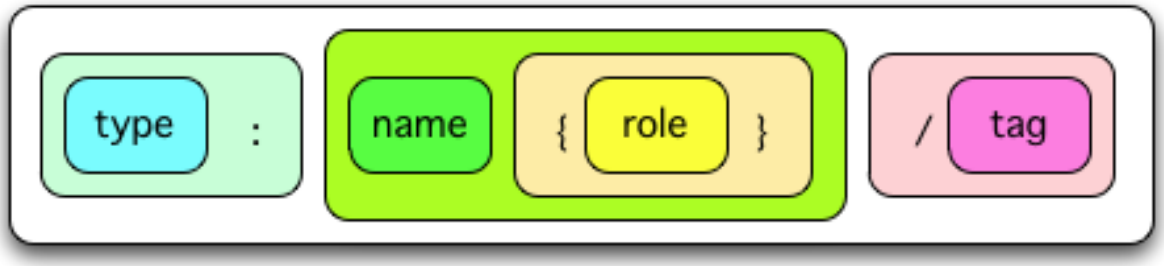
to set up appropriate constants.Then later on (in your `describe_to()` function), you can do:

```
if not os.path.exists(ARM_TOOLCHAIN):
    raise GiveUp("Required toolchain %s not present"%ARM_TOOLCHAIN)
```

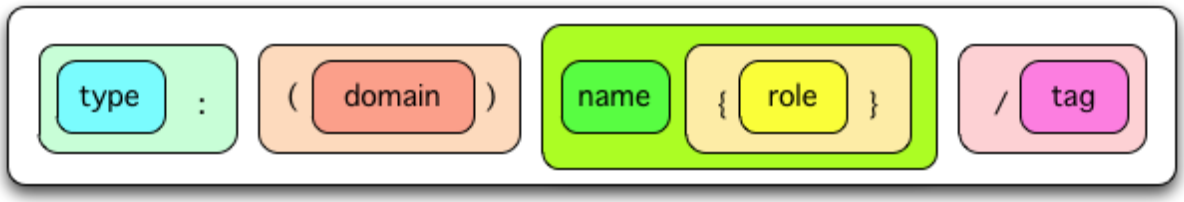
This is rather horrible, but may be less worse than the alternative.

13.13 Do you have a picture of a label?

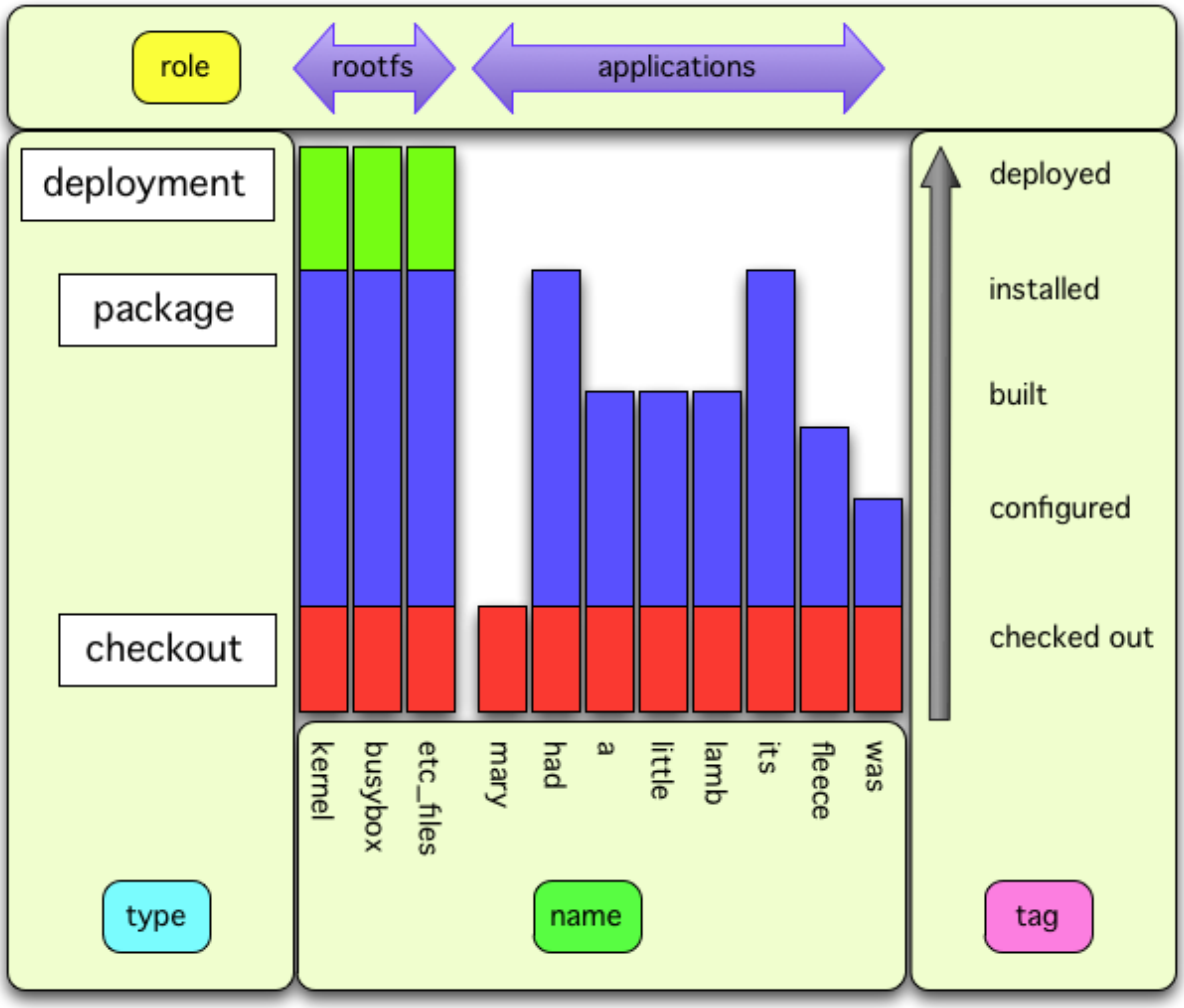
Why, yes I do. Here is a label without domains (a "normal" label):



and here is a label with domains:



There's even an attempt to show how these relate to the build sequence:



although I'm not entirely sure if that's helpful or not.

13.14 Mechanics: how "promotion" of subdomain labels works

Warning: This is a note on the techincal innards of muddle, written in part for my own consumption. Please feel free to ignore it.

The code being discussed here is centred around functions in `muddled/mechanics.py`.

When muddle includes a subdomain (via the `include_domain()` function), it has to change all the labels in the subdomain (e.g., `checkout:fred/*`) into valid, equivalent labels in the including domain (so, `checkout:(subdomain)fred/*`).

The basic approach is that the function `include_domain()` calls `_new_sub_domain()` to do nasty underhand things to the labels in the subdomains dependency tree, and then merges the subdomain dependency tree into the main tree.

To allow this, every entity that might be hoarding labels (using them in sets or list or dictionaries or whatever) must have a way of returning *all* of those labels. For Actions, this method is called `_inner_labels()`.

So `_new_sub_domain()` retrieves labels from various Well Known Places within the subdomain's Builder, and from all of the actions, and from the dictionaries managed for that subdomain by `db.py`, and puts them into a single grand list.

We rely on the fact that we have *every* label - so for instance, we might have a list starting and ending like:

```
<obj id   99> "checkout:fred/*"
<obj id 1274> "checkout:marshmallow/checked_out"
<obj id  321> "package:wholemeal{x86}/built"
...
<obj id   99> "checkout:fred/*"
...
<obj id 5823> "checkout:fred/*"
...
```

In this list we note that: (a) we have more than one occurrence of object id `99` (object ids are obviously made up for this purpose and do not represent real objects), and that we also have two different objects that represent `checkout:fred/*` - these will compare as equal, of course. We don't care if we have more than one occurrence of the identical object, but we *really do* care that we have at least one occurrence of every non-identical label object.

The next bit is very simple and slightly horrible.

We know all of these label objects are from the subdomain, and we want to make them correct for the main domain. So we:

1. Go through the list and mark each label object "unswept". Doing this more than once on the same label object is alright, we can't make them any more unswept.
2. Go through the list and for each label object, if it is still marked "unswept":
   (a) amend the `domain` field to add in the subdomain's name - so a label called `checkout:fred/*` would become `checkout:(subdomain)fred/*`, and a label called `checkout:(earlier)jim/*` would become `checkout:(subdomain(earlier))jim/*`.
   (b) undo the "unswept" marker

Once we've done that, we've promoted every single label to the new domain, in-situ.

All that then remains is for `include_domain()` to merge all the rules, dictionaries and so on into the main domain. We can then discard the old Builder, since they're no longer of any interest.

CHAPTER 14

Issues with GNU autotools

Many source packages use GNU autotools to manage their building. You can typically recognise this because their build instructions tell you to:

```
./configure; make; make install
```

and you will typically see files called `configure.ac` and `Makefile.in` in the source tree.

The GNU autotools derive from a time when writing portable C code was quite difficult, and typically required hand-editing a Makefile (often derived from the X11 build system) in arcane ways. They have, in many ways, been a great success, and for most people, most of the time, are a good thing.

However, their decision to rely only on `bash`, `m4` and GNU `make` renders the system somewhat cumbersome (the original designers did not want to add yet another tool which would need to be built and/or distributed - in retrospect a mistake). The lack of adequate documentation, the tendency of packages to "copy" mostly working autotools setups from other packages, and the way that changes to autotools propagate do not help (there's a rant there, which I shall try to avoid - it's very boring).

Regardless, when one is assembling a muddle build system, various common problems keep reappearing, and this chapter of the documentation is meant to be a place where we can keep a note of such problems, and their solutions.

14.1 Resources

There are, at time of writing, two invaluable resources for learning about autotools.

The first is:

> **Autotools: A practitioner's guide to GNU AUTOCONF, AUTOMAKE and LIBTOOL**
> John Calcote
> No Starch Press, 2010
> ISBN 1-59327-206-5

This is an introduction to the toolset, with a worked explanation on how to setup a new project (or convert an existing one) to use autotools, written by someone who believes strongly in the system. If you need to use autotools it is worth finding a copy.

The second is online:

> Autotools Mythbuster
> https://autotools.io/
> Diego Elio "Flameeyes" Pettenò

His blog, at https://blog.flameeyes.eu/, can also be fairly instructive.

14.2 Muddle won't pull because files have changed

Typically, muddle will grumble that:

```
Failure pulling checkout:SOMENAME/pulled[T] in src/libs/SOMENAME:
There are uncommitted changes/untracked files
     M Makefile.in
     M aclocal.m4
     M configure
     M include/Makefile.in
```

Useful resources for an explanation include:

- http://blog.flameeyes.eu/2008/06/maintaner-mode
- http://www.flameeyes.eu/autotools-mythbuster/automake/maintainer.html
- http://stackoverflow.com/questions/5731023/autotools-force-make-not-to-rebuild-configure-makefile

There are several things going on that cause the problem:

- Various revision control systems (the stackoverflow thread talks about CVS, but this is true of git as well) do not preserve timestamps when creating working files. This means that the relative timestamps of files in a newly clone git directory are not necessarily predictable, and are certainly not guaranteed to be related to the timestamps in the original checkin.
- The main operating system may have a different version of autotools installed (or its constituent packages) than are present/referenced in the checkout SOMENAME.
- There is a thing called "maintainer mode" in the autotools world. This can cause various source files to be regenerated if the autotools system believes this to be necessary. How it works has changed over time, as is often the case with autotools.
- The `configure` script is generated from `configure.ac` (which is much more readable!), and the `Makefile.in` files are generated from `Makefile.am` templates.

So, it is probable that this source checkout SOMENAME has a `configure.ac` which either does not contain the `AM_MAINTAINER_MODE` macro (causing it to default to enabled), or else does contain it but explicitly enables maintainer mode.

Since maintainer mode is enabled, running `./configure` can trigger a check to see whether the `configure` script is "more recent" than its `configure.ac`, and similarly for other derived files. A check may also be made against the system autools setup. If the check makes it look as if something is out of date, then the autotools system can decide to regenerate files *in the source directory*, leading to our problem.

There are three possible solutions, of varying complexity.

1. The Makefile.muddle is building with reference to the source directory - for instance:

```
    config:
            -mkdir -p $(MUDDLE_OBJ_OBJ)
            (cd $(MUDDLE_OBJ_OBJ); \
                $(MUDDLE_SRC)/configure  --host=armv7-none-linux-gnueabi \
                --prefix=/  )
```

   and thus the autotools system can overwrite files in that source directory. So a simple, if blunt, solution is to copy the source tree first, and any amendments will then be made in that copy:

```
    config:
            -mkdir -p $(MUDDLE_OBJ_OBJ)
            $(MUDDLE) copywithout $(MUDDLE_SRC) $(MUDDLE_OBJ_OBJ) .git
            (cd $(MUDDLE_OBJ_OBJ); \
```

```
                ./configure  --host=armv7-none-linux-gnueabi \
                --prefix=/  )
```

The snag with this approach is that if a file in the source directory is edited, `muddle reconfigure` is needed to copy the entire source tree over again.

However, this is undoubtedly the simplest and quickest solution.

Note: This is a traditional approach, which is known to work, although with the disadvantages mentioned.

---

2. If the `configure.ac` file *does* include the `AM_MAINTAINER_MODE` macro, then it *should* be possible to specify `--disable-maintainer-mode` to the `configure` step:

```
config:
        -mkdir -p $(MUDDLE_OBJ_OBJ)
        (cd $(MUDDLE_OBJ_OBJ); \
            $(MUDDLE_SRC)/configure  --host=armv7-none-linux-gnueabi \
            --disable-maintainer-mode \
            --prefix=/  )
```

(Note that the GNU autotools documentation does not mention the limitation that the `AM_MAINTAINER_MODE` macro needs to be present for this to work).

Unfortunately, many packages do not include that macro in their `configure.ac`, and there is some debate in the community as to whether it *should* be there or not (for instance, see http://old.nabble.com/-PATCH-libwacom–Drop-AM_MAINTAINER_MODE-td34561358.html).

> **Warning:** This approach should work, but I have not personally had the opportunity to try it, as most of the packages I've had to fix were either before I discovered this solution, or have not had `AM_MAINTAINER_MODE` defined.

3. If the `configure.ac` does *not* contain the `AM_MAINTAINER_MODE` macro, it is possible to add it, regenerate the autotools setup for the package, and then commit the lot. The `AM_MAINTAINER_MODE` macro can then either explicitly disable maintainer mode, or enable it and the `configure` usage can disable it with the `--disable-maintainer-mode` switch.

   As an example of someone external doing this sort of thing, see this pull request on github: https://github.com/craftIk/craftIk/pull/12

   To regenerate the `configure` and other files, use `autoreconf` - this is silent without `-v` switch:

```
cd SOMENAME
autoreconf -v
```

Note that this will typically generate a diretory called `autom4te.cache`, which you should add to your `.gitignore` (and perhaps also delete after use).

> **Warning:** This is an inadequate summary of what to do, and may be misleading. I hope to improve it in a later version of this document...

CHAPTER 15

Muddle patch

`muddle_patch.py` is an independent program in the same directory as the normal `muddle` program.

15.1 Summary

This program was written to address issue 111: *Implement "diff build tree against version stamp" and corresponding patch command*.

Specifically, given an up-to-date muddle build tree, and the stamp file for a less up-to-date build tree, it attempts to work out the patches to make the latter into a copy of the former. When it can, it tries to do this in such a way that the underlying VCSs will really believe it.

- In this document, the local (most up-to-date) build tree is "our" build tree, and the other (to be updated) build tree is the "far" build tree.
- If a checkout is present in our build tree, but absent in the far build tree, then it is NEW, and a copy of the current state of the checkout directory will be transferred. Note that this script will not mark a NEW checkout as checked out (see below).
- If a checkout is absent in our build tree, but present in the far build tree, then it is DELETED. `muddle_patch.py` does nothing about this - it is up to the user to delete the checkout at the far end if necessary.
- If a checkout has changed (just) its revision, then it is CHANGED, and an attempt will be made to determine the differences in the best way possible for the VCS concerned. Again, see below.
- If a checkout has changed anything else (its location in the build tree, its domain, or the VCS it is using), then it will be reported as a problem, and will not have a patch generated for it. It may be that this is overkill, and some of these should be coped with - if you have a good use-case, please raise it as an issue.

Any checkouts with problems in either stamp file will not be dealt with - in other words, both build trees should really be in a state suitable for `muddle stamp version` before using `muddle_patch.py`.

15.2 Writing a patch directory

The command line is:

```
muddle_patch.py write [-f[orce]] <our_stamp> <far_stamp> <patch_dir>
```

<patch_dir> is the directory to which the patch data will be written. If `-force` is given, then an existing <patch_dir> will be deleted, otherwise <patch_dir> may not already exist.

<our_stamp> is either a stamp file describing the current build tree, or -. In the latter case, a "virtual" stamp file will be built in memory and used as "our" stamp file. This virtual stamp file is equivalent to that which would be produced by `muddle stamp save -f`.

<far_stamp> is the stamp file for the far build tree.

`muddle_stamp.py write` must be run in the top-level directory of "our" build tree - i.e,, the directory containing the `.muddle/` and `src/` directories. It is not supported to run the program in a sub-domain.

For instance:

```
muddle_patch.py  write  -f  -  ../far.v8_missing.stamp  ../v8-patches/
```

15.3 Using a patch directory

The command line is:

```
muddle_patch.py read <patch_dir>
```

<patch_dir> is the directory containing the patch data.

`muddle_stamp.py read` must be run in the top-level directory of the "far" build tree - i.e,, the directory containing the `.muddle/` and `src/` directories. It is not supported to run the program in a sub-domain.

Note: If some part of `muddle_patch.py read` fails (for instance, if an attempt is made to write a NEW checkout but the target directory already exists) then `muddle_patch.py` will give up immediately.

The current version of the code does not know how to continue from a partial "read", so will typically fail if run again (since the first amendment it tries to make will presumably now fail because the relevant patch has already been applied).

The simplest thing to do in this case is probably to edit the MANIFEST.txt file in the <patch_dir> and comment out (using '#') the lines for those checkouts that have already been patched successfully.

Some future version of the program may be more robust.

15.4 How it works (or doesn't)

15.4.1 The MANIFEST.txt file

In the <patch_dir>, `muddle_patch.py` creates a file called MANIFEST.txt. This is in the traditional INI format, and contains an entry for each checkout described in the <patch_dir>, indicating its VCS and other useful information.

Typical entries look like:

```
[BZR helpers]
checkout=helpers
directory=None
patch=helpers.bzr_send
old=1
new=3
[TAR v8]
checkout=v8
directory=platform
patch=v8.tgz
```

There is nothing very magical about this file, and it may sometimes be useful to edit it. Lines starting with '#' or ';' are commented out, and will be ignored. Lines may not start with whitespace.

Entries are of the form:

- `[<vcs> <name>]` – where <vcs> is one of BZR, SVN, GIT or NEW, and <name> is the checkout name.
- `checkout=<name>` – again, <name> is the checkout name.
- `directory=<subdirectory>` – <subdirectory> is either None, or the name of the subdirectory of `src/` where the checkout directory may be found (so if the checkout is `src/platforms/v8`, then <subdirectory> would be `platforms`).
- `patch=<filename>` names the patch file (or, for GIT, the patch directory) in <patch_dir>
- `old=<revision>` and `new=<revision>` give the old and new revision ids (or equivalent) for the two checkouts. "new" corresponds to "our" checkout, and "old" to the "far" checkout. These are not specified for NEW patches.

15.4.2 Changed bzr checkouts

The problem

The original version of this program was tested, and seemed to work as described in *How it should be* below.

Unfortunately, on retesting with *bzr*' 2.1.1 and 2.2.1, it no longer seem to work, instead exploding with a `bzrlib.errors.NoSuchRevision` exception. At time of writing I am still investigating this, in the fond hope that it is something I am doing wrong, rather than a bug in Bazaar (although an exception rather than an error report still seems rather nasty).

Thus for the moment, the program is instead doing *What we do instead*.

If you wish to change the behaviour back to the original, then edit the `muddle_patch.py` file, and change:

```
BZR_DO_IT_PROPERLY = False
```

to:

```
BZR_DO_IT_PROPERLY = True
```

Note: The `BZR_DO_IT_PROPERLY` flag actually only affects the "write" operation. The "read" operation actually decides what to do based on the extension of the `<checkout>.bzr_send` or `<checkout>.diff` file in the patch directory.

How it should be

When writing a changed Bazaar checkout, `muddle_patch.py` uses:

```
bzr send
```

to create a file (in <patch_dir>) containing the differences between the two revisions of the checkout. The file will be named `<checkout>.bzr_send`, where `<checkout>` is the muddle checkout name.

When reading the <patch_dir>, `muddle_patch.py` uses:

```
bzr merge --pull
```

to update the checkout. This should produce a result identical to having done an appropriate merge/pull. The use of `merge --pull` means that Bazaar should attempt to do a pseudo-pull of the changes, and only revert to "merge" behaviour if it has to.

Note: Determine if a `bzr commit` is still required at this stage.

What we do instead

A simple difference file is generated, using:

```
bzr diff -p1
```

to create a file (in <patch_dir>) containing the differences between the two revisions of the checkout. The file will be named `<checkout>.diff`, where `<checkout>` is the muddle checkout name.

When reading the <patch_dir>, `muddle_patch.py` uses:

```
patch -p1
```

to update the checkout. This will produce a result similar to that in the "near" checkout, with two important exceptions:

1. History is not propagated, and
2. Any new but empty files in the "near" checkout will not be created in the "far" checkout. This is a limitation of the diff/patch sequence we have available.

After the patch has succeeded, it will be necessary to do a `bzr add` (to catch any new files), followed by a `bzr commit` to actually commit the result.

If any files have been removed by the patch process, these will have to be manually removed from bazaar.

15.4.3 Changed svn checkouts

When writing a changed Subversion checkout, `muddle_patch.py` uses:

```
svn diff
```

to create a file (in <patch_dir> containing the differences between the two revisions of the checkout.

When reading the <patch_dir>, `muddle_patch.py` uses:

```
patch
```

to update the checkout. This should result in a checkout with the same source file content as wished, but of course the Subversion metadata will not have been changed, so Subversion will think it has a different revision number.

15.4.4 Changed git checkouts

When writing a changed git checkout, `muddle_patch.py` uses:

```
git format-patch
```

to create a diretory (in <patch_dir> containing the differences between the two revisions of the checkout.

When reading the <patch_dir>, `muddle_patch.py` uses:

```
git am
```

to update the checkout.

`git am` leaves HEAD detached, so this needs fixing – for instance, if one *was* on branch `master`, one might do:

```
git branch post-update-branch   # to reattach HEAD to a branch
git checkout master             # to go back to master, if that's correct
git merge post-update-branch    # to merge in our new stuff to master
```

Note: The writer of this document is not a git expert, so please treat the above with caution.

15.4.5 New checkouts

When writing a NEW checkout, `muddle_patch.py` uses `tar` to create a gzipped tarfile for the checkout source directory, in <patch_dir>. This contains all of the content of that directory, whether under version control or not (and it also includes any "magic" VCS directories, such as `.bzr/`).

When reading the <patch_dir>, `muddle_patch.py` simply uses `tar` to unzip and unpack the checkout in the appropriate place. It will refuse to do this if a directory of the right name already exists (i.e., it does not overwrite an existing directory).

It will not attempt to add the new checkout to any version control system.

CHAPTER 16

Useful `muddle` classes, methods and functions

This is the start of a section describing just those parts of the `muddle` API that are generally likely to be useful in build descriptions. For the moment, much of it is just duplicates of the docstring descriptions already found in the next section, The muddled package.

16.1 An example build description

```
#! /usr/bin/env python

"""Muddle build description for a "plain" OMAP build on the Beagleboard.

At least initially, we're trying to build a Linux+busybox system.

The role we're providing is 'omap'.
"""

import os

import muddled.deployments.filedep as filedep
import muddled.pkgs.aptget as aptget
import muddled.checkouts.simple
import muddled.depend
from muddled.depend import label_from_string
import muddled.pkgs.make as make

import wget
import repo

def describe_to(builder):

    role = 'omap'
    roles = ['omap']

    builder.add_default_role(role)

    # filedep.deploy(builder, target_dir, name, roles)
    #
    # Register a file deployment.
    #
    # The deployment will take the roles specified in the role list, and build
    # them into a deployment at deploy/[name].
```

```
    #
    # The deployment should eventually be located at target_dir.

    filedep.deploy(builder,
                   '/',
                   'omap',
                   roles)

    # There's a variety of things we need on (this, the host) system
    # in order to build - I hope I've got this right (difficult to tell
    # as I already have some stuff installed on *my* development system)
    aptget.simple(builder, 'development_packages', role,
            ['zlib1g-dev', 'uboot-mkimage'])

    # According to http://omappedia.org/wiki/LinuxOMAP_Kernel_Project, we get
    # our OMAP kernel from:
    muddled.checkouts.simple.absolute(builder, 'omap_kernel',
            'git+git://git.kernel.org/pub/scm/linux/kernel/git/tmlind/linux-omap-2.6.git')

    # Once we've got one worked out, we'll also want to retrieve a default
    # kernel configuration (we don't, eventually, want to make the developer
    # have to work that out every time!)

    # We'll aim to make that with an out-of-tree makefile
    # We could make a tailored subclass of muddled.pkgs.linux_kernel, and use
    # that to build our kernel. I may still do that once I've figured out how
    # it is different (one notable change is we're building uImage instead of
    # zImage). For now, it's probably easier to have a Makefile.muddle
    make.medium(builder,
                name = "omap_kernel",    # package name
                roles = roles,
                checkout = "builders",
                deps = [],
                makefileName = os.path.join("omap_kernel","Makefile.muddle"))

    muddled.pkg.package_depends_on_checkout(builder.ruleset,
                                    "omap_kernel",  # this package
                                    role,           # in this role
                                    "omap_kernel")  # depends on this checkout

    # On top of that, we want to build busybox
    #
    # According to the busybox website, we can retrieve sources via git:
    #
    # To grab a copy of the BusyBox repository using anonymous git access::
    #
    #   git clone git://busybox.net/busybox.git
    #
    # Once you have the repository, stable branches can be checked out by
    # doing::
    #
    #   git checkout remotes/origin/1_NN_stable
    #
    # Once you've checked out a copy of the source tree, you can update your
    # source tree at any time so it is in sync with the latest and greatest by
    # entering your BusyBox directory and running the command::
    #
    #   git pull
```

```
    # So, for the moment, at least, let's go with the latest from source
    # control (when we're finalising this, we're maybe better identifying
    # a particular release to stick to)
    muddled.checkouts.simple.absolute(builder, 'busybox',
            'git+git://busybox.net/busybox.git')

    # We'll aim to make that with an out-of-tree makefile
    #
    # 'deps' is a list of package names, which our make depends on.
    #
    # Specifically, for each <name> in 'deps', and for each <role> in 'roles',
    # we will depend on "package:<name>{<role>}/postinstalled"
    make.medium(builder,
                name = "busybox",     # package name
                roles = roles,
                checkout = "builders",
                deps = [ 'omap_kernel' ],
                makefileName = os.path.join("busybox","Makefile.muddle"))

    # And we also depend on having actually checked out busybox
    muddled.pkg.package_depends_on_checkout(builder.ruleset,
                                    "busybox",      # this package
                                    role,           # in this role
                                    "busybox")      # depends on this checkout


    # Can we use the same bootloader and such that we already had for the
    # android build?

    # The bootloader and related items, which go into the FAT32 partition on
    # the flash card, are retrieved from the net (eventually, I hope we'll be
    # building u-boot, but for now the binary should do)
    muddled.checkouts.simple.absolute(builder, 'MLO',
            'wget+http://free-electrons.com/pub/demos/beagleboard/android/MLO')
    muddled.checkouts.simple.absolute(builder, 'u-boot',
            'wget+http://free-electrons.com/pub/demos/beagleboard/android/u-boot.bin')

    # We need some way of getting them installed - let's foreshadow the day when
    # we actually want to build u-boot ourselves, and pretend
    make.medium(builder,
                name = "u-boot",     # package name
                roles = roles,
                checkout = "builders",
                deps = [],
                makefileName = os.path.join("u-boot","Makefile.muddle"))
    muddled.pkg.package_depends_on_checkout(builder.ruleset,
                                    "u-boot",       # this package
                                    role,           # in this role
                                    "u-boot")       # depends on this checkout
    # Oh, and this one as well...
    rule = muddled.depend.depend_one(None,
                               label_from_string('package:u-boot/installed'),
                               label_from_string('checkout:MLO/checked_out'))
    builder.ruleset.add(rule)


    # And, of course, we need (the rest of) our Linux filesystem
    muddled.checkouts.simple.absolute(builder, 'rootfs',
```

```
            'bzr+ssh://bzr@palmera.c.kynesim.co.uk//opt/kynesim/projects/052/rootfs')
    make.simple(builder, 'rootfs', role, 'rootfs', config=False,
            makefileName='Makefile.muddle')

    # But we depend on busybox to have installed the various binaries first
    rule = muddled.depend.depend_one(None,
                                label_from_string('package:rootfs/installed'),
                                label_from_string('package:busybox/installed'))
    builder.ruleset.add(rule)

    # Deploy all our roles
    builder.by_default_deploy_list(roles)
```

16.2 muddled.mechanics.Builder

The `describe_to()` function in a build description takes a Builder instance as its argument. Thus the methods on the Builder class are all simply available. For instance:

```
builder.by_default_deploy_list(roles)
```

class `muddled.mechanics.`**`Builder`**(*root_path*, *muddle_binary*, *domain_params=None*, *default_domain=None*)

Bases: `object`

A builder does stuff following rules derived from a build description.

Don't construct a Builder directly, always use the 'load_builder()' function, or 'minimal_build_tree()' if that is more appropriate.

- self.db - The metadata database for this project.
- self.ruleset - The rules describing this build
- self.env - A dictionary of label to environment
- self.default_roles - The roles to build when you don't specify any. These will also be used for "guessing" a role for a package when one is not specified. '_default_roles' is calculated from this.
- self.default_deployment_labels - The deployments to deploy when you don't specify any specific roles. This is the meaning of '_default_deployments'.

There are then a series of values used in managing subdomains:

- self.banned_roles - An array of pairs of the form (role, domain) which aren't allowed to share libraries.
- self.domain_params - Maps domain names to dictionaries storing parameters that other domains can retrieve. This is used to communicate values from a build to its subdomains.
- self.unifications - This is a list of tuples of the form (source-label, target-label), where one "replaces" the other in the build tree.

And:

- self.what_to_release - a set of entities to build for a release build. It may contain labels and also "special" names, such as '_default_deployments' or even '_all'. You may not include '_release' (did you need to ask?). "_just_pulled" is not allowed either.

Construct a fresh Builder with a .muddle directory at the given 'root_path'.

'muddle_binary' is the full path to our muddle "binary" - the program that is muddle. This is needed when running muddle Makefiles that invoke $(MUDDLE).

'domain_params' is the set of domain parameters in effect when this builder is loaded. It's used to communicate values down to sub-domains.

Note that you MUST NOT set 'domain_params' Null unless you are the top-level domain - it MUST come from the enclosing domain's builder or modifications made by the subdomain's buidler will be lost, and as this is the only way to communicate values to a parent domain, this would be bad. Ugh.

'default_domain' is the default domain value to add to anything in local_pkgs , etc - it's used to make sure that if you're cd'd into a domain subdirectory, we build the right labels.

add_default_deployment_label(*label*)
Set the label that's built when you call muddle from the root directory

add_default_role(*role*)
Add role to the list of roles built when you don't ask for something different.

Returns False if we didn't actually add the role (it was already there), True if we did.

add_default_roles(*roles*)
Add the given roles to the list of default roles for this build.

add_to_release_build(*thing*)
Add a thing to the set of entities to build for a release build.

'thing' must be:

- a Label
- one of the "special" names, "_all", "_default_deployments", "_default_roles".
- a sequence of either/both

It may not be "_release" (!) or "_just_pulled".

Special names are expanded after all build descriptions have been read.

The special name "_release" corresponds to this set.

all_checkout_labels(*tag=None*)
Return a set of the labels of all the checkouts in our rule set.

Note that if 'tag' is None then all the labels will be of the form:

checkout:<co_name>/*

otherwise 'tag' will be used as the checkout label tag:

checkout:<co_name>/<tag>

all_checkout_rules()
Returns a set of the labels of all the checkouts in our rule set.

Specifically, all the rules for labels of the form:

```
checkout:*{*}/checked_out
checkout:(*)*{*}/checked_out
```

Returns a set of labels, thus allowing one to know the domain of each checkout as well as its name.

all_checkouts()
Return a set of the names of all the checkouts in our rule set.

Returns a set of strings.

This is not domain aware. Consider using all_checkout_labels(), which is.

all_deployment_labels(*required_tag*)
Return a set of all the deployment labels in our ruleset.

If 'required_tag' is given, then the labels returned will all have that tag (this, of course, may result in a smaller set of labels).

all_deployments()
Return a set of the names of all the deployments in our rule set.

Returns a set of strings.

all_domains()
Return a set of the names of all the domains in our rule set.

Returns a set of strings. The 'None' domain (the unnamed, top-level domain) is returned as the empty string, "".

all_package_labels()
Return a set of the labels of all the packages in our rule set.

all_packages()
Return a set of the names of all the packages in our rule set.

Returns a set of strings.

Note that if '*' is one of the package "names" in the ruleset, then it will be included in the names returned.

all_packages_with_roles()
Return a set of the names of all the packages/roles in our rule set.

Returns a set of strings.

Note that if '*' is one of the package "names" in the ruleset, then it will be included in the names returned.

However, any labels with role '*' will be ignored.

all_roles()
Return a set of the names of all the roles in our rule set.

Returns a set of strings.

apply_unifications(*source*)

build_co_and_path()
Return a pair (build_co, build_path).

build_desc_repo
The Repository for our build description.

This used to be a simple value, but is now a shim around looking it up in builder.db.checkout_data - so that we only have the information stored in one place.

Returns None if there is no Repository registered yet

build_label(*label*, *silent=False*)
The fundamental operation of a builder - build this label.

build_label_with_options(*label*, *useDepends=True*, *useTags=True*, *silent=False*)
The fundamental operation of a builder - build this label.

- useDepends - Use dependencies?

build_name
The build name is meant to be a short description of the purpose of a build. It might thus be something like "ProjectBlue_intel_STB" or "XWing-minimal".

The name may only contain alphanumerics, underlines and hyphens - this is to facilitate its use in version stamp filenames. Also, it is a superset of the allowed characters in a Python module name, which means that the build description filename (excluding its ".py") will be a legal build name (so we can use that as a default).

by_default_deploy(*deployment*)
Add a deployment label to the deployments to build by default.

by_default_deploy_list(*deployments*)
Now we've got a list of default labels, we can just add them ..

checkout_label_exists(*label*)
Return True if this checkout label is in any rules (i.e., is used).

Note that this method does not understand wildcards, so the match must be exact.

checkout_path(*label*)
Return the path in which the given checkout resides.

This is a simple wrapper around builder.db.get_checkout_path(), provided for use in scripts and build descriptions, since it "matches" builder.package_obj_path, builder.deploy_path, and so on.

checkouts_for_package(*pkg_label*)
Return a set of the checkouts that the given package depends upon

This only looks at *direct* dependencies (so if a package depends on something that in turn depends on a checkout that it does not directly depend on, then that indirect checkout will not be returned).

It does, however, expand wildcards.

deploy_path(*label*)
Where should deployment 'label' deploy to?

diagnose_unused_labels(*labels*, *arg*, *required_type=None*, *required_tag='*'*)
Concoct a useful report on why none of 'labels' is used.

We rely on 'labels' having been generated by our label_from_fragment() method, which means that all the labels will have the same type

We assume quite a lot of knowledge about how that method works.

effective_environment_for(*label*)
Return an environment which embodies the settings that should be used for the given label. It's the in-order merge of the output of `list_environments_for()`.

expand_release()
Expand the command line argument "_release"

expand_underscore_arg(*word*, *type_for_all=None*)
Given a command line argument ('word') that starts with an underscore, try to expand it to a list of labels.

If the argument is _all, then if 'type_for_all' is given, expand it to all labels of that type, and otherwise reject it.

Raises a GiveUp exception if the argument is not recognised.

expand_wildcards(*label*, *default_to_obvious_tag=True*)
Given a label which may contain wildcards, return a set of labels that match.

As per the normal definition of labels, the <type>, <name>, <role> and <tag> parts of the label may be wildcarded.

If default_to_obvious_tag is true, then if label has a tag of '*', it will be replaced by the "obvious" (final) tag for this label type, before any searching (so for a checkout: label, /checked_out would be used).

find_local_package_labels(*dir*, *tag*)
This is slightly horrible because if you're in a source checkout (as you normally will be), there could be several packages.

Returns a list of the package labels involved. Uses the given tag for the labels.

find_location_in_tree(*dir*)
Find the directory type and name of subdirectory in a repository. This is used by the find_local_package_labels method to work out which packages to rebuild

•dir - The directory to analyse

If nothing sensible can be determined, we return None. Otherwise we return a tuple of the form:

(DirType, label, domain_name)

where:

•'DirType' is a utils.DirType value,

•'label' is None or a label describing our location,

•'domain_name' None or the subdomain we are in and

If 'label' and 'domain_name' are both given, they will name the same domain.

follow_build_desc_branch

follows_build_desc_branch

get_all_checkout_labels_below(*dir*)
Get the labels of all the checkouts in or below directory 'dir'

NOTE that this will not work if you are in a subdirectory of a checkout. It's not meant to. Consider using find_location_in_tree() to determine that, before calling this method.

get_build_desc_branch(*verbose=False*)
Return the current branch of the top-level build description.

(Returns None if the build description is not on a branch, or if its VCS does not support this operation.)

get_default_domain()

get_dependent_package_dirs(*label*)
Find all the dependent packages for label and return a set of the object directories for each. Mainly used as a helper function by `set_default_variables()`.

get_distribution()
Retrieve the current distribution name and target directory.

Raises GiveUp if there is no current distribution set.

get_domain_parameter(*domain*, *name*)

get_domain_parameters(*domain*)

get_environment_for(*label*)
Return the environment store for the given label, inventing one if necessary.

get_labels_in_default_roles()
Return a list of the package labels in the default roles.

get_parameter(*name*)
Returns the given domain parameter, or None if it wasn't defined.

include_domain(*domain_builder*, *domain_name*)
Import the builder domain_builder into the current builder, giving it domain_name.

We first import the db, then we rename None to domain_name in banned_roles

instruct(*pkg*, *role*, *instruction_file*, *domain=None*)
Register the existence or non-existence of an instruction file. If instruction_file is None, we unregister the instruction file.

•instruction_file - A db.InstructionFile object to save.

is_release_build()
Are we a release build (i.e., a build tree created by "muddle release")?

We look to see if there is a file called .muddle/Release

kill_label(*label*, *useTags=True*, *useMatch=True*)
Kill everything that matches the given label and all its consequents.

label_from_fragment(*fragment*, *default_type*)
A variant of Label.from_fragment that understands types and wildcards

In particular, it knows that:

1.packages have roles, but checkouts and deployments do not.

2.wildcards expand to their appropriate values

Returns a list of labels. This method does not check that all of the labels returned actually exist as targets in the dependency tree.

labels_for_role(*kind*, *role*, *tag*, *domain=None*)
Find all the target labels with the specified kind, role and tag and return them in a set.

If 'domain' is specified, also require the domain to match.

list_environments_for(*label*)
Return a list of environments that contribute to the environment for the given label.

Returns a list of triples (match level, label, environment), in order.

load_instructions(*label*)
Load the instructions which apply to the given label (usually a wildcard on a role, from a deployment) and return a list of triples (label, filename, instructionfile).

map_unifications(*source_list*)

mark_domain(*domain_name*)
Write a file that marks this directory as a domain so we don't mistake it for the root.

note_unification(*source*, *target*)

package_install_path(*label*)
Where should pkg install itself, by default?

package_obj_path(*label*)
Where should the package with this label build its object files?

packages_for_deployment(*dep_label*)
Return a set of the packages that the given deployment depends upon

This only looks at *direct* dependencies (so if a deployment depends on something that in turn depends on a package that it does not directly depend on, then that indirect package will not be returned).

It does, however, expand wildcards.

packages_using_checkout(*co_label*)
Return a set of the packages which directly use a checkout (this does not include dependencies)

print_banned_roles()

resource_body(*file_name*)
Return the body of a resource as a string.

resource_file_name(*file_name*)

role_combination_acceptable_for_lib(*r1*, *r2*, *domain1=None*, *domain2=None*)
True only if (r1,r2) does not appear in the list of banned roles.

role_install_path(*role*, *domain=None*)
Where should this role find its install to deploy?

roles_do_not_share_libraries(*r1*, *r2*, *domain1=None*, *domain2=None*)
Assert that roles a and b do not share libraries

Either a or b may be * to mean wildcard

Add (r1,r2) to the list of role pairs that do not share their libraries.

set_default_variables(*label*, *store*)
Muddle defines a variety of environment variables which are available whilst a label is being built. The particular variables provided depend on the type of label being built, or the type of build.

Package labels are associated with muddle Makefiles, so any environment variable specific to a package label will be available within a muddle Makefile (i.e., commands such as "muddle build" work on package labels).

Unexpected section title.

```
All labels
----------
```

MUDDLE The muddle executable itself. This can be used in muddle Makefiles, for instance:

```
fred_objdir = $(shell $(MUDDLE) query objdir package:fred{base})
```

MUDDLE_ROOT The absolute path to the root of the build tree (where the '.muddle' and 'src' directories are).

MUDDLE_LABEL The label currently being built.

MUDDLE_KIND, MUDDLE_NAME, MUDDLE_ROLE, MUDDLE_TAG, MUDDLE_DOMAIN Broken-down bits of the label being built. Values will not exist if the label does not contain them (so if 'label' is a checkout label, `MUDDLE_ROLE` will not be set).

MUDDLE_OBJ Where we should build object files for this label - the `obj` directory for packages, the `src` directory for checkouts, and the `deploy` directory for deployments. See "muddle query objdir".

Unexpected section title.

```
Package labels
--------------
```

For package labels, we also set:

MUDDLE_OBJ_OBJ, MUDDLE_OBJ_INCLUDE, MUDDLE_OBJ_LIB, MUDDLE_OBJ_BIN
`$(MUDDLE_OBJ)/obj`, `$(MUDDLE_OBJ)/include`, `$(MUDDLE_OBJ)/lib`, `$(MUDDLE_OBJ)/bin`, respectively. Note that we do not *create* these directories - it is up to the muddle Makefile to do so.

MUDDLE_INSTALL Where we should install package files to.

MUDDLE_INSTRUCT A shortcut to the 'muddle instruct' command for this package. Essentially "$(MUDDLE) instruct $(MUDDLE_LABEL)"

MUDDLE_UNINSTRUCT A shortcut to the 'muddle uninstruct' command for this package. Essentially "$(MUDDLE) uninstruct $(MUDDLE_LABEL)"

MUDDLE_PKGCONFIG_DIRS A path suitable for passing to pkg-config to tell it to look only at packages this label is declared to be dependent on. It will be empty if the label doesn't have any dependencies.

MUDDLE_PKGCONFIG_DIRS_AS_PATH The same as `MUDDLE_PKGCONFIG_DIRS`, for historical reasons.

MUDDLE_INCLUDE_DIRS A space separated list of include directories, constructed from the packages that this label depends on. Names will have been intelligently escaped for the shell. Only directories that actually exist will be included.

Typically used in a muddle Makefile as:

```
CFLAGS += $(MUDDLE_INCLUDE_DIRS:%=-I%)
```

MUDDLE_LIB_DIRS A space separated list of library directories, constructed from the packages that this label depends on. Names will have been intelligently escaped for the shell. Only directories that actually exist will be included.

Typically used in a muddle Makefile as:

```
LDFLAGS += $(MUDDLE_LIB_DIRS:%=-L%)
```

MUDDLE_LD_LIBRARY_PATH The same values as in `MUDDLE_LIB_DIRS`, but with items separated by colons. This is useful for passing (as LD_LIBRARY_PATH) to configure scripts that try to look for libraries when linking test programs.

MUDDLE_KERNEL_DIR If any of the packages that this label depends on has a directory called `kerneldir` in its `obj` dir (so, in its own terms, $(MUDDLE_OBJ)/kerneldir), then we set this value to that directory. Otherwise it is not set. If there is more than one candidate, then the last found is used (but the order of search is not defined, so this would be confusing).

If the build tree is building a Linux kernel, it can be useful to build the kernel into a directory of this name.

MUDDLE_KERNEL_SOURCE_DIR Like `MUDDLE_KERNEL_DIR`, but it looks for a directory called `kernelsource`. The same comments apply.

Unexpected section title.

```
Deployment labels
-----------------
```

For deployment labels we also set:

MUDDLE_DEPLOY_FROM Where we should deploy from (probably just `MUDDLE_INSTALL` with the last component removed)

MUDDLE_DEPLOY_TO Where we should deploy to, if we're a deployment.

Unexpected section title.

```
Release build values
--------------------
```

When building in a release tree (typically by use of "muddle release") extra environment variables are set to allow the build to know useful information about the release. In a non-release build, these will all be set to "(unset)".

MUDDLE_RELEASE_NAME The release name.

MUDDLE_RELEASE_VERSION The release version.

MUDDLE_RELEASE_HASH The hash of the release stamp file. This acts as a useful unique identifier for a particular release, as it is calculated from the stamp file information describing all the release checkouts.

Two releases with the same name and version, but with different checkout information, will have different release hashes.

set_distribution(*name*, *target_dir*)
Set the current distribution name and target directory.

set_domain_parameter(*domain*, *name*, *value*)

set_parameter(*name*, *value*)

Set a domain parameter: Danger Will Robinson! This is a very odd thing to do - domain parameters are typically set by their enclosing domains. Setting your own is an odd idea and liable to get you into trouble. It is, however, the only way of communicating values back from a domain to its parent (and you shouldn't really be doing that either!)

setup_environment(*label*, *src_env*)
Modify src_env to reflect the environments which apply to label, in match order.

target_label_exists(*label*)
Return True if this label is a target.

If it is not, then we are not going to be able to build it.

Note that this method does not understand wildcards, so the match must be exact.

unify_environments(*source*, *target*)
Given a source label and a target label, find all the environments which might apply to source and make them also apply to target.

This is (slightly) easier than one might imagine ..

unify_labels(*source*, *target*)
Unify the 'source' label with/into the 'target' label.

Given a dependency tree containing rules to build both 'source' and 'target', this edits the tree such that the any occurrences of 'source' are replaced by 'target', and dependencies are merged as appropriate.

Free variables (i.e. wildcards in the labels) are untouched - if you need to understand that, see depend.py for quite how this works.

Why is it called "unify" rather than "replace"? Mainly because it does more than replacement, as it has to merge the rules/dependencies together. In retrospect, though, some variation on "merge" might have been easier to remember (if also still inaccurate).

uninstruct_all()

Note: The Builder class actually lives in `muddled.mechanics`, but is available directly as `muddled.Builder` for convenience.

16.3 muddled.depend

Sometimes it is useful to work directly with dependencies. For instance:

```
rule = muddled.depend.depend_one(None,
         muddled.depend.label_from_string('package:rootfs/installed'),
         muddled.depend.label_from_string('package:busybox/installed'))
builder.ruleset.add(rule)
```

Dependency sets and dependency management

class muddled.depend.**Action**
Bases: object

Represents an object you can call to "build" a tag.

build_label(*builder*, *label*)
Build the given label. Your dependencies have been satisfied.

- in_deps - Is the set whose dependencies have been satisified.

Returns True on success, False or throw otherwise.

class muddled.depend.**Label**(*type*, *name*, *role=None*, *tag='*'*, *transient=False*, *system=False*, *domain=None*)
Bases: object

A label denotes an entity in muddle's dependency hierarchy.

A label is structured as:

```
<type>:<name>{<role>}/<tag>[<flags>]
```

or:

```
<type>:(<domain>)<name>{<role>}/<tag>[<flags>]
```

The <type>, <name>, <role> and <tag> parts are composed of the characters [A-Za-z0-9-+_], or the wildcard character '*'.

The <domain> name is composed of the same plus '(' and ')'.

The domain, role and flags are all optional.

Note: The label strings "type:name/tag" and "type:name{}/tag[]" are identical, although the former is the more usual form.)

The '+' is allowed in label parts to allow for names like "g++".

Domains are used when relating sub-builds to each other, and are not necessary when relating labels within the same build. It is not allowed to specify the empty domain as "()" - just omit the parentheses.

If necessary "sub-domains" are specified using nested domains – for instance:

```
(outer)
(outer(inner))
(outer(inner(even.innerer)))
```

This is intended to be unambiguous rather than pretty.

Note that wildcarding of a domain name currently only supports one level (i.e., the top "(*)"), and not wildcarding of nested domains.

If you do find yourself using multi-level domains, we would strongly suggest reconsidering your overall build design.

The "core" part of a label is the `<name>{<role>}` or `(<domain>)<name>{<role>}`. The <type> and <tag> can (typically) be thought of as tracking the progress of the "core" entity through its lifecycle (build sequence, etc.).

Names beginning with an underscore are reserved by muddle, so do not use them for other purposes.

(Why is the 'domain' argument at the end of the argument list? Because it was added after the other arguments were already well-established, and some uses of Label use positional arguments.)

Label instances are treated as immutable by the muddle system, although the implementation does not currently enforce this. Please don't try to abuse this, as Bad Things will happen.

Note: The *flags* on a label are not immutable, and are regarded as transient annotations.

Note: When a domain is included as a subdomain, all of its labels are "adjusted" to have the new, appropriate domain name. This is clearly a special meaning of the word "immutable". However, it should only be the muddle system itself doing this.

Because of this (potential change in content of a label), the domain name does not contribute to a label's hash value. Thus a label that whose domain name is changed will continue to work as the same key in a dictionary (for instance).

Type What kind of label this is. The standard muddle values are "checkout", "package" and "deployment". These values are defined programmatically via muddled.utils.LabelType. Thus the 'type' is conventionally used to indicate what general "stage" of the build process a label belongs to.

Name The name of this checkout/package/whatever. This should be a useful mnemonic for the labels purpose.

Role The role for this checkout/package/whatever. A role might delimit the target architecture of the labels it is used in (roles such as "x86", "arm", "beagleboard"), or the sort of purpose ("role" in the more traditionale sense, such as "boot", "firmware", "packages"), or some other useful delineation of a partition in the general label namespace (thinking of labels as points in an N-dimensional space).

Tag A tag indicating more precisely what stage the label belongs to within each 'type'. There are different conventional values according to the 'type' of the label (for instance, "checked_out", "built", "installed", etc.). These values are defined programmatically via muddled.utils.LabelTag.

Transient If true, changes to this tag will not be persistent in the muddle database. 'transient' is used to denote something which will go away when muddle is terminated - e.g. environment variables.

System If true, marks this label as a system label and not to be reported (by 'muddle depend') unless asked for. System labels are labels "invented" by muddle itself to satisfy implicit dependencies, or to allow the build system as a whole to work.

Domain The domain is used to specify which build or sub-build this label corresponds to. Nested "tail recursive" parenthesised components may be used to specify sub-domains (but this is not recommended). The domain defaults to the current build.

The role may be None, indicating (for instance) that roles are not relevant to this particular label.

The domain may be None, indicating that the label belongs to the current build. If the domain is given as '' (empty string) then this is equivalent to None, and is stored as such. Do not specify domains unless you need to.

The kind, name, role and tag may be wildcarded, by being set to '*'. When evaluating dependencies between labels, for instance, a wildcard indicates "for any value of this part of the label".

Domains can be wildcarded, and that probably means the obvious (that the label applies across all domains), but this may not yet be implemented. Wildcarding of sub-domains may never be supported.

Note that label flags (including specifically 'transient' and 'system') are not equality-preserving properties of a label - two labels are not made unequal just because they have different flags.

(In fact, no two labels should ever have different values for transience, for obvious reasons, and the system flag is intended only to limit over-reporting of information.)

For instance:

```
>>> Label('package', 'busybox')
Label('package', 'busybox', role=None, tag='*')
>>> str(_)
'package:busybox/*'
>>> Label('package', 'busybox', tag='installed')
Label('package', 'busybox', role=None, tag='installed')
>>> str(_)
'package:busybox/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed')
Label('package', 'busybox', role='rootfs', tag='installed')
>>> str(_)
'package:busybox{rootfs}/installed'
>>> Label('package', 'busybox', 'rootfs', 'installed')
Label('package', 'busybox', role='rootfs', tag='installed')
>>> str(_)
'package:busybox{rootfs}/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed', domain="arm.helloworld")
Label('package', 'busybox', role='rootfs', tag='installed', domain='arm.helloworld')
>>> str(_)
'package:(arm.helloworld)busybox{rootfs}/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed', domain="arm(helloworld)")
Label('package', 'busybox', role='rootfs', tag='installed', domain='arm(helloworld)')
>>> str(_)
'package:(arm(helloworld))busybox{rootfs}/installed'
```

FLAG_DOMAIN_SWEEP = 'D'

FLAG_SYSTEM = 'S'

FLAG_TRANSIENT = 'T'

copy()
Return a copy of this label.

copy_and_unify_with(*target*)
Return a copy of ourserlves, unified with the target.

All the non-wildcard parts of 'target' are copied, to overwrite the equivalent parts of the new label.

copy_with_domain(*new_domain*)
Return a copy of self, with the domain changed to new_domain.

Note that if 'new_domain' is given as "" (empty string) then that is treated as if it were given as None.

copy_with_role(*new_role*)
Return a copy of self, with the role changed to new_role.

copy_with_tag(*new_tag*, *system=None*, *transient=None*)
Return a copy of self, with the tag changed to new_tag.

domain

domain_part = '[()A-Za-z0-9._+-]+|*'

domain_part_re = <_sre.SRE_Pattern object>

fragment_re = <_sre.SRE_Pattern object at 0x20eddd0>

static **from_fragment**(*fragment*, *default_type*, *default_role=None*, *default_domain=None*)
Given a string containing a label fragment, return a Label.

The caller indicates the default type, role and domain.

The fragment must contain a <name>, but otherwise *may* contain any of:

- <type>: - if this is not given, the default is used
- (<domain>) - if this is not given, the default is used.
- {<role>} - if this is not given, the default is used.
- /<tag> - if this is not given, a tag appropriate to the <type> is chosen (checked_out, postinstalled or deployed)

Any of the default_xx values may be None.

static **from_string**(*label_string*)
Construct a Label from its string representation.

The string should be of the correct form:

- <type>:<name>/<tag>
- <type>:<name>{<role>}/<tag>
- <type>:<name>/<tag>[<flags>]
- <type>:<name>{<role>}/<tag>[<flags>]
- <type>:(<domain>)<name>/<tag>
- <type>:(<domain>)<name>{<role>}/<tag>
- <type>:(<domain>)<name>/<tag>[<flags>]
- <type>:(<domain>)<name>{<role>}/<tag>[<flags>]

See the docstring for Label itself for the meaning of the various parts of a label.

<flags> is a set of individual characters indicated as flags. There are two flags that will be recognised and used, 'T' for Transience and 'S' for System. Any other flag characters will be ignored.

If the label string is valid, a corresponding Label will be returned, otherwise a utils.GiveUp exception will be raised.

```
>>> Label.from_string('package:busybox/installed')
Label('package', 'busybox', role=None, tag='installed')
>>> Label.from_string('package:busybox{firmware}/installed[ABT]')
Label('package', 'busybox', role='firmware', tag='installed', transient=True)
>>> Label.from_string('package:(arm.hello)busybox{firmware}/installed[ABT]')
Label('package', 'busybox', role='firmware', tag='installed', transient=True, domain='arm.he
>>> Label.from_string('*:(*)*{*}/*')
```

```
        Label('*', '*', role='*', tag='*', domain='*')
        >>> Label.from_string('*:*{*}/*')
        Label('*', '*', role='*', tag='*')
        >>> Label.from_string('foo:bar{baz}/wombat[T]')
        Label('foo', 'bar', role='baz', tag='wombat', transient=True)
        >>> Label.from_string('foo:(ick)bar{baz}/wombat[T]')
        Label('foo', 'bar', role='baz', tag='wombat', transient=True, domain='ick')
        >>> Label.from_string('foo:(ick(ack))bar{baz}/wombat[T]')
        Label('foo', 'bar', role='baz', tag='wombat', transient=True, domain='ick(ack)')
```

A tag must be supplied:

```
        >>> Label.from_string('package:busybox')
        Traceback (most recent call last):
        ...
        GiveUp: Label string 'package:busybox' is not a valid Label
```

If you specify a domain, it may not be "empty":

```
        >>> Label.from_string('package:()busybox/*')
        Traceback (most recent call last):
        ...
        GiveUp: Label string 'package:()busybox/*' is not a valid Label
```

is_definite()
Return True iff this label contains no wildcards

is_wildcard()
Return True iff this label contains at least one wildcard.

This is the dual of is_definite(), but is provided so whichever seems more appropriate to the task at hand can be chosen.

just_match(*other*)
Return True if the labels match, False if they do not

label_part = '[A-Za-z0-9._+-]+|*'

label_part_re = <_sre.SRE_Pattern object>

label_string_re = <_sre.SRE_Pattern object at 0x20ec900>

match(*other*)
Return an integer indicating the match specicifity - which we do by counting '*' s and subtracting from 0.

Returns the match specicifity, None if there wasn't one.

match_without_tag(*other*)
Returns True if other matches self without the tag, False otherwise

Specifically, tests whether the two Labels have identical type, domain, name and role.

middle()
Return the "middle" portion of our name, between type and tag.

That is, the domain, name and role (as appropriate).

For instance:

```
        >>> checkout('fred').middle()
        'fred'
        >>> checkout('jim', domain='a(b)', tag='*').middle()
        '(a(b))jim'
```

```
        >>> package('fred', role='bob', domain='a').middle()
        '(a)fred{bob}'
```

name

role

static split_domain(*value*)

Split a domain into its parts and check that it is valid.

Note that the 'value' should *not* include the outermost parentheses (see the examples below).

Raises a utils.GiveUp exception if it's Bad.

For instance:

```
        >>> Label.split_domain('fred')
        ['fred']
        >>> Label.split_domain('fred(jim)')
        ['fred', 'jim']
        >>> Label.split_domain('fred(jim(bob))')
        ['fred', 'jim', 'bob']
        >>> Label.split_domain('')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain '()' is not allowed
        >>> Label.split_domain('()')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain '(())' starts with zero length domain, '()', i.e. '(('
        >>> Label.split_domain('(')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain part '(()' has unbalanced parentheses, '('
        >>> Label.split_domain(')')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain '())' has unbalanced parentheses, ')'
        >>> Label.split_domain('fred(jim')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain part '(fred(jim)' has unbalanced parentheses, 'fred(jim'
        >>> Label.split_domain('fred((jim(bob)))')
        Traceback (most recent call last):
        ...
        GiveUp: Label domain '(fred((jim(bob))))' starts with zero length domain, '((jim(bob))', i.e
```

split_domains()

Returns a list of the domains for this Label, in order.

If there are no subdomains, then a zero length list is returned.

Raises a utils.GiveUp exception if the parentheses do not match up (the check is only fairly crude), or if there are two adjacent opening parentheses.

This is similar to utils.split_domain, but behaves somewhat differently.

tag

type

unifies(*other*)
Returns True if and only if every field in self is either equal to a field in other , or if other is a wildcard. Wildcards in self do not match anything but a wildcard in other.

class muddled.depend.**Rule**(*target_dep*, *action*)
Bases: object

A rule or "dependency set".

Every Rule has:

- a target Label (its desired result),
- an optional Action object (to do the work to produce that result),
- and a set of Labels on which the target depends (which must have been satisfied before this Rule can be triggered).

In other words, once all the dependency Labels are satisfied, the object can be called to 'build' the target Label.

(And if there is no object, the target is automatically satisfied.)

Note that the "set of Labels" is indeed a set, so adding the same Label more than once will not have any effect (caveat: adding a label with different flags from a previous label may have an effect, but it's not something that should be relied on).

Note: The actual "satisfying" of labels is done in muddled.mechanics. For instance, Builder.build_label() "builds" a label in the context of the rest of its environment, and uses 'action' to "build" the label.

- *target_dep* is the Label this Rule intends to "make".
- *action* is None or an Action, which will be used to "make" the *target_dep*.

add(*label*)
Add a dependency on the given Label.

catenate_and_merge(*other_rule*, *complainOnDuplicate=False*, *replaceOnDuplicate=True*)
Merge ourselves with the given rule.

If replaceOnDuplicate is true, other_rule get priority - this is the target for a unify() and makes the source build instructions go away.

depend_checkout(*co_name*, *tag*)
Add a dependency on label "checkout:<co_name>/<tag>".

depend_deploy(*dep_name*, *tag*)
Add a dependency on label "deployment:<dep_name>/tag".

depend_pkg(*pkg*, *role*, *tag*)
Add a dependency on label "package:<pkg>{<role>}/tag".

merge(*deps*)
Merge another Rule with this one.

Adds all the dependency labels from *deps* to this Rule.

If *deps.action* is not None, replaces our *action* with the one from *deps*.

replace_target(*new_t*)

to_string(*showSystem=True*, *showUser=True*)
Return a string representing this dependency set.

If *showSystem* is true, include dependency labels with the System tag (i.e., dependencies inserted by the muddle system itself), otherwise ignore such.

If *showUser* is true, include dependency labels without the System tag (i.e., "normal" dependencies, explicitly added by the user), otherwise ignore such.

The default is to show all of the dependencies.

For instance (not a very realistic example):

```
>>> tgt = Label.from_string('package:fred{jim}/*')
>>> r = Rule(tgt,None)
>>> r.to_string()
'package:fred{jim}/* <- [ ]'
>>> r.add(Label.from_string('package:bob{bob}/built'))
>>> r.depend_checkout('fred','jim')
>>> r.depend_pkg('albert','jim','built')
>>> r.depend_deploy('hall','deployed')
>>> r.to_string()
'package:fred{jim}/* <- [ checkout:fred/jim, deployment:hall/deployed, package:albert{jim}/b
```

The "<-" is to be read "depends on".

Note that the order of the dependencies in the output is sorted by label.

unify_dependencies (*source*, *target*)
Whenever source appears in our dependencies, replace it with source.unify(target)

class muddled.depend.**RuleSet**
Bases: object

A collection of rules that encapsulate how you can get from A to B.

Formally, this is just a mapping of labels to Rules. Duplicate targets are merged - it's assumed that the objects will be the same.

Informally, we cache some of the label look-ups for a major improvement in build time; the half billion lookups were taking over a hundred seconds to do!

CAVEAT: Be aware that new rules (when added) can be merged into existing rules. Since we don't *copy* rules when we add them, this could be a cause of unexpected side effects...

add (*rule*)
Add the Rule 'rule'.

Specifically, if we already have a rule for this rule's target label, merge the new rule into the old (see Rule.merge).

If this rule is for a new target, just remember it.

merge (*other_deps*)
Merge another RuleSet into this one.

Simply adds each rule from the other RuleSet to this one (see the 'RuleSet.add' method)

rule_for_target (*target*, *createIfNotPresent=False*)
Return the rule for this target - this contains all the labels that need to be asserted in order to build the target.

If createIfNotPresent is true, and there is no rule for this target, then we will create (and add to our internal map) an empty Rule for this target.

Otherwise, if there is no rule for this target, we return None

rules_for_target (*label*, *useTags=True*, *useMatch=True*)
Return the set of rules for any target(s) matching the given label.

- If useTags is true, then we should take account of tags when matching, otherwise we should ignore them. If useMatch is true, then useTags is ignored.
- If useMatch is true, then we allow wildcards in 'label', otherwise we do not.

Returns the set of Rules found, or an empty set if none were found.

rules_which_depend_on (*label*, *useTags=True*, *useMatch=True*)
Given a label, return a set of the rules which have it as one of their dependencies.

If there are no rules which have this label as one of their dependencies, we return the empty set.

- If useTags is true, then we should take account of tags when matching, otherwise we should ignore them. If useMatch is true, then useTags is ignored.
- If useMatch is true, then we allow wildcards in 'label', otherwise we do not.

targets_match (*target*, *useMatch=True*)
Return the set of targets matching the given 'target' label.

If useMatch is true, allow wildcards in 'target' (in which case more than one result may be obtained). If useMatch is false, then at most one match can be found ('target' itself).

Returns a set of suitable targets, or an empty set if there are none.

to_string (*matchLabel=None*, *showUser=True*, *showSystem=True*, *ignore_empty=False*)
Return a string representing this rule set.

If *showSystem* is true, include dependency labels with the System tag (i.e., dependencies inserted by the muddle system itself), otherwise ignore such.

If *showUser* is true, include dependency labels without the System tag (i.e., "normal" dependencies, explicitly added by the user), otherwise ignore such.

The default is to show all of the dependencies.

For instance (not a very realistic example):

```
>>> l = Label.from_string('package:fred{bob}/initial')
>>> r = RuleSet()
>>> depend_chain(None, l, ['built', 'bamboozled'], r)
>>> print str(r)
-----
package:fred{bob}/bamboozled <- [ package:fred{bob}/built ]
package:fred{bob}/built <- [ package:fred{bob}/initial ]
package:fred{bob}/initial <- [ ]
-----
```

The "<-" is to be read "depends on".

Note that the order of the rules in the output is sorted by target label, and is thus reproducible.

unify (*source*, *target*)
Merge source into target.

This is a pain, and depends heavily on CatenatedObject

wrap_actions (*generator*, *label*)

class muddled.depend.**SequentialAction** (*a*, *b*)
Bases: object

Invoke two actions in turn

build_label(*builder*, *label*)

muddled.depend.**checkout**(*name*, *tag='*'*, *domain=None*)
A simple convenience function to return a checkout label

- 'name' is the checkout name
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
>>> l = checkout('fred')
>>> l == Label.from_string('checkout:fred/*')
True
```

muddled.depend.**depend_chain**(*action*, *label*, *tags*, *ruleset*)
Add a chain of dependencies to the given ruleset.

This is perhaps best explained with an example:

```
>>> l = Label.from_string('package:fred{bob}/initial')
>>> r = RuleSet()
>>> depend_chain(None, l, ['built', 'bamboozled'], r)
>>> print str(r)
-----
package:fred{bob}/bamboozled <- [ package:fred{bob}/built ]
package:fred{bob}/built <- [ package:fred{bob}/initial ]
package:fred{bob}/initial <- [ ]
-----
```

muddled.depend.**depend_empty**(*action*, *label*)
Create a dependency set with no prerequisites - simply signals that a tag is available to be built at any time.

muddled.depend.**depend_none**(*action*, *label*)
Quick rule that makes label depend on nothing.

muddled.depend.**depend_one**(*action*, *label*, *dep_label*)
Quick rule that makes label depend only on dep_label.

muddled.depend.**depend_self**(*action*, *label*, *old_tag*)
Make a quick dependency set that depends just on you. Used by some of the standard package and checkout classes to quickly build standard dependency sets.

muddled.depend.**deployment**(*name*, *tag='*'*, *domain=None*)
A simple convenience function to return a deployment label

- 'name' is the deployment name
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
>>> l = deployment('fred')
>>> l == Label.from_string('deployment:fred/*')
True
```

muddled.depend.**label_from_string**(*str*)
Do not use this!!! Can you say "deprecated"?

This function was originally removed, since it is replaced by Label.from_string. However, too many old builds still attempt to import it, which can cause problems at the "muddle init" stage, and also with "muddle unstamp".

Please do not use this function in new builds.

muddled.depend.**label_list_to_string**(*labels*, *join_with=' '*)

muddled.depend.**label_set_to_string**(*label_set*, *start_with='['*, *end_with=']'*, *join_with=', '*)
Utility function to convert a label set to a string.

muddled.depend.**needed_to_build**(*ruleset*, *target*, *useTags=True*, *useMatch=False*)
Given a rule set and a target, return a complete list of the rules needed to build the target.

- If useTags is true, then we should take account of tags when looking for the rules for this 'target', otherwise we should ignore them.
- If useMatch is true, then we allow wildcards in 'target', otherwise we do not.

Returns a list of rules.

muddled.depend.**normalise_checkout_label**(*label*, *tag='*'*)
Given a checkout label with random "other" fields, normalise it.

Returns a normalised checkout label, with the role unset and the tag set to 'tag' (normally, "*"). This may be the same label (if it was already normalised), or it may be a new label. No guarantee is given of either.

Raise a MuddleBug exception if the label is not a checkout label.

A normalised checkout label:

1. Has the given tag (normally '*', sometimes LabelTag.CheckedOut)
2. Does not have a role (checkout labels do not use the role)
3. Does not have the system or transient flags set
4. Has the same name and (if present) domain

muddled.depend.**package**(*name*, *role*, *tag='*'*, *domain=None*)
A simple convenience function to return a package label

- 'name' is the package name
- 'role' is the package role
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
>>> l = package('fred', 'jim')
>>> l == Label.from_string('package:fred{jim}/*')
True
```

muddled.depend.**required_by**(*ruleset*, *label*, *useTags=True*, *useMatch=True*)
Given a ruleset and a label, form the list of labels that (directly or indirectly) depend on label. We deliberately do not give you the associated rules since you will want to call needed_to_build() individually to ensure that other prerequisites are satisfied.

The order in which we give you the labels gives you a hint as to a logical order to rebuild in (i.e. one the user will vaguely understand).

- useMatch - If True, do wildcard matches, else do an exact comparison.
- **useTags - If False, we discount the value of a tag - this effectively** results in a wildcard tag search.

Returns a set of labels to build.

muddled.depend.**retag_label_list**(*labels*, *new_tag*)
Does what it says on the tin, returning the new label list.

That is, returns a list formed by copying each Label in 'labels' and setting its tag to the given 'new_tag'.

muddled.depend.**rule_list_to_string**(*rule_list*)
Utility function to convert a rule list to a string.

muddled.depend.**rule_target_str**(*rule*)
Take a rule and return its target as a string. Mainly used as an argument for map so we can print lists of rules sensibly.

muddled.depend.**rule_with_least_dependencies**(*rules*)
Given a (Python) set of rules, find the 'best' one to use.

This is actually impossible by any rational metric, so you usually only expect to call this function with a set of size 1, in which case our metric really doesn't matter.

However, in a vague attempt to be somewhat intelligent, we return the element with the fewest direct dependencies.

16.4 muddled.utils

Muddle utilities.

class muddled.utils.**Choice**(*choices*)
Bases: object

A choice "sequence".

A choice sequence is:

- a string or dictionary, the only choice. For instance:

```
choice = Choice("libxml-dev2")
assert choice.choose('any string at all') == 'libxml-dev2)
```

- a sequence of the form [(pattern, value), ...]; that is a sequence of one or more '(pattern, value)' pairs, where each 'pattern' is an fnmatch pattern (see below) and each 'value' is a string or dict.

  The patterns are compared to 'what_to_match' in turn, and if one matches, the corresponding 'value' is returned. If none match, a ValueError is raised.

  For instance:

```
choice = Choice([ ('ubuntu-12.*', 'package-v12'),
                  ('ubuntu-1?.*', 'package-v10') ])
try:
    match = choice.choose_to_match_os()
except ValueError:
    print 'No package matched OS %s'%get_os_version_name()
```

- a sequence of the form [(pattern, value), ..., default]; that is a sequence of one or more pairs (as above), with a final "default" value, which must be a string or dict or None.

  The patterns are compared to 'what_to_match' in turn, and if one matches, the corresponding 'value' is returned. If none match, the final default value is returned. None is allowed so the caller can easily tell that no choice was actually made.

  choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), 'package-v09'])

Definition list ends without a blank line; unexpected unindent.

'match' will always have a "sensible" value match = choice.choose_to_match_os()

choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), None])

Definition list ends without a blank line; unexpected unindent.

match = choice.choose_to_match_os() if match is None:

Unexpected indentation.

return # We know there was no given value

- as a result of the previous, we also allow [default], although [None] is of questionable utility.

  choice = Choice(["libxml-dev2"]) assert choice.choose('any string at all') == 'libxml-dev2)

  choice = Choice(["None"]) assert choice.choose('any string at all') is None

  (although that latter is the only way of "forcing" a Choice that will always return None, if you did need such a thing...)

Why not just use a list of pairs (possibly with a default string at the end, essentially just what we pass to Choice)? Well, it turns out that if you want to do something like:

```
pkgs.apt_get(["fromble1",
              Choice([ ('ubuntu-12.*', 'fromble'),
                       ('ubuntu-11.*', 'alex'),
                       None ]),
              "ribbit",
              Choice([ ('ubuntu-12.*', 'package-v12'),
                       ('ubuntu-1?.*', 'package-v10'),
                       'package-v7' ]),
              "libxml-dev2",
             ])
```

it is (a) really hard to type it right if it is just nested sequences, and (b) terribly hard to give useful error messages when the user doesn't get it right. There are already enough brackets of various sorts, and if we don't have the "Choice" delimiters, it just gets harder to keep track.

choose(*what_to_match*)

Try to match 'what_to_match', and return the appropriate value.

Raises ValueError if there is no match.

Returns None if (and only if) that was given as a fall-back default value.

choose_to_match_os(*version_name=None*)

A special case of 'decide' to match OS id/version

If 'version_name' is None, then it looks up the system 'id' (the "name" of the OS, e.g., "ubuntu"), and 'version' of the OS (e.g., "12.10") from /etc/os-release, and concatenates them separated by a space (so "ubuntu 12.10").

It returns the result of calling:

choose(version_name)

So, on an Ubuntu system (which also includes a Linux Mint system, since its /etc/os-release identifies it as the underlying Ubuntu system), one might do:

choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), 'package-v7'])

Definition list ends without a blank line; unexpected unindent.

choice.choose_to_match_os()

to choose the appropriate version of "package" depending on the OS.

muddled.utils.**Error**
alias of *MuddleBug*

muddled.utils.**Failure**
alias of *GiveUp*

exception muddled.utils.**GiveUp**(*message=None*, *retcode=1*)
Bases: exceptions.Exception

Use this to indicate that something has gone wrong and we are giving up.

This is not an error in muddle itself, however, so there is no need for a traceback.

By default, a return code of 1 is indicated by the 'retcode' value - this can be set by the caller to another value, which __main__.py should then use as its return code if the exception reaches it.

retcode = 1

class muddled.utils.**HashFile**(*name*, *mode='r'*, *ignore_comments=False*, *ignore_blank_lines=False*)
Bases: object

A very simple class for handling files and calculating their SHA1 hash.

We support a subset of the normal file class, but as lines are read or written, we calculate a SHA1 hash for the file.

Optionally, comment lines and/or blank lines can be ignored in calculating the hash, where comment lines are those starting with a '#', or whitespace and a '#', and blank lines are those which contain only whitespace (which includes empty lines).

Open the file, for read or write.

- 'name' is the name (path) of the file to open
- 'mode' is either 'r' (for read) or 'w' (for write). If 'w' is specified, then if the file doesn't exist, it will be created, otherwise it will be truncated.
- if 'ignore_comments' is true, then lines starting with a '#' (or whitespace and a '#') will not be used in calculating the hash.
- if 'ignore_blank_lines' is true, then lines that are empty (zero length), or contain only whitespace, will not be used in calculating the hash.

Note that this "ignore" doesn't mean "don't write to the file", it just means "ignore when calculating the hash".

close()
Close the file.

hash()
Return the SHA1 hash, calculated from the lines so far, as a hex string.

next()

readline()
Read the next line from the file, and add it to the SHA1 hash as well.

Returns '' if there is no next line (i.e., EOF is reached).

write(*text*)
Write the give text to the file, and add it to the SHA1 hash as well.

(Unless we are ignoring comment lines and it is a comment line, or we are ignoring blank lines and it is a blank line, in which case it will be written to the file but not added to the hash.)

As is normal for file writes, the 'n' at the end of a line must be specified.

exception muddled.utils.**MuddleBug**(*message=None*, *retcode=1*)
Bases: *muddled.utils.GiveUp*

Use this to indicate that something has gone wrong with muddle itself.

We thus expect that a traceback will be produced.

class muddled.utils.**MuddleOrderedDict**
Bases: _abcoll.MutableMapping

A simple dictionary-like class that returns keys in order of (first) insertion.

class muddled.utils.**MuddleSortedDict**
Bases: _abcoll.MutableMapping

A simple dictionary-like class that returns keys in sorted order.

exception muddled.utils.**ShellError**(*cmd*, *retcode*, *output=None*)
Bases: *muddled.utils.GiveUp*

exception muddled.utils.**Unsupported**(*message=None*, *retcode=1*)
Bases: *muddled.utils.GiveUp*

Use this to indicate that an action is unsupported.

This is used, for instance, when git reports that it will not pull to a shallow clone, which is not an error, but the user will want to know.

This is deliberately a subclass of GiveUp, because it *is* telling muddle to give up an operation.

class muddled.utils.**VersionNumber**(*major=0*, *minor=0*)
Bases: object

Simple support for two part "semantic version" numbers.

Such version numbers are of the form <major>.<minor>

static from_string(*s*)

next()
Return the next (minor) version number.

static unset()
Return an unset version number.

Unset version numbers compare less than proper ones.

muddled.utils.**arch_name**()
Retrieve the name of the architecture on which we're running. Some builds require packages to be built on a particular (odd) architecture.

muddled.utils.**c_escape**(*v*)
Escape sensitive characters in v.

muddled.utils.**calc_file_hash**(*filename*)
Calculate and return the SHA1 hash for the named file.

muddled.utils.**copy_file**(*from_path*, *to_path*, *object_exactly=False*, *preserve=False*, *force=False*)
Copy a file (either a "proper" file, not a directory, or a symbolic link).

Just like recursively_copy, only not recursive :-)

If the target file already exists, it is overwritten.

Caveat: if the target file is a directory, it will not be overwritten. If the source file is a link, being copied as a link, and the target file is not a link, it will not be overwritten.

If 'object_exactly' is true, then if 'from_path' is a symbolic link, it will be copied as a link, otherwise the referenced file will be copied.

If 'preserve' is true, then the file's mode, ownership and timestamp will be copied, if possible. Note that on Un*x file ownership can only be copied if the process is running as 'root' (or within 'sudo').

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

muddled.utils.**copy_file_metadata**(*from_path*, *to_path*)
Copy file metadata.

If 'to_path' is a link, then it tries to copy whatever it can from 'from_path', treated as a link.

If 'to_path' is not a link, then it copies from 'from_path', or, if 'from_path' is a link, whatever 'from_path' references.

Metadata is: mode bits, atime, mtime, flags and (if the process has an effective UID of 0) the ownership (uid and gid).

muddled.utils.**copy_name_list_with_dirs**(*file_list*, *old_root*, *new_root*, *object_exactly=True*, *preserve=False*)
Given file_list, create file_list[new_root/old_root], creating any directories you need on the way.

file_list is a list of full path names. old_root is the old root directory new_root is where we want them copied

muddled.utils.**copy_without**(*src*, *dst*, *without=None*, *object_exactly=True*, *preserve=False*, *force=False*, *verbose=True*)
Copy files from the 'src' directory to the 'dst' directory, without those in 'without'

If given, 'without' should be a sequence of filenames - for instance, ['.bzr', '.svn'].

If 'object_exactly' is true, then symbolic links will be copied as links, otherwise the referenced file will be copied.

If 'preserve' is true, then the file's mode, ownership and timestamp will be copied, if possible. Note that on Un*x file ownership can only be copied if the process is running as 'root' (or within 'sudo').

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

If 'verbose' is true (the default), print out what we're copying.

Creates directories in the destination, if necessary.

Uses copy_file() to copy each file.

muddled.utils.**current_machine_name**()
Return the identity of the current machine - possibly including the domain name, possibly not

muddled.utils.**current_user**()
Return the identity of the current user, as an email address if possible, but otherwise as a UNIX uid

muddled.utils.**debian_version_is**(*test*, *ref*)
Return 1 if test > ref, -1 if ref > test, 0 if they are equal

muddled.utils.**do_shell_quote**(*str*)

muddled.utils.**domain_subpath**(*domain_name*)
Calculate the sub-path for a given domain name.

For instance:

```
    >>> domain_subpath('a')
    'domains/a'
    >>> domain_subpath('a(b)')
    'domains/a/domains/b'
    >>> domain_subpath('a(b(c))')
    'domains/a/domains/b/domains/c'
    >>> domain_subpath('a(b(c)')
    Traceback (most recent call last):
    ...
    GiveUp: Domain name "a(b(c)" has mis-matched parentheses
```

muddled.utils.**dynamic_load**(*filename*)

muddled.utils.**ensure_dir**(*dir*, *verbose=True*)
Ensure that dir exists and is a directory, or throw an error.

muddled.utils.**find_by_predicate**(*source_dir*, *accept_fn*, *links_are_symbolic=True*)

Given a source directory and an acceptance function fn(source_base, file_name) -> result

Obtain a list of [result] if result is not None.

muddled.utils.**find_domain**(*root_dir*, *dir*)
Find the domain of 'dir'.

'root_dir' is the root of the (entire) muddle build tree.

This function basically works backwards through the path of 'dir', until it reaches 'root_dir'. As it goes, it assembles the full domain name for the domain enclosing 'dir'.

Returns the domain name, or None if 'dir' is not within a subdomain, and the directory of the root of the domain. That is:

(domain_name, domain_dir) or (None, None)

muddled.utils.**find_label_dir**(*builder*, *label*)
Given a label, find the corresponding directory.

- for checkout labels, the checkout directory
- for package labels, the install directory
- for deployment labels, the deployment directory

This is the heart of "muddle query dir".

muddled.utils.**find_local_relative_root**(*builder*, *label*)
Given a label, find its "local" root directory, relative to toplevel.

Calls find_local_root() and then calculates the location of that relative to the root of the entire muddle build tree.

muddled.utils.**find_local_root**(*builder*, *label*)
Given a label, find its "local" root directory.

For a normal label, this will be the normal muddle root directory (where the top-level .muddle/ directory is).

For a label in a subdomain, it will be the root directory of that subdomain - again, where its .muddle/ directory is.

muddled.utils.**find_root_and_domain**(*dir*)
Find the build tree root containing 'dir', and find the domain of 'dir'.

This function basically works backwards through the path of 'dir', until it finds a directory containing a '.muddle/' directory, that is not within a subdomain. As it goes, it assembles the full domain name for the domain enclosing 'dir'.

Returns a pair (root_dir, current_domain).

If 'dir' is not within a subdomain, then 'current_domain' will be None.

If 'dir' is not within a muddle build tree, then 'root_dir' will also be None.

muddled.utils.**get_cmd_data**(*thing*, *env=None*, *show_command=False*)
Run the command 'thing', and return its output.

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

Note that the output of the command is not shown whilst the command is running.

If the command returns a non-zero exit code, then we raise a ShellError.

(This is basically a muddle-flavoured wrapper around subprocess.check_output)

muddled.utils.**get_domain_name_from**(*dir*)
Given a directory 'dir', extract the domain name.

'dir' should not end with a trailing slash.

It is assumed that 'dir' is of the form "<something>/domains/<domain_name>", and we want to return <domain_name>.

muddled.utils.**get_os_version_name**()
Retrieve a string identifying this version of the operating system

Looks in /etc/os-release, which gives a different result than platform.py, which looks in /etc/lsb-release.

muddled.utils.**get_prefix_pair**(*prefix_one*, *value_one*, *prefix_two*, *value_two*)
Returns a pair (prefix_onevalue_one, prefix_twovalue_two) - used by rrw.py as a utility function

muddled.utils.**indent**(*text*, *indent*)
Return the text indented with the 'indent' string.

(i.e., place 'indent' in front of each line of text).

muddled.utils.**is_release_build**(*dir*)
Check if the given 'dir' is the top level of a release build.

'dir' should be the path to the directory contining the build's `.muddle` directory (the "top" of the build).

The build is assumed to be a release build if there is a file called `.muddle/Release`.

muddled.utils.**is_subdomain**(*dir*)
Check if the given 'dir' is a (sub)domain.

'dir' should be the path to the directory contining the build's `.muddle` directory (the "top" of the build).

The build is assumed to be a (sub)domain if there is a file called `.muddle/am_subdomain`.

muddled.utils.**iso_time**()
Retrieve the current time and date in ISO style `YYYY-MM-DD HH:MM:SS`.

muddled.utils.**join_domain**(*domain_parts*)
Re-join a domain name we split with split_domain.

muddled.utils.**mark_as_domain**(*dir*, *domain_name*)
Mark the build in 'dir' as a (sub)domain

This is done by creating a file `.muddle/am_subdomain`

'dir' should be the path to the directory contining the sub-build's `.muddle` directory (the "top" of the sub-build).

'dir' should thus be of the form "<somewhere>/domains/<domain_name>", but we do not check this.

The given 'domain_name' is written to the file, but this should not be particularly trusted - refer to the containing directory structure for the canonical domain name.

`muddled.utils.`**`maybe_shell_quote`**(*str*, *doQuote*)
If doQuote is False, do nothing, else shell-quote `str`.

Annoyingly, shell quoting things correctly must use backslashes, since quotes can (and will) be misinterpreted. Bah.

NB: Despite the name, this is actually "escaping", rather then "quoting". Specifically, any single quote, double quote or backslash characters in the original string will be converted to a backslash followed by the original character, in the final string.

`muddled.utils.`**`normalise_dir`**(*dir*)

`muddled.utils.`**`normalise_path`**(*dir*)

`muddled.utils.`**`num_cols`**()
How many columns on our terminal?

If it can't tell (e.g., because it curses is not available), returns 70.

`muddled.utils.`**`pad_to`**(*str*, *val*, *pad_with=' '*)
Pad the given string to the given number of characters with the given string.

`muddled.utils.`**`page_text`**(*progname*, *text*)
Try paging 'text' by piping it through 'progname'.

Looks for 'progname' on the PATH, and if os.environ['PATH'] doesn't exist, tries looking for it on os.defpath.

If an executable version of 'progname' can't be found, just prints the text out.

If 'progname' is None (or an empty string, or otherwise false), then just print 'text'.

`muddled.utils.`**`parse_etc_os_release`**()
Parse /etc/os-release and return a dictionary

This is *not* a good parser by any means - it is the quickest and simplest thing I could do.

Note that a line like:

```
FRED='Fred's name'
```

will give us:

```
key 'Fred' -> value r"Fred's name"
```

i.e., we do not treat backslashes in a string in any way at all. In fact, we don't do anything with strings other than throw away paired outer "" or "".

Oh, also we don't check whether the names before the '=' signs are those that are expected, although we do provide a default value for 'ID' if it is not given (as the documentation for /etc/os-release' specified).

(Note that the standard library platform.py (in 2.7) looks at /etc/lsb-release, instead of /etc/os-release, which gives different results.)

`muddled.utils.`**`parse_gid`**(*builder*, *text_gid*)

Todo

One day, we should do something more intelligent than just assuming your gid is numeric

muddled.utils.**parse_mode**(*in_mode*)
Parse a UNIX mode specification into a pair (clear_bits, set_bits).

muddled.utils.**parse_uid**(*builder*, *text_uid*)

Todo

One day, we should do something more intelligent than just assuming your uid is numeric

muddled.utils.**print_string_set**(*ss*)
Given a string set, return a string representing it.

muddled.utils.**quote_list**(*lst*)
Given a list, quote each element of it and return them, space separated

muddled.utils.**recursively_copy**(*from_dir*, *to_dir*, *object_exactly=False*, *preserve=True*, *force=False*)
Take everything in from_dir and copy it to to_dir, overwriting anything that might already be there.

Dot files are included in the copying.

If object_exactly is true, then symbolic links will be copied as links, otherwise the referenced file will be copied.

If preserve is true, then the file's mode, ownership and timestamp will be copied, if possible. This is only really useful when copying as a privileged user.

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

muddled.utils.**recursively_remove**(*a_dir*)
Recursively demove a directory.

muddled.utils.**rel_join**(*vroot*, *path*)
Find what path would be called if it existed inside vroot. Differs from os.path.join() in that if path contains a leading '/', it is not assumed to override vroot.

If vroot is none, we just return path.

muddled.utils.**replace_root_name**(*base*, *replacement*, *filename*)
Given a filename, a base and a replacement, replace base with replacement at the start of filename.

muddled.utils.**run0**(*thing*, *env=None*, *show_command=True*, *show_output=True*)
Run the command 'thing', returning nothing.

(Run and return 0 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used.

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs. Note that this is the default.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

muddled.utils.**run1**(*thing*, *env=None*, *show_command=True*, *show_output=False*)
Run the command 'thing', returning its output.

(Run and return 1 value)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

Otherwise, the command output (stdout and stderr combined) is returned.

muddled.utils.**run2**(*thing*, *env=None*, *show_command=True*, *show_output=False*)
Run the command 'thing', returning the return code and output.

(Run and return 2 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

The output of the command (stdout and stderr) goes to the normal stdout whilst the command is running.

The command return code and output are returned as a tuple:

(retcode, output)

muddled.utils.**run3**(*thing*, *env=None*, *show_command=True*, *show_output=False*)
Run the command 'thing', returning the return code, stdout and stderr.

(Run and return 3 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

The output of the command is shown whilst the command is running; its stdout goes to the normal stdout, and its stderr to stderr.

The command return code, stdout and stderr are returned as a tuple:

(retcode, stdout, stderr)

muddled.utils.**shell**(*thing*, *env=None*, *show_command=True*)
Run the command 'thing' in the shell.

If 'thing' is a string (e.g., "ls -l"), then it will be used as it is given.

If 'thing' is a sequence (e.g., ["ls", "-l"]), then each component will be escaped with pipes.quote(), and the result concatenated (with spaces between) to give the command line to run.

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

The output of the command will always be printed out as it runs.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

Unlike the various 'runX' functions, this calls subprocess.Popen with 'shell=True'. This makes things like "cd" available in 'thing', and use of shell specific things like value expansion. It also, morei mportantly for muddle, allows commands like "git clone" to do their progress report "rolling" output. However, the warnings in the Python subprocess documentation should be heeded about not using unsafe command lines.

NB: If you *do* want to do "cd xxx; yyy", you're probably better doing:

```
with Directory("xxx"):
    shell("yyy")
```

muddled.utils.**sort_domains**(*domains*)

Given a sequence of domain names, return them sorted by depth.

So, given some random domain names (and we forgot to forbid strange names starting with '+' or '-'):

```
>>> a = ['a', '+1', '-2', 'a(b(c2))', 'a(b(c1))', '+1(+2(+4(+4)))',
...     'b(b)', 'b', 'b(a)', 'a(a)', '+1(+2)', '+1(+2(+4))', '+1(+3)']
```

sorting "alphabetically" gives the wrong result:

```
>>> sorted(a)
['+1', '+1(+2(+4(+4)))', '+1(+2(+4))', '+1(+2)', '+1(+3)', '-2', 'a', 'a(a)', 'a(b(c1))', 'a(b(c
```

so we needed this function:

```
>>> sort_domains(a)
['+1', '+1(+2)', '+1(+2(+4))', '+1(+2(+4(+4)))', '+1(+3)', '-2', 'a', 'a(a)', 'a(b(c1))', 'a(b(c
```

If we're given a domain name that is None, we'll replace it with ''.

muddled.utils.**split_debian_version**(*v*)

Takes a debian-style version string - <major>.<minor>.<subminor>-<issue><additional> - and turns it into a dictionary with those keys.

muddled.utils.**split_domain**(*domain_name*)

Given a domain name, return a tuple of the hierarchy of sub-domains.

For instance:

```
>>> split_domain('a')
['a']
>>> split_domain('a(b)')
['a', 'b']
>>> split_domain('a(b(c))')
['a', 'b', 'c']
>>> split_domain('a(b(c)')
Traceback (most recent call last):
...
GiveUp: Domain name "a(b(c)" has mis-matched parentheses
```

We don't actually allow "sibling" sub-domains, so we try to complain helpfully:

```
>>> split_domain('a(b(c)(d))')
Traceback (most recent call last):
...
GiveUp: Domain name "a(b(c)(d))" has 'sibling' sub-domains
```

If we're given '' or None, we return [''], "normalising" the domain name.

```
>>> split_domain('')
['']
>>> split_domain(None)
['']
```

muddled.utils.**split_path_left**(*in_path*)

Given a path `a/b/c ...`, return a pair `(a, b/c..)` - ie. like `os.path.split()`, but leftward.

What we actually do here is to split the path until we have nothing left, then take the head and rest of the resulting list.

For instance:

```
>>> split_path_left('a/b/c')
('a', 'b/c')
>>> split_path_left('a/b')
('a', 'b')
```

For a single element, behave in sympathy (but, of course, reversed) to `os.path.split`:

```
>>> import os
>>> os.path.split('a')
('', 'a')
>>> split_path_left('a')
('a', '')
```

The empty string isn't really a sensible input, but we cope:

```
>>> split_path_left('')
('', '')
```

And we take some care with delimiters (hopefully the right sort of care):

```
>>> split_path_left('/a///b/c')
('', 'a/b/c')
>>> split_path_left('//a/b/c')
('', 'a/b/c')
>>> split_path_left('///a/b/c')
('', 'a/b/c')
```

muddled.utils.**split_vcs_url**(*url*)

Split a URL into a vcs and a repository URL. If there's no VCS specifier, return (None, None).

muddled.utils.**string_cmp**(*a*, *b*)

Return -1 if a < b, 0 if a == b, +1 if a > b.

muddled.utils.**text_in_node**(*in_xml_node*)

Return all the text in this node.

muddled.utils.**total_ordering**(*cls*)

Class decorator that fills-in missing ordering methods

muddled.utils.**truncate**(*text*, *columns=None*, *less=0*)
Truncate the given text to fit the terminal.

More specifically:

1.Split on newlines

2.If the first line is too long, cut it and add '...' to the end.

3.Return the first line

If 'columns' is 0, then don't do the truncation of the first line.

If 'columns' is None, then try to work out the current terminal width (using "curses"), and otherwise use 80.

If 'less' is specified, then the actual width used will be the calculated or given width, minus 'less' (so if columns=80 and less=2, then the maximum line length would be 78). Clearly this is ignored if 'columns' is 0.

muddled.utils.**unescape_backslashes**(*str*)
Replace every string 'X' with X, as if you were a shell

muddled.utils.**unix_time**()
Return the current UNIX time since the epoch.

muddled.utils.**unquote_list**(*lst*)
Given a list of objects, potentially enclosed in quotation marks or other shell weirdness, return a list of the actual objects.

muddled.utils.**well_formed_dot_muddle_dir**(*dir*)
Return True if this seems to be a well-formed .muddle directory

We're not trying to be absolutely rigorous, but do want to detect (for instance) an erroneous file with that name, or an empty directory

muddled.utils.**wrap**(*text*, *width=None*, ***kwargs*)
A convenience wrapper around textwrap.wrap()

(basically because muddled users will have imported utils already).

muddled.utils.**xml_elem_with_child**(*doc*, *elem_name*, *child_text*)
Return an element 'elem_name' containing the text child_text in doc.

16.5 muddled.checkouts

16.5.1 muddled.checkouts.simple

Simple entry points so that descriptions can assert the existence of checkouts easily

muddled.checkouts.simple.**absolute**(*builder*, *co_name*, *repo_url*, *rev=None*, *branch=None*)
Check out a repository from an absolute URL.

<repo_url> must be of the form <vcs>+<url>, where <vcs> is one of the support version control systems (e.g., 'git', 'svn').

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

The repository <repo_url>/<co_name> will be checked out into src/<co_name>.

muddled.checkouts.simple.**relative**(*builder*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A simple, VCS-controlled, checkout.

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out into src/<co_name>, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

For example:

```
<base_url>/<co_name>  -->  src/<co_name>
```

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> will be checked out instead:

```
<base_url>/<repo_relative>  -->  src/<co_name>
```

16.5.2 muddled.checkouts.twolevel

Two-level checkouts. Makes it slightly easier to separate checkouts out into roles. I've deliberately not implemented arbitrary-level checkouts for fear of complicating the checkout tree.

muddled.checkouts.twolevel.**absolute**(*builder*, *co_dir*, *co_name*, *repo_url*, *rev=None*, *branch=None*)

Check out a twolevel repository from an absolute URL.

<repo_url> must be of the form <vcs>+<url>, where <vcs> is one of the support version control systems (e.g., 'git', 'svn').

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

The repository <repo_url>/<co_name> will be checked out into src/<co_dir>/<co_name>.

muddled.checkouts.twolevel.**relative**(*builder*, *co_dir*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A two-level version of checkout.simple.relative().

It attempts to check out <co_dir>/<co_name> (but see below).

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> ...

In the normal case, the location in the repository and in the checkout is assumed the same (i.e., <co_dir>/<co_name>). So, for instance, with co_dir="A" and co_name="B", the repository would have:

```
<base_url>/A/B
```

which we would check out into:

```
src/A/B
```

Occasionally, though, the repository is organised differently, so for instance, one might want to checkout:

```
<base_url>/B
```

into:

```
src/A/B
```

In this latter case, one can use the 'repo_relative' argument, to say where the checkout is relative to the repository's "base". So, in the example above, we still have co_dir="A" and co_name="B", but we also want to say repo_relative=B.

muddled.checkouts.twolevel.**twolevel**(*builder*, *co_dir*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A two-level version of checkout.simple.relative().

It attempts to check out <co_dir>/<co_name> (but see below).

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> ...

In the normal case, the location in the repository and in the checkout is assumed the same (i.e., <co_dir>/<co_name>). So, for instance, with co_dir="A" and co_name="B", the repository would have:

```
<base_url>/A/B
```

which we would check out into:

```
src/A/B
```

Occasionally, though, the repository is organised differently, so for instance, one might want to checkout:

```
<base_url>/B
```

into:

```
src/A/B
```

In this latter case, one can use the 'repo_relative' argument, to say where the checkout is relative to the repository's "base". So, in the example above, we still have co_dir="A" and co_name="B", but we also want to say repo_relative=B.

16.6 muddled.deployments

File deployment indicates where the final deployment should end up. For instance:

```
muddled.deployments.filedep.deploy(builder, '/', 'omap', roles)
```

16.6.1 muddled.deployments.collect

Collect deployment.

This deployment is used to collect elements from:

- checkout directories
- package 'obj' directories
- package role 'install' directories
- other deployments

into deployment directories, usually to be processed by some external tool.

class muddled.deployments.collect.**AssemblyDescriptor**(*from_label*, *from_rel*, *to_name*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *usingRSync=False*, *obeyInstructions=True*)

Bases: object

Construct an assembly descriptor.

We copy from the directory from_rel in from_label (package, deployment, checkout) to the name to_name under the deployment.

Give a package of '*' to copy from the install directory for a given role.

If recursive is True, we'll copy recursively.

•failOnAbsentSource - If True, we'll fail if the source doesn't exist.

•copyExactly - If True, keeps links. If false, copies the file they point to.

get_source_dir(*builder*)

class muddled.deployments.collect.**CollectApplyChmod**

Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.collect.**CollectApplyChown**

Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.collect.**CollectDeploymentBuilder**

Bases: *muddled.depend.Action*

Builds the specified collect deployment.

add_assembly(*assembly_descriptor*)

apply_instructions(*builder*, *label*, *prepare*, *deploy_path*)

build_label(*builder*, *label*)

Actually do the copies ..

deploy(*builder*, *label*, *target_base*)

sort_out_and_run_instructions(*builder*, *label*)

class muddled.deployments.collect.**InstructionImplementor**

Bases: object

apply(*builder*, *instruction*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instruction*, *role*, *path*)
Prepares for rsync. This means fixing up the destination file (e.g. removing it if it may have changed uid by a previous deploy) so we will be able to rsync it.

muddled.deployments.collect.**copy_from_checkout**(*builder*, *name*, *checkout*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

muddled.deployments.collect.**copy_from_deployment**(*builder*, *name*, *dep_name*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

usingRSync - set to True to copy with rsync - substantially faster than cp

muddled.deployments.collect.**copy_from_package_obj**(*builder*, *name*, *pkg_name*, *pkg_role*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

- **If 'usingRSync' is true, copy with rsync - substantially faster than** cp, if you have rsync. Not very functional if you don't :-)

muddled.deployments.collect.**copy_from_role_install**(*builder*, *name*, *role*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*, *obeyInstructions=True*)

Add a requirement to copy from the given role's install to the named deployment.

'name' is the name of the collecting deployment, as created by:

```
deploy(builder, name)
```

which is remembered as a rule whose target is `deployment:<name>/deployed`, where <name> is the 'name' given.

'role' is the role to copy from. Copying will be based from 'rel' within the role's `install`, to 'dest' within the deployment.

The label `package:(<domain>)*{<role>}/postinstalled` will be added as a dependency of the collecting deployment rule.

An AssemblyDescriptor will be created to copy from 'rel' in the install directory of the label `package:*{<role>}/postinstalled`, to 'dest' within the deployment directory of 'name', and added to the rule's actions.

So, for instance:

```
copy_from_role_install(builder,'fred','data','public','data/public',
                       True, False, True)
```

might copy (recursively) from:

```
install/data/public
```

to:

```
deploy/fred/data/public
```

'rel' may be the empty string ('') to copy all files in the install directory.

- If 'recursive' is true, then copying is recursive, otherwise it is not.
- If 'failOnAbsentSource' is true, then copying will fail if the source does not exist.
- If 'copyExactly' is true, then symbolic links will be copied as such, otherwise the linked file will be copied.
- **If 'usingRSync' is true, copy with rsync - substantially faster than** cp, if you have rsync. Not very functional if you don't :-)
- If 'obeyInstructions' is False, don't obey any applicable instructions.

muddled.deployments.collect.**deploy**(*builder*, *name*)

Create a collection deployment builder.

This adds a new rule linking the label `deployment:<name>/deployed` to the collection deployment builder.

You can then add assembly descriptors using the other utility functions in this module.

Dependencies get registered when you add an assembly descriptor.

16.6.2 muddled.deployments.cpio

cpio deployment.

Most commonly used to create Linux ramdisks, this deployment creates a CPIO archive from the relevant install directory and applies the relevant instructions.

Because python has no native CPIO support, we need to do this by creating a tar archive and then invoking cpio in copy-through mode to convert the archive to cpio. Ugh.

class muddled.deployments.cpio.**CIApplyChmod**

Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CIApplyChown**

Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CIApplyMknod**

Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CpioDeploymentBuilder**(*target_file*, *target_base*, *compressionMethod=None*, *pruneFunc=None*)

Bases: *muddled.depend.Action*

Builds the specified CPIO deployment.

- 'target_file' is the CPIO file to construct.
- 'target_base' is an array of pairs mapping labels to target locations, or (label, src) -> location

- 'compressionMethod' is the compression method to use, if any - gzip -> gzip, bzip2 -> bzip2.
- if 'pruneFunc' is not None, it is a function to be called like pruneFunc(Hierarchy) to prune the hierarchy prior to packing. Usually something like deb.deb_prune, it's intended to remove spurious stuff like manpages from initrds and the like.

attach_env(*builder*)
Attaches an environment containing:

MUDDLE_TARGET_LOCATION - the location in the target filesystem where this deployment will end up.

to every package label in this role.

build_label(*builder*, *label*)
Actually cpio everything up, following instructions appropriately.

class muddled.deployments.cpio.**CpioInstructionImplementor**
Bases: object

apply(*builder*, *instruction*, *role*, *path*)

class muddled.deployments.cpio.**CpioWrapper**(*builder*, *action*, *label*)
Bases: object

copy_from_role(*from_role*, *from_fragment*, *to_fragment*, *with_base=None*)
Copy the relative path from_fragment in from_role to to_fragment in the CPIO package or deployment given by 'action'

Use 'with_base' to change the base offset we apply when executing instructions; this is useful when using repeated copy_from_role() invocations to copy a subset of one role to a package/deployment.

done()
Call this once you've added all the roles you want; it attaches the deployment environment to them and generally finishes up

muddled.deployments.cpio.**create**(*builder*, *target_file*, *name*, *compressionMethod=None*, *pruneFunc=None*)
Create a CPIO deployment and return it.

- 'builder' is the muddle builder that is driving us
- 'target_file' is the name of the CPIO file we want to create. Note that this may include a sub-path (for instance, "fred/file.cpio" or even "/fred/file.cpio").
- 'name' is either:
  1. The name of the deployment that will contain this CPIO file (in the builder's default domain), or
  2. A deployment or package label, ditto
- 'comporessionMethod' is the compression method to use:
  - None means no compression
  - 'gzip' means gzip
  - 'bzip2' means bzip2
- if 'pruneFunc' is not None, it is a function to be called like pruneFunc(Hierarchy) to prune the hierarchy prior to packing. Usually something like deb.deb_prune, it's intended to remove spurious stuff like manpages from initrds and the like.

Normal usage is thus something like:

```
    fw = cpio.create(builder, 'firmware.cpio', deployment)
    fw.copy_from_role(role1, '', '/')
    fw.copy_from_role(role2, 'bin', '/bin')
    fw.done()
```

or:

```
    fw = cpio.create(builder, 'firmware.cpio', package('firmware', role))
    fw.copy_from_role(role, '', '/')
    fw.done()
```

16.6.3 muddled.deployments.filedep

File deployment. This deployment just copies files into a role subdirectory in the /deployed directory, applying appropriate instructions.

class muddled.deployments.filedep.**FIApplyChmod**
Bases: *muddled.deployments.collect.CollectApplyChmod*

needs_privilege(*builder*, *instr*, *role*, *path*)

class muddled.deployments.filedep.**FIApplyMknod**
Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.filedep.**FileDeploymentBuilder**(*roles*, *target_dir*)
Bases: *muddled.depend.Action*

Builds the specified file deployment

role is actually a list of (role, domain) pairs.

apply_instructions(*builder*, *label*)

attach_env(*builder*)
Attaches an environment containing:

MUDDLE_TARGET_LOCATION - the location in the target filesystem where this deployment will end up.

to every package label in this role.

build_label(*builder*, *label*)
Performs the actual build.

We actually do need to copy all files from install/ (where unprivileged processes can modify them) to deploy/ (where they can't).

Then we apply instructions to deploy.

deploy(*builder*, *label*)

muddled.deployments.filedep.**deploy**(*builder*, *target_dir*, *name*, *roles*)
Register a file deployment.

This is a convenience wrapper around deploy_with_domains().

'roles' is a sequence of role names. The deployment will take the roles specified, and build them into a deployment at deploy/[name].

More specifically, a rule will be created for label:

"deployment:<name>/deployed"

which depends on "package:*{<role}/postinstalled" (in the builder's default domain) for each <role> in 'roles'.

In other words, the deployment called 'name' will depend on the given roles (in the default domain) having been "finished" (postinstalled).

An "instructions applied" label "deployment:<name>/instructionsapplied" will also be created.

The deployment should eventually be located at 'target_dir'.

`muddled.deployments.filedep.`**`deploy_with_domains`**(*builder*, *target_dir*, *name*, *role_domains*)

Register a file deployment.

'role_domains' is a sequence of (role, domain) pairs. The deployment will take the roles and domains specified, and build them into a deployment at deploy/[name].

More specifically, a rule will be created for label:

"deployment:<name>/deployed"

which depends on "package:(<domain>)*{<role}/postinstalled" for each (<role>, <domain>) pair in 'role_domains'.

In other words, the deployment called 'name' will depend on the given roles in the appropriate domains having been "finished" (postinstalled).

An "instructions applied" label "deployment:<name>/instructionsapplied" will also be created.

The deployment should eventually be located at 'target_dir'.

16.6.4 muddled.deployments.tools

Tools deployment. This deployment merely adds the appropriate environment variables to use the tools in the given role install directories to everything in another list of deployments.

Instructions are ignored - there's no reason to follow them (yet) and it's simpler not to.

XXX Do we support this anymore?

class `muddled.deployments.tools.`**`ToolsDeploymentBuilder`**(*dependent_roles*)

Bases: *`muddled.depend.Action`*

Copy the dependent roles into the tools deployment.

`build_label`(*builder*, *label*)

`deploy`(*builder*, *label*)

`muddled.deployments.tools.`**`attach_env`**(*builder*, *role*, *env*, *name*)

Attach suitable environment variables for the given input role to the given environment store.

We set:

- LD_LIBRARY_PATH - Prepend $role_installl/lib
- PATH - Append $role_install/bin
- PKG_CONFIG_PATH - Prepend $role_install/lib/pkgconfig

- $role_TOOLS_PATH - Prepend $role_install/bin

The PATH/TOOLS_PATH stuff is so you can still locate tools which were in the path even if they've been overridden with your built tools.

muddled.deployments.tools.**deploy**(*builder*, *name*, *rolesThatUseThis=[]*, *rolesNeededForThis=[]*)

Register a tools deployment.

This is used to:

1. Set the environment for each role in 'rolesThatUseThis' so that PATH, LD_LIBRARY_PATH and PKG_CONFIG_PATH include the 'name' deployment
2. Make deployment:<name>/deployed depend upon the 'rolesNeededForThis'
3. Register cleanup for this deployment

The intent is that we have a "tools" deployment, which provides useful host tools (for instance, something to mangle a file in a particular manner). Those roles which need to use such tools in their builds (normally in a Makefile.muddle) then need to have the environment set appropriately to allow them to find the tools (and ideally, not system provided tools which mighth have the same name).

16.7 muddled.pkgs

16.7.1 muddled.pkgs.aptget

An apt-get package. When you try to build it, this package pulls in a pre-canned set of packages via apt-get.

class muddled.pkgs.aptget.**AptGetBuilder**(*name*, *role*, *pkgs_to_install*, *os_version=None*)

Bases: *muddled.pkg.PackageBuilder*

Make sure that particular OS packages have been installed.

The "build" action for AptGetBuilder uses the Debian tool apt-get to ensure that each package is installed.

Our arguments are:

- 'name' - the name of this builder
- 'role' - the role to which it belongs
- 'pkgs_to_install' - a sequence specifying which packages are to be installed.

Each item in the sequence 'pkgs_to_install' can be:

- the name of an OS package to install - for instance, 'libxml2-dev'

  (this is backwards compatible with how this class worked in the past)

- a Choice allowing a particular package to be selected according to the operating system.

  See "muddle doc Choice" for details on the Choice class.

  Note that a choice resulting in None (i.e., where the default value is None, and the default is selected) will not do anything.

  If 'os_version' is given, then it will be used as the version name, otherwise the result of calling utils.get_os_version_name() will be used.

We also allow a single string, or a single Choice, treated as if they were wrapped in a list.

already_installed(*pkg*)
Decide if the quoted debian package is already installed.

We use dpkg-query:

```
$ dpkg-query -W -f=\${Status}\n libreadline-dev
install ok installed
```

That third word means what it says (installed). Contrast with a package that is either not recognised or has not been downloaded at all:

```
$ dpkg-query -W -f=\${Status}\n a0d
dpkg-query: no packages found matching a0d
```

So we do some fairly simple processing of the output...

build_label(*builder*, *label*)
This time, build is the only one we care about.

muddled.pkgs.aptget.**depends_on_aptget**(*builder*, *name*, *role*, *pkg*, *pkg_role*)
Make a package dependant on a particular apt-builder.

- pkg - The package we want to add a dependency to. '*' is a good thing to add here ..

muddled.pkgs.aptget.**medium**(*builder*, *name*, *role*, *apt_pkgs*, *roles*, *os_version=None*)
Construct an apt-get package and make every package in the named roles depend on it.

Note that apt_pkgs can be an OS package name or a choices sequence - see the documentation for AptGetBuilder.

muddled.pkgs.aptget.**simple**(*builder*, *name*, *role*, *apt_pkgs*, *os_version=None*)
Construct an apt-get package in the given role with the given apt_pkgs.

Note that apt_pkgs can be an OS package name or a Choice - see the documentation for AptGetBuilder for more details.

For instance (note: not a real example - the dependencies don't make sense!):

```
from muddled.utils import Choice
from muddled.pkgs import aptget
aptget.simple(builder, "host_packages", "host_environment",
        [
        "gcc-multilib",
        "g++-multilib",
        "lib32ncurses5-dev",
        "lib32z1-dev",
        "bison",
        "flex",
        "gperf",
        "libx11-dev",
        # On Ubuntu 11 or 12, choose icedtea-7, otherwise icedtea-6
        Choice([ ("ubuntu 1[12].*", "icedtea-7-jre"),
                 ("ubuntu *", "icedtea-6-jre") ]),
        # On Ubuntu 10 or later, use libgtiff5
        # On Ubuntu 3 through 9, use libgtiff4
        # Otherwise, just don't try to use libgtiff
        Choice([ ("ubuntu 1?", "libgtiff5"),
                 ("ubuntu [3456789]", "libgtiff4"),
                 None ])
        ])
```

16.7.2 muddled.pkgs.deb

Some code which sneakily steals binaries from Debian/Ubuntu.

Quite a lot of code for embedded systems can be grabbed pretty much directly from the relevant Ubuntu binary packages - this won't work with complex packages like exim4 without some external frobulation, since they have relatively complex postinstall steps, but it works supported architecture it's a quick route to externally maintained binaries which actually work and it avoids having to build absolutely everything in your linux yourself.

This package allows you to 'build' a package from a source file in a checkout which is a .deb. We run dpkg with enough force options to install it in the relevant install directory.

You still need to provide any relevant instruction files (we'll register <filename>.instructions.xml for you automatically if it exists).

We basically ignore the package database (there is one, but it's always empty and stored in the object directory).

class muddled.pkgs.deb.**DebAction**(*name*, *role*, *co*, *pkg_name*, *pkg_file*, *instr_name=None*, *postInstallMakefile=None*)

Bases: *muddled.pkg.PackageBuilder*

Use dpkg to extract debian archives from the given checkout into the install directory.

- co - is the checkout name in which the package resides.
- pkg_name - is the name of the package (dpkg needs it)
- pkg_file - is the name of the file the package is in, relative to the checkout directory.
- instr_name - is the name of the instruction file, if any.
- postInstallMakefile - if not None:

```
make -f postInstallMakefile <pkg-name>
```

will be run at post-install time to make links, etc.

build_label(*builder*, *label*)

Build the relevant label.

ensure_dirs(*builder*, *label*)

class muddled.pkgs.deb.**DebDevAction**(*name*, *role*, *co*, *pkg_name*, *pkg_file*, *instr_name=None*, *postInstallMakefile=None*, *nonDevCoName=None*, *nonDevPkgFile=None*)

Bases: *muddled.pkg.PackageBuilder*

Use dpkg to extract debian archives into obj/include and obj/lib directories so we can use them to build other packages.

As for a DebAction, really.

build_label(*builder*, *label*)

Actually install the dev package.

ensure_dirs(*builder*, *label*)

muddled.pkgs.deb.**deb_prune**(*h*)

Given a cpiofile hierarchy, prune it so that only the useful stuff is left.

We do this by lopping off directories, which is easy enough in cpiofile heirarchies.

muddled.pkgs.deb.**dev**(*builder*, *coName*, *name*, *roles*, *depends_on=[]*, *pkgFile=None*, *debName=None*, *nonDevCoName=None*, *nonDevDebName=None*, *instrFile=None*, *postInstallMakefile=None*)

A wrapper for 'deb.simple', with the "idDev" flag set True.

- nonDevCoName is the checkout in which the non-dev version of the package resides.
- nonDevDebName is the non-dev version of the package; this is sometimes needed because of the odd way in which debian packages the '.so' link in the dev package and the sofiles themselves into the non-dev.

muddled.pkgs.deb.**extract_into_obj**(*inv*, *co_name*, *label*, *pkg_file*)

muddled.pkgs.deb.**rewrite_links**(*inv*, *label*)

muddled.pkgs.deb.**simple**(*builder*, *coName*, *name*, *roles*, *depends_on=[]*, *pkgFile=None*, *debName=None*, *instrFile=None*, *postInstallMakefile=None*, *isDev=False*, *nonDevCoName=None*, *nonDevPkgFile=None*)

Build a package called 'name' from co_name / pkg_file with an instruction file called instr_file.

'name' is the name of the muddle package and of the debian package. if you want them different, set deb_name to something other than None.

Set isDev to True for a dev package, False for an ordinary binary package. Dev packages are installed into the object directory where MUDDLE_INC_DIRS etc. expects to look for them. Actual packages are installed into the installation directory where they will be transported to the target system.

16.7.3 muddled.pkgs.initscripts

Write an initialisation script into $(MUDDLE_TARGET_LOCATION)/bin/$(something)

This is really just using utils.subst_file() with the current environment, on a resource stored in resources/.

We also write a setvars script with a suitable set of variables for running code in the context of the deployment, and any variables you've set in the environment store retrieved with get_env_store().

class muddled.pkgs.initscripts.**InitScriptBuilder**(*name*, *role*, *script_name*, *deployments*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)

Bases: *muddled.pkg.PackageBuilder*

Build an init script.

build_label(*builder*, *label*)

Install is the only one we care about ..

muddled.pkgs.initscripts.**get_effective_env**(*builder*, *name*, *role*, *domain=None*)

Retrieve the effective runtime environment for this initscripts package. Note that setting variables here will have no effect.

muddled.pkgs.initscripts.**get_env**(*builder*, *name*, *role*, *domain=None*)

Retrieve an environment to which you can make changes which will be reflected in the generated init scripts. The actual environment used will have extra values inserted from wildcarded environments - see get_effective_env() above.

muddled.pkgs.initscripts.**medium**(*builder*, *name*, *roles*, *script_name*, *deployments=[]*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)

Build an init script for the given roles.

muddled.pkgs.initscripts.**setup_default_env**(*builder*, *env_store*)

Set up the default environment for this initscript.

muddled.pkgs.initscripts.**simple**(*builder*, *name*, *role*, *script_name*, *deployments=[]*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)

Build an init script for the given role.

16.7.4 muddled.pkgs.make

Some standard package implementations to cope with packages that use Make

class muddled.pkgs.make.**ExpandingMakeBuilder**(*name*, *role*, *co_name*, *archive_file*, *archive_dir*, *makefile='Makefile.muddle'*)

Bases: *muddled.pkgs.make.MakeBuilder*

A MakeBuilder that first expands an archive file.

A MakeBuilder that first expands an archive file.

For package 'name' in role 'role', look in checkout 'co_name' for archive 'archive_file'. Unpack that into $MUDDLE_OBJ, as 'archive_dir', with 'obj/' linked to it, and use 'makefile' to build it.

build_label(*builder*, *label*)
Build our label.

Cleverly, Richard didn't define anything for MakeBuilder to do at the PreConfigure step, which means we can safely do whatever we need to do in this subclass...

unpack_archive(*builder*, *label*)

class muddled.pkgs.make.**MakeBuilder**(*name*, *role*, *co*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *rewriteAutoconf=False*, *usesAutoconf=False*, *execRelPath=None*)

Bases: *muddled.pkg.PackageBuilder*

Use make to build your package from the given checkout.

We assume that the makefile is smart enough to build in the object directory, since any other strategy (e.g. convolutions involving cp) will lead to dependency-based disaster.

Constructor for the make package.

build_label(*builder*, *label*)
Build the relevant label. We'll assume that the checkout actually exists.

ensure_dirs(*builder*, *label*)
Make sure all the relevant directories exist.

muddled.pkgs.make.**attach_env**(*builder*, *name*, *role*, *checkout*, *domain=None*)
Write the environment which attaches MUDDLE_SRC to makefiles.

We retrieve the environment for `package:<name>{<role>}/*`, and set MUDDLE_SRC therein to the checkout path for 'checkout:<checkout>'.

muddled.pkgs.make.**deduce_makefile_name**(*makefile_name*, *per_role*, *role*)
Deduce our actual muddle Makefile name.

'makefile_name' is the base name. If it is None, then we use DEFAULT_MAKEFILE_NAME.

If 'per_role' is true, and 'role' is not None, then we add the extension '.<role>' to the end of the makefile name.

Abstracted here so that it can be used outside this module as well.

muddled.pkgs.make.**expanding_package**(*builder*, *name*, *archive_dir*, *role*, *co_name*, *co_dir*, *makefile='Makefile.muddle'*, *deps=None*, *archive_file=None*, *archive_ext='.tar.bz2'*)
Specify how to expand and build an archive file.

As normal, 'name' is the package name, 'role' is the role to build it in, 'co_name' is the name of the checkout, and 'co_dir' is the directory in which that lives.

We expect to unpack an archive

<co_dir>/<co_name>/<archive_dir><archive_ext>

into $(MUDDLE_OBJ)/<archive_dir>. (NB: if the archive file does not expand into a directory of the obvious name, you can specify the archive file name separately, using 'archive_file').

So, for instance, all of our X11 "stuff" lives in checkout "X11R7.5" which is put into directory "x11" – i.e., "src/X11/X11R7.5".

That lets us keep stuff together in the repository, without leading to a great many packages that are of no direct interest to anyone else.

Within that we then have various muddle makefiles, and a set of .tar.bz archive files.

1. The archive file expands into a directory called 'archive_dir'
2. It is assumed that the archive file is named 'archive_dir' + 'archive_ext'. If this is not so, then specify 'archive_file' (and/or 'archive_ext') appropriately.

This function is used to say: take the named archive file, use package name 'name', unpack the archive file into $(MUDDLE_OBJ_OBJ), and build it using the named muddle makefile.

This allows various things to be build with the same makefile, which is useful for (for instance) X11 proto[type] archives.

Note that in $(MUDDLE_OBJ), 'obj' (i.e., $(MUDDLE_OBJ_OBJ)) will be a soft link to the expanded archive directory.

muddled.pkgs.make.**medium**(*builder*, *name*, *roles*, *checkout*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

muddled.pkgs.make.**multilevel**(*builder*, *name*, *roles*, *co_dir=None*, *co_name=None*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *repo_relative=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

muddled.pkgs.make.**simple**(*builder*, *name*, *role*, *checkout*, *rev=None*, *branch=None*, *simpleCheckout=False*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, called name with role role from the sources in checkout checkout.

- simpleCheckout - If True, register the checkout too.
- config - If True, we have make config. If false, we don't.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.
- **usesAutoconf - If True, this package is given access to .la and .pc** files from things it depends on.

- rewriteAutoconf - If True, we will rewrite .la and .pc files in the output directory so that packages which use autoconf continue to depend correctly. Intended for use with the MUDDLE_PKGCONFIG_DIRS environment variable.
- **execRelPath - Where, relative to the object directory, do we find** binaries for this package?

muddled.pkgs.make.**single**(*builder*, *name*, *role*, *deps=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

A simple make package with a single checkout named after the package and a single role.

muddled.pkgs.make.**twolevel**(*builder*, *name*, *roles*, *co_dir=None*, *co_name=None*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *repo_relative=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

CHAPTER 17

The muddled package

Muddle - A VCS-agnostic package build and configuration management system

Note: *This is a first pass at semi-automated documentation.*

It is mostly derived from docstrings within the package, and these in could certainly do with expansion and clarification. Also, the organisation/layout of this section itself leaves something to be desired. Please be patient.

17.1 Modules available via `import muddle`

After doing:

```
>>> import muddled
```

you have available:

- `muddled`
- `muddled.depend`
- `muddled.pkg`
- `muddled.repository`
- `muddled.utils`
- `muddled.vcs`
- `muddled.version_control`

17.2 Top-level modules

These are the "top-level" modules, i.e., those obtainable by (for instance):

```
>>> import muddled.depends
```

17.2.1 muddled.cmdline

Note: *Internal use, handles the muddle command line*

Main command line support for the muddle program.

See:

muddle help

for help on how to use it.

muddled.cmdline.**cmdline**(*args*, *muddle_binary=None*)
Work out what to do from a muddle command line.

'args' should be all of the "words" after the actual command name itself.

'muddle_binary' should be the __file__ value for the Python script that is calling us, or whatever other value we wish $(MUDDLE) to be set to by muddle itself. It is important to get this right, as it is used in Makefiles to run muddle itself. If it is given as None then we shall make up what should be a sensible value.

muddled.cmdline.**find_and_load**(*specified_root*, *muddle_binary*)
Find our .muddle root, and then load our builder, and return it.

muddled.cmdline.**guess_cmd_in_build**(*builder*, *current_dir*)
Returns a tuple (command_name, args)

muddled.cmdline.**lookup_command**(*command_name*, *args*, *cmd_dict*, *subcmd_dict*)
Look the command up, and return an instance of it and any remaining args

muddled.cmdline.**our_cmd**(*cmd_list*, *error_ok=True*)
Command processing for calculating muddle version

muddled.cmdline.**show_version**()
Show something akin to a version of this muddle.

Simply run git to do it for us. Of course, this will fail if we don't have git...

17.2.2 muddled.commands

Note: *Internal use, handles the muddle command line*

Muddle commands - these get run more or less directly by the main muddle command and are abstracted out here in case your programs want to run them themselves

class muddled.commands.**AnyLabelCommand**
Bases: *muddled.commands.Command*

A Command that takes any sort of label. Always requires a build tree.

We don't try to turn one sort of label into another, and we don't alter the order of the labels given. At least one label must be provided.

build_these_labels(*builder*, *checkouts*)
Do whatever is necessary to each label

decode_args(*builder*, *args*, *current_dir*)
Turn the arguments into full labels.

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Assert**

Bases: *muddled.commands.AnyLabelCommand*

Syntax muddle assert <label> [<label> ...]

Assert the given labels.

This sets the tags indicated by the specified label(s), and only those tags.

This is *not* the same as if muddle had performed the equivalent "muddle buildlabel" command, because setting the "/installed" tag in this way will not also set the "/built" (or any other) tag.

Thus this is mostly for use by experts and scripts.

Each <label> is a label fragment, in the normal manner. The <type> defaults to "package:", and the <tag> defaults to the normal default <tag> for that type. Wildcards are expanded.

<label> may also be "_all", "_default_deployments", "_default_roles" or "_just_pulled".

See "muddle help labels" for more help on label fragments and the "_xxx" values.

build_these_labels(*builder*, *labels*)

cmd_name = 'assert'

class muddled.commands.**Bootstrap**

Bases: *muddled.commands.Command*

Syntax muddle bootstrap [-subdomain] <repo> <build_name>

Create a new build tree, from scratch, in the current directory. The current directory should ideally be empty.

- <repo> should be the root URL for the repository which you will be using as the default remote location. It is assumed that it will contain a 'builds' and a 'versions' repository/subdirectory/whatever (this depends a bit on version control system being used).

  <repo> should be the same value that you would give to the 'init' command, if that was being used instead.

- <build_name> is the name for the build. This should be a simple string, usable as a filename. It is strongly recommended that it contain only alphanumerics, underline, hyphen and dot (or period). Ideally it should be a meaningful (but not too long) description of the build.

For instance:

```
$ cd project33
$ muddle bootstrap git+http://example.com/fred/ build-27
```

You will end up with a build tree of the form:

```
.muddle/
    RootRepository      -- containing "git+http://example/com/fred/"
    Description         -- containing "builds/01.py"
    VersionsRepository  -- containing "git+http://example/com/fred/versions/"
src/
    builds/
        .git/           -- assuming "http://example/com/fred/builds/"
        01.py           -- wth a bare minimum of content
versions/
        .git/           -- assuming "http://example/com/fred/versions/"
```

Note that 'src/builds/01.py' will have been *added* to the VCS (locally), but will not have been committed (this may change in a future release).

Also, muddle cannot currently set up VCS support for Subversion in the subdirectories.

If you try to do this in a directory that is itself within an existing build tree (i.e., there's a parent directory somewhere with a `.muddle` directory), then it will normally fail because you are trying to create a build within an existing build. If you are actually doing this because you are bootstrapping a subdomain, then specify the `-subdomain` switch.

Note that this command will never bootstrap a new build tree in the same directory as an existing `.muddle` directory.

allowed_in_release_build()

bootstrap(*root_path*, *args*)

cmd_name = 'bootstrap'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)
Bootstrap a build tree.

class muddled.commands.**BranchTree**
Bases: *muddled.commands.Command*

Syntax muddle branch-tree [-c[check] | -f[orce]] [-v] <branch>

Move all checkouts in the build tree (if they support it) to branch <branch>.

This works as follows:

1. First inspect each checkout, and check if:
   (a) the checkout is using a VCS which does not support this operation (which probably means it is not using git), or
   (b) the build description explicitly specifies a particular revision, or
   (c) the build description explicitly specifies a particular branch, or
   (d) it is a shallow checkout, in which case there is little point branching it as it cannot be pushed, or
   (e) the checkout already has a branch of the requested name.

   If any checkouts report problems, then the command will be aborted, and muddle will exit with status 1.
2. Then, branch each checkout as requested, and change to that branch.
3. Finally, remind the user to add "builder.follow_build_desc_branch = True" to the build description.

If the user specifies "-c" or "-check", omit steps 2 and 3 (i.e., just do the checks).

If the user specifies "-f" or "-force", omit step 1 (i.e., do not do the checks), and ignore any checkouts which do not support this operation, have an explicit revision specified, or are shallow. If a checkout already has a branch of the requested name, check it out.

If the '-v' flag is used, report on each checkout (actually, each checkout directory) as it is entered.

It is recommended that <branch> include the build name (as specified using `builder.build_name = <name>` in the build description).

Muddle does not itself provide a means of branching only some checkouts. Use the appropriate VCS commands to do that (possibly in combination with 'muddle runin').

Unexpected section title.

```
Normal usage
------------
```

1. Choose a branch name that incorporates the build name and reason for branching. For instance, "acme_stb_alpha_v1.0_maintenance".

2. Perform the branch:

```
muddle branch-tree acme_stb_alpha_v1.0_maintenance
```

3. Edit the (newly branched) build description, and add:

```
builder.follow_build_desc_branch = True
```

   to it, so that muddle knows the build tree has been branched as an entity.

4. Commit all the new branches with an appropriate message. At the moment, there is no convenient single line way to do this with muddle, but the somewhat inconvenient muddle runin command can be used, for instance:

```
muddle runin _all_checkouts git commit -a -m 'Branch for v1.0 maintenance'
```

   (this, of course, relies upon the fact that muddle only supports tree branching with git at the moment).

5. Possibly do a "muddle push _all" at this point, or perhaps do some editing, some more committing, and then push.

Unexpected section title.

```
Dealing with problems
---------------------
```

If a checkout does not support lightweight branching, then the solution is to '-force' the tree branch, and then "branch" the offending checkout by hand. The branched version of the build description will then need to be edited to indicate (in whatever manner is appropriate) the new "branch" to be used.

If the build description explicitly specifies a particular branch or revision for a checkout, then check the build out as normal, then branch the build description and remove the explicit branch or revision, and then use branch-tree to branch the checkout.

> (If you can promise absolutely that it will never be necessary to edit the offending checkout, then it would also be OK to leave the explicit branch or revision, but experience proves this is often a mistake.)

If a checkout is marked as shallow, then the solution is to edit the branched build description and specify the required revision id explicitly, to guarantee that you keep on getting the particular shallow checkout that is needed (since the main build will continue to track HEAD).

If a checkout already has a branch of the name you wanted to use, then either use it, or change the branch name you ask for - this can only be decided by yourself, because you know what the name *means*.

Note that the check for a clashing branch name is done last. That means that if you change the build description so any of the previous checks no longer fail, you still need to re-run the "check" phase to re-check for clashing branch names.

allowed_switches = {'-force': 'force', '-v': 'verbose', '-f': 'force', '-check': 'check', '-c': 'check'}

branch_checkouts(*builder*, *all_checkouts*, *branch*, *verbose*)
Branch our checkouts.

If we can't, say so but continue anyway.

If 'verbose', show each pushd into a checkout directory.

Returns the number of branched checkouts.

check_checkouts (*builder*, *checkouts*, *branch*, *verbose*)
Check if we can branch our checkouts.

Returns a list of problem reports, one per problem checkout.

cmd_name = 'branch-tree'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**Build**
Bases: *muddled.commands.PackageCommand*

Syntax muddle build [<package> ...]

Build packages.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/postinstalled". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

What is done depends upon the tags set for the package:

1. If the package has not yet been preconfigured, any preconfigure actions will be done (for most packages, this is an empty step).
2. If the package has not yet been configured, then it will be configured. This normally involves performing the actions for the "config" target in the muddle Makefile.
3. If the package has not yet been built, then it will be built. This normally involves performing the actions for the "all" target in the muddle Makefile.
4. If the package has not yet been installed, then it will be installed. This normally involves performing the actions for the "install" target in the muddle Makefile.
5. If the package has not yet been post-installed, then it will be post-installed (for most packages, this is an empty step).

Steps 1. and 2. are identical to those in "muddle configure".

This sequence is why a dependency on a package should normally be made on package:<name>{<role>}/postinstalled - that is the final stage of building any package.

build_these_labels (*builder*, *labels*)

cmd_name = 'build'

class muddled.commands.**BuildLabel**
Bases: *muddled.commands.AnyLabelCommand*

Syntax muddle buildlabel <label> [<label> ...]

Performs the appropriate actions to 'build' each <label>.

Each <label> is a label fragment, in the normal manner. The <type> defaults to "package:", and the <tag> defaults to the normal default <tag> for that type. Wildcards are expanded.

<label> may also be "_all", "_default_deployments", "_default_roles" or "_just_pulled".

See "muddle help labels" for more help on label fragments and the "_xxx" values.

Unlikes the checkout, package or deployment specific commands, buildlabel does not try to guess what to do based on which directory the command is given in. At least one <label> must be specified.

This command is mainly used internally to build defaults (specifically, when you type a bare "muddle" command in the root directory) and the privileged half of instruction executions.

build_these_labels (*builder*, *labels*)

cmd_name = 'buildlabel'

class muddled.commands.**CPDCommand**
Bases: *muddled.commands.Command*

A command that takes checkout, package or deployment arguments.

This is purely an intermediate class for common code for the classes using it (I coult have done a mixin class instead)

build_these_labels (*builder*, *checkouts*)
Do whatever is necessary to each label

current_dir = None

decode_args (*builder*, *args*, *current_dir*)
Interpret 'args' as partial labels, and return a list of proper labels.

default_args (*builder*, *current_dir*)
Decide on default labels, based on where we are in the build tree.

expand_labels (*builder*, *args*)

interpret_labels (*builder*, *args*, *initial_list*)
Turn 'initial_list' into a list of labels of the required type.

This method should attempt to GiveUp with a useful message if it would otherwise return an empty list.

original_labels = []

required_tag = None

required_type = 'checkout'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**Changed**
Bases: *muddled.commands.PackageCommand*

Syntax muddle changed <package> [<package> ...]

Mark packages as having been changed so that they will later be rebuilt by anything that needs to.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/built". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each label, unset the '/built' tag for the label, and the tags of any labels that depend on it. Note that this will include the same label with its '/installed' and '/postinstalled' tags.

Note that we don't reconfigure (or indeed clean) packages - we just clear the tags asserting that they've been built.

build_these_labels (*builder*, *labels*)

cmd_name = 'changed'

required_tag = 'built'

class muddled.commands.**Checkout**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle checkout [<checkout> ...]

Checks out the specified checkouts.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

Copies (clones/branches) the content of each checkout from its remote repository.

The value "_just_pulled" will be set to the labels of the checkouts whose working directories are altered by "muddle pull" or "muddle checkout" - i.e., those for which the "pull" or "checkout" operation did something tangible. One can then do "muddle rebuild _just_pulled" or "muddle distrebuild _just_pulled".

(The value of _just_pulled is cleared at the start of "muddle pull" or "muddle checkout", and set at the end - the list of checkout labels is actually stored in the file .muddle/_just_pulled.)

build_these_labels(*builder*, *labels*)

cmd_name = 'checkout'

class muddled.commands.**CheckoutCommand**
Bases: *muddled.commands.CPDCommand*

A Command that takes checkout arguments. Always requires a build tree.

If no explicit labels are given, then the default is to find all the checkouts below the current directory.

allowed_in_release_build()

default_args(*builder*, *current_dir*)
Decide on default labels, based on where we are in the build tree.

interpret_labels(*builder*, *args*, *initial_list*)
Turn 'initial_list' into a list of labels of the required type.

required_tag = 'checked_out'

required_type = 'checkout'

class muddled.commands.**Clean**
Bases: *muddled.commands.PackageCommand*

Syntax muddle clean [<package> ...]

Clean packages. Subsequently, packages are regarded as having been configured but not built.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/built". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each label:

1. Build the label with its tag set to '/clean'. This normally involves performing the actions for the "clean" target in the muddle Makefile.
2. Unset the '/built' tag for the label, and the tags of any labels that depend on it. Note that this will include the same label with its '/installed' and '/postinstalled' tags.

build_these_labels(*builder*, *labels*)

cmd_name = 'clean'

required_tag = 'built'

class muddled.commands.**Cleandeploy**
Bases: *muddled.commands.DeploymentCommand*

Syntax muddle cleandeploy [<deployment> ...]

Clean the named deployments, and remove their '/deployed' tags.

Note that this also deletes the 'deploy/<deployment>' directory for each deployment named (but it does not delete the overall 'deploy/' directory).

It also sets the 'clean' tag for each deployment.

<deployment> should be a label fragment specifying a deployment, or one of _all and friends, as for any deployment command. The <type> defaults to "deployment", and the deployment <tag> will be "/clean". See "muddle help labels" for more information.

If no deployments are named, what we do depends on where we are in the build tree. See "muddle help labels".

build_these_labels(*builder*, *labels*)

cmd_name = 'cleandeploy'

class muddled.commands.**Command**
Bases: object

Abstract base class for muddle commands. Stuffed with helpful functionality.

Each subclass is a muddle command, and its docstring is the "help" text for that command.

allowed_in_release_build()
Returns True iff this command is allowed in a release build (a build tree that has been created using "muddle release").

allowed_switches = {}

check_for_broken_build(*current_dir*)
Check to see if there is a "partial" build in the current directory.

Intended for use in 'without_build_tree()', in classes such as UnStamp, which want to operate in an empty directory (or, at least, one without a muddle build tree in it).

The top-level muddle code does a simple check for whether there is a build tree in the current directory before calling a 'without_build_tree()' method, but sometimes we want to be a bit more careful and check for a "partial" build tree, presumably left by a previous, failed, command.

If it finds a problem, it prints out a description of the problem, and raises a GiveUp error with retcode 4, so that muddle will exit with exit code 4.

cmd_name = '<Undefined>'

help()

no_op()
Is this is a no-op (just print) operation?

remove_switches(*args*, *allowed_more=True*)
Find any switches, remember them, return the remaining arguments.

Switches are assumed to all come before any labels. We stop with an exception if we encounter something starting with '-' that is not a recognised switch.

If 'allowed_more' is False, then the command line must end after any switches.

requires_build_tree()
Returns True iff this command requires an initialised build tree, False otherwise.

set_old_env(*old_env*)
Take a copy of the environment before muddle sets its own variables - used by commands like subst to substitute the variables in place when muddle was called rather than those that would apply when the susbt command was executed.

set_options(*opt_dict*)
Set command options - usually from the options passed to mudddle.

switches = []

with_build_tree(*builder*, *current_dir*, *args*)
Run this command with a build tree.

Arguments are:

- 'builder' is the Builder instance, as constructed from the build tree we are in.
- 'current_dir' is the current directory.
- 'args' - this is any other arguments given to muddle, that occurred after the command name.

without_build_tree(*muddle_binary*, *current_dir*, *args*)
Run this command without a build tree.

Arguments are:

- 'muddle_binary' is the location of the muddle binary - this is only needed if "muddle" is going to be run explicitly, or if a Makefile.muddle is going to use $(MUDDLE_BINARY)
- 'current_dir' is the current directory.
- 'args' - this is any other arguments given to muddle, that occurred after the command name.

class muddled.commands.**Commit**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle commit [<checkout> ...]

Commit the specified checkouts to their local repositories.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/changes_committed". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

For a centralised VCS (e.g., Subversion) where the repository is remote, this will not do anything. See the update command.

build_these_labels(*builder*, *labels*)

cmd_name = 'commit'

required_tag = 'changes_committed'

class muddled.commands.**Configure**
Bases: *muddled.commands.PackageCommand*

Syntax muddle configure [<package> ...]

Configure packages.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/configured". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

What is done depends upon the tags set for the package:

1. If the package has not yet been preconfigured, any preconfigure actions will be done.
2. If the package has not yet been configured, then it will be configured. This normally involves performing the actions for the "config" target in the muddle Makefile.

build_these_labels(*builder*, *labels*)

cmd_name = 'configure'

required_tag = 'configured'

class muddled.commands.**CopyWithout**
Bases: *muddled.commands.Command*

Syntax muddle copywithout [-f[orce]] <src-dir> <dst-dir> [<without> ...]

Many VCSs use '.XXX' directories to hold metadata. When installing files in a Makefile, it's often useful to have an operation which copies a hierarchy from one place to another without these dotfiles.

This is that operation. We copy everything from the source directory, <src-dir>, into the target directory, <dst-dir>, without copying anything which is in [<without> ...]. If you omit without, we just copy - this is a useful, working, version of 'cp -a'

If you specify -f (or -force), then if a destination file cannot be overwritten because of its permissions, and attempt will be made to remove it, and then copy again. This is what 'cp' does for its '-f' flag.

cmd_name = 'copywithout'

do_copy(*args*)

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Deploy**
Bases: *muddled.commands.DeploymentCommand*

Syntax muddle deploy <deployment> [<deployment> ...]

Build (deploy) the named deployments.

<deployment> should be a label fragment specifying a deployment, or one of _all and friends, as for any deployment command. The <type> defaults to "deployment", and the deployment <tag> will be "/deployed". See "muddle help labels" for more information.

If no deployments are named, what we do depends on where we are in the build tree. See "muddle help labels".

build_these_labels(*builder*, *labels*)

cmd_name = 'deploy'

class muddled.commands.**DeploymentCommand**
Bases: *muddled.commands.CPDCommand*

A Command that takes deployment arguments. Always requires a build tree.

default_args (*builder*, *current_dir*)
Decide on default labels, based on where we are in the build tree.

interpret_labels (*builder*, *args*, *initial_list*)
Turn 'initial_list' into a list of labels of the required type.

required_tag = 'deployed'

required_type = 'deployment'

class muddled.commands.**DistClean**
Bases: *muddled.commands.PackageCommand*

Syntax muddle distclean [<package> ...]

Distclean packages. Subsequently, packages are regarded as not having been configured or built.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/built". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each label:

1. Build the label with its tag set to '/distclean'. This normally involves performing the actions for the "distclean" target in the muddle Makefile.
2. Unset the '/preconfig' tag for the label, and the tags of any labels that depend on it. Note that this will include the same label with its '/configured','/built', '/installed' and '/postinstalled' tags.

Notes:

- The "distclean" target in the Makefile is independent of the "clean" target - "muddle distclean" does not trigger the "clean" target.
- "muddle distclean" itself does not delete the 'obj/' directory, although this is normally sensible. Muddle makefiles are thus recommended to do this themselves - for instance:

```
.PHONY: distclean
distclean:
    @rm -rf $(MUDDLE_OBJ)
```

It is possible, though, that future versions of muddle might perform this deletion.

build_these_labels (*builder*, *labels*)

cmd_name = 'distclean'

required_tag = 'built'

class muddled.commands.**Distrebuild**
Bases: *muddled.commands.PackageCommand*

Syntax muddle distrebuild [<package> ...]

A rebuild that does a distclean before attempting the rebuild.

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/postinstalled". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

1. Do a "muddle distclean" for all the labels

2.Do a "muddle build" for all the labels

build_these_labels(*builder*, *labels*)

cmd_name = 'distrebuild'

class muddled.commands.**Distribute**

Bases: *muddled.commands.CPDCommand*

Syntax muddle distribute [<switches>|-no-muddle-makefile] <name> <target_directory> [<label> ...]

•<switches> may be any of:

–--with-versions

–--with-vcs

–--no-muddle-makefile

See below for more information on each.

•<name> is a distribution name.

Several special distribution names exist:

–"_source_release" is a distribution of all checkouts, without their VCS directories.

"muddle distribute _source_release" is typically useful for generating a directory to archive with tar and send out as a source code release.

"muddle distribute -with-vcs _source_release" is a way to get a "clean copy" of the current build tree, although perhaps not as clean as starting over again from "muddle init",

–"_binary_release" - this is a distribution of all the install directories, as well as the build description checkout(s) implied by the packages distributed and (unless -no-muddle-makefiles is given) the muddle Makefiles needed by each package (as the only file in each appropriate checkout directory)

–"_deployment" - this is a distribution of the deploy directory, and all its subdirectories, as well as a version stamp. It can be useful for customers who do not yet have the ability to use muddle (as is necessary with a _binary_release).

It is generally necessary to specify the deployment labels as <label> arguments, and the labels used will be written to a MANIFEST.txt file.

Note that it is done by a different mechanism than the other commands, specificaly more or less as if the user had done:

```
muddle deploy <label> ...
mkdir -p <target_directory>
cp -a deploy <target_directory>
muddle stamp save <target_directory>/`muddle query name`.stamp
cat "muddle distribute _deployment <target_directory> <labels>"
```

Consequently, the switches are not allowed with this variant.

–"_for_gpl" is a distribution that satisfies the GPL licensing requirements. It is all checkouts that have an explicit GPL license (including LGPL), plus any licenses which depend on them, and do not explicitly state that they do not need distributing under the GPL terms, plus appropriate build descriptions.

It will fail if "propagated" GPL-ness clashes with declared "binary" or "private" licenses for any checkouts.

- "_all_open" is a distribution of all open-source licensed checkouts. It contains everything from "_for_gpl", plus any other open source licensed checkouts.

  It will fail for the same reasons that _for_gpl" fails.

- "_by_license" is a distribution of everything that is not licensed with a "private" license. It is equivalent to "_all_open" plus any proprietary source checkouts plus those parts of a "_binary_release" that are not licensed "private".

  It will fail if "_all_open" would fail, or if any of the install directories to be distributed could contain results from building "private" packages (as determined by which packages are in the appropriate role).

- <target_directory> is where to distribute to. If it already exists, it should preferably be an empty directory.
- If given, each <label> is a label fragment specifying a deployment, package or checkout, or one of _all and friends. The <type> defaults to "deployment". See "muddle help labels" for more information.

If specific labels are given, then the distribution will only concern those labels and those they depend on. Deployment labels will be expanded to all of the packages that the deployment depends upon. Package labels (including those implied by deployment labels) will be remembered, and also expanded to the checkouts that they depend directly upon. Checkout labels (including those implied by packages) will be remembered. When the distribution is calculated, only packages and checkouts that have been remembered will be candidates for distribution.

If no labels are given, then the whole of the build tree is considered.

If the -with-versions switch is specified, then if there is a stamp "versions/" directory it will also be copied. By default it is not.

If the -with-vcs switch is specified, then VCS "special" files (that is, ".git", ".gitignore", ".gitmodules" for git, and so on) are requested:

- for the build description directories
- for the "versions/" directory, if it is being copied
- to all checkouts in a "_source_release" distribution

It does not apply to checkouts specified with "distribute_checkout" in the build description, as they use the "copy_vcs_dirs" argument to that function instead.

If the -no-muddle-makefile switch is specified, then the _binary_release distribution will not include Muddle makefiles for each package distributed. It does not override the setting of the "with_muddle_makefile" argument explicitly set in any calls of "distribute_package" in the build description, nor does it stop distribution of any extra files explicitly chosen with "distribute_checkout_files" in the build description. It also does not affect the "_by_license" distribution.

Note that "muddle -n distribute" can be used in the normal manner to see what the command would do. It shows the labels that would be distributed, and the actions that would be used to do so. This is especially useful for the "_source_release" and "_binary_release" commands. Output will typically be something like:

```
$ m3 -n distribute -with-vcs _binary_release ../fred
Writing distribution _binary_release to ../fred
checkout:builds/distributed        DistributeBuildDescription: _binary_release[vcs]
checkout:main_co/distributed       DistributeCheckout: _binary_release[1], role-x86[*]
package:main_pkg{arm}/distributed  DistributePackage: _binary_release[install]
package:main_pkg{x86}/distributed  DistributePackage: _binary_release[install], role-x86[obj,ins
```

- For each action, all the available distribution names are listed.
- Each distribution name may be followed by values in [..], depending on what action it is associated with.

- For a DistributeBuildDescription, the value may be [vcs], or [-<n>], or [vcs, -<n>]. 'vcs' means that VCS files will be distributed. A negative number indicates the number of "private" files that will not be distributed.
- For a DistributeCheckout, the values are [*], [<n>], [*,vcs] or [<n>,vcs]*. '' means that all files will be distributed, a single integer (<n>) that just that many specific files have been selected for distribution. [1] typically means the muddle Makefile, or perhaps a license file. A 'vcs' means that the VCS files will be distributed.
- For a DistributePackage, the values are [obj], [install] or [obj,install], indicating if "obj" or "install" directories are being distributed. It's also possible (but not much use) to have a DistributePackage distribution name that doesn't do either.

See also "muddle query checkout-licenses" for general information on the licenses in the current build, and "muddle query role-licenses" for how licenses are distributed between the roles in the build. Both of these will report on license clashes that appear to exist.

BEWARE: THIS COMMAND IS STILL NEW, AND DETAILS MAY CHANGE

In particular, the "-no-muddle-makefile" switch may go away, the details of use of the "-copy-vcs" switch may change. and the standard distribution names may change.

allowed_switches = {'-no-muddle-makefile': 'no-muddle-makefile', '-with-vcs': 'with-vcs', '-with-versions': 'with-ve

cmd_name = 'distribute'

deployment(*builder*, *current_dir*, *target_dir*, *fragments*)
Do "muddle distribute _deployment".

interpret_labels(*builder*, *args*, *initial_list*)
Return selected packages and checkouts.

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)
We're sufficiently unlike other commands to do this ourselves.

class muddled.commands.**Doc**
Bases: *muddled.commands.Command*

Syntax muddle doc [<switch> ...] [<what>]

To get documentation on modules, classes, methods or functions in muddle, use:

```
muddle doc <name>               for help on <name>
muddle doc -contains <what>     to list all names that contain <what>
```

There are also (mainly for use in debugging muddle itself - beware, they all produce long output):

```
muddle doc -duplicates     to list all duplicate (partial) names
muddle doc -list           to list all the "full" names we know
muddle doc -dump           to dump the internal map of names/values
```

<switch> may also be:

```
-p[ager] <pager>    to specify a pager through which the text will be
                    piped. The default is $PAGER (if set) or else
                    'more'.
-nop[ager]          don't use a pager, just print the text out.
-pydoc              Use pydoc's rendering to output the text about the
                    item. This tends to produce more information. It
                    is also (more or less) the format that the older
                    "muddle doc" command used.
```

The plain "muddle doc <name>" can be used to find out about any muddle module, class, method or function. Leading parts of the name can be omitted ("Builder" and "mechanics.Builder" and "muddled.mechanics.Builder" are all the same), provided that doesn't make <name> ambiguous, and if it does, you will be given a list of the possible alternatives. So, for instance:

```
$ muddle doc Builder
```

will report on muddled.mechanics.Builder, but:

```
$ muddle doc simple
```

will give a list of all the names that contain 'simple'.

If you're not sure of a name, then "-contains" can be used to look for all the (full names - i.e., starting with "muddled.") that contain that string. For instance:

```
$ muddle doc -contains absolute
The following names contain "absolute":
  muddled.checkouts.multilevel.absolute
  muddled.checkouts.simple.absolute
  muddled.checkouts.twolevel.absolute
```

and one can then safely do:

```
$ muddle doc simple.absolute
```

For a module, the module docstring is reported, and then a list of the names of all the classes and functions in that module.

For a class, the classes it inherits from, its docstring and its __init__ method are all reported, followed by a list of the methods in that class.

For a method or function, its argument list (signature) and docstring are reported.

If "-pydoc" is specified, then the layout of various things will be different, and also the full documentation of internal items (methods inside classes, etc.) will be reported - this can lead to substantially longer output.

cmd_name = 'doc'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Env**

Bases: *muddled.commands.PackageCommand*

Syntax muddle env <language> <mode> <name> <label> [<label> ...]

Produce a setenv script in the requested language listing all the runtime environment variables bound to <label> (or the cumulation of the variables for several labels).

- <language> may be 'sh', 'c', or 'py'/'python'
- <mode> may be 'build' (build time variables) or 'run' (run-time variables)
- <name> is used in various ways depending upon the target language. It should be a legal name/symbol in the aforesaid target language (for instance, in C it will be uppercased and used as part of a macro name).
- <label> should be a label fragment specifying a package, or one of _all and friends, as for any package command. See "muddle help labels" for more information.

So, for instance:

```
$ muddle env sh run 'encoder_settings' encoder > encoder_vars.sh
```

might produce a file `encoder_vars.sh` with the following content:

```
# setenv script for encoder_settings
# 2010-10-19 16:24:05

export BUILD_SDK=y
export MUDDLE_TARGET_LOCATION=/opt/encoder/sdk
export PKG_CONFIG_PATH=$MUDDLE_TARGET_LOCATION/lib/pkgconfig:$PKG_CONFIG_PATH
export PATH=$MUDDLE_TARGET_LOCATION/bin:$PATH
export LD_LIBRARY_PATH=$MUDDLE_TARGET_LOCATION/lib:$LD_LIBRARY_PATH

# End file.
```

build_these_labels(*builder*, *args*)

cmd_name = 'env'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Help**

Bases: *muddled.commands.Command*

To get help on commands, use:

muddle help [<switch>] [<command>]

specifically:

```
muddle help <cmd>          for help on a command
muddle help <cmd> <subcmd> for help on a subcommand
muddle help _all           for help on all commands
muddle help <cmd> _all     for help on all <cmd> subcommands
muddle help commands       just list the (top level) commands
muddle help categories     shows command names sorted by category
muddle help labels         for help on using labels
muddle help subdomains     for help on subdomains
muddle help aliases        says which commands have more than one name
muddle help vcs [name]     name the supported version control systems,
                           or give details about one in particular
muddle help environment    list the environment variables muddle defines
                           for use in muddle Makefiles
```

<switch> may be:

```
-p[ager] <pager>    to specify a pager through which the help will be piped.
                    The default is $PAGER (if set) or else 'more'.
-nop[ager]          don't use a pager, just print the help out.
```

cmd_name = 'help'

command_line_help = 'Usage:\n\n muddle [<options>] <command> [<arg> ...]\n\nAvailable <options> are:\n\n –help,

get_help(*args*)

Return help for args, or a summary of all commands.

help_aliases()

Return a list of all commands with aliases

help_all()

Return help for all commands

help_categories()

help_command_list()
Return a list of the top-level commands.

help_environment(*args*)
Return help on MUDDLE_xxx environment variables

help_label_absent = ‘“muddle” with no label arguments\n————————————\nMuddle tries quite hard to do t

help_label_all_and_friends = ‘_all and friends\n————————-\nThere are some special command line arguments

help_label_fragments = ‘Label fragments\n————————\nTyping all of a label on the command line can be onerous.

help_label_star = ‘Unexpected results\n————————\nNote: at a Unix shell, typing:\n\n $ muddle build *\n\nis unli

help_label_summary = ‘More complete documentation on labels is available in the muddle documentation\nat http://

help_label_wrong = ‘How “muddle” commands intepret labels of the “wrong” type\n————————————————————

help_labels(*args*)
Help on labels within muddle.

“muddle help label” and “muddle help labels” are equivalent.

help label - show this text help label summary - a summary of how labels work help label fragments - using partial labels in commands help label fragment - the same help label absent - what “muddle” or “muddle <cmd>” does with no label

Unexpected indentation.

arguments

Block quote ends without a blank line; unexpected unindent.

help label wrong - what “muddle <cmd>” does with a label of the “wrong” type (e.g., “muddle build checkout:something”)

Definition list ends without a blank line; unexpected unindent.

help label _all help label <any name starting with underscore>

Unexpected indentation.

•these explain the “special” command line arguments

Block quote ends without a blank line; unexpected unindent.

help label star - the unexpected result of “muddle build *” and its like help label everything - all the “muddle help label” subtopics

Inline emphasis start-string without end-string.

help_subcmd_all(*cmd_name*, *sub_dict*)
Return help for all commands in this dictionary

help_subdomains()
Return help on how to use subdomains

help_summary()
Return a summary of usage and a list of all commands

help_vcs(*args*)
Return help on supported VCS

print_help(*args*)

requires_build_tree()

subdomains_help = 'Your build contains subdomains if "muddle query domains" prints out subdomain\nnames. In th

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Import**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle import [<checkout> ...]

Assert that the given checkouts (which may include the builds checkout) have been checked out.

This is mainly used when you've just written a package you plan to commit to the central repository - muddle obviously can't check it out because the repository doesn't exist yet, but you probably want to add it to the build description for testing (and in fact you may want to commit it with muddle push). For convenience in the expected use case, it goes on to prime the relevant VCS module (by way of "muddle reparent") so it can be pushed once ready; this should be at worst harmless in all cases.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

This command is really just a wrapper to "muddle assert" and "muddle reparent", with the right magic label names.

build_these_labels(*builder*, *labels*)

cmd_name = 'import'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Init**
Bases: *muddled.commands.Command*

Syntax muddle init <repository> <build_description>

Or muddle init -b[ranch] <branch_name> <repository> <build_description>

Initialise a new build tree with a given repository and build description. We check out the build description but don't actually build anything. It is traditional to create a new muddle build tree in an empty directory.

For instance:

```
$ mkdir project32
$ cd project32
$ muddle init  git+file:///somewhere/else/examples/d  builds/01.py
```

This initialises a muddle build tree, creating two new directories:

- '.muddle/', which contains the build tree state, and
- 'src/', which contains the build description in 'src/builds/01.py' (complex build desscriptions may use multiple Python files, and so other files may have been checked out into 'src/builds/')

You haven't told muddle which actual repository the build description is in - you've only told it where the repository root is and where the build description file is. Muddle assumes that <repository> "git+file:///somewhere/else/examples/d" and <build_description> "builds/01.py" means repository "git+file:///somewhere/else/examples/d/builds" and file "01.py" therein.

If the -branch switch is given, then the named branch of the build description will be checked out. It thus an error if either the muddle support for the build description VCS does not support this (at the moment, that probably means "not git"), or if there is no such branch.

Note: if you find yourself trying to "muddle init" a subdomain, don't. Instead, add the subdomain to the current build description (using a call of 'include_domain()'), and it will automatically get checked out during the "muddle init" of the top-level build. Or see "muddle bootstrap -subdomain".

cmd_name = 'init'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)
Initialise a build tree.

class muddled.commands.**Instruct**
Bases: *muddled.commands.Command*

Syntax muddle instruct <package>{<role>} <instruction-file>

Or muddle instruct (<domain>)<package>{<role>} <instruction-file>

Sets the instruction file for the given package name and role to the file specified in instruction-file. The role must be explicitly given as it's considered more likely that bugs will be introduced by the assumption of default roles than they are likely to prove useful.

This command is typically issued by 'make install' for a package, as:

```
$(MUDDLE_INSTRUCT) <instruction-file>
```

If you don't specify an instruction file, we will unregister instructions for this package and role.

If you want to clear all instructions, you'll have to edit the muddle database directly - this leaves the database in an inconsistent state - there's no guarantee that the instruction files will ever be rebuilt correctly - so it is not a command.

You can list instruction files and their ordering with "muddle query inst-files".

cmd_name = 'instruct'

decode_package_label(*builder*, *arg*, *tag*)
Convert a 'package' or 'package{role}' or '(domain)package{role} argument to a label.

If role or domain is not specified, use the default (which may be None).

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Merge**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle merge [-s[top]] [<checkout> ...]

Merge the specified checkouts from their remote repositories.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/merged". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each checkout named, retrieve changes from the corresponding remote repository (as described by the build description) and merge them (into the checkout). The merge process is handled in a VCS specific manner, as each checkout is dealt with.

The value "_just_pulled" will be set to the labels of the checkouts whose working directories are altered by "muddle merge" - i.e., those for which the "merge" operation did something tangible. One can then do "muddle rebuild _just_pulled" or "muddle distrebuild _just_pulled".

> (The value of _just_pulled is cleared at the start of "muddle merge", and set at the end - the list of checkout labels is actually stored in the file .muddle/_just_pulled.)

If '-s' or '-stop' is given, then we'll stop at the first problem, otherwise an attempt will be made to process all the checkouts, and any problems will be re-reported at the end.

allowed_switches = {'-stop': 'stop', '-s': 'stop'}

build_these_labels(*builder*, *labels*)

cmd_name = 'merge'

required_tag = 'merged'

class muddled.commands.**PackageCommand**
Bases: *muddled.commands.CPDCommand*

A Command that takes package arguments. Always requires a build tree.

default_args(*builder*, *current_dir*)
Decide on default labels, based on where we are in the build tree.

interpret_labels(*builder*, *args*, *initial_list*)
Turn 'initial_list' into a list of labels of the required type.

required_tag = 'postinstalled'

required_type = 'package'

class muddled.commands.**Pull**
Bases: *muddled.commands.CheckoutCommand*

> **Syntax** muddle pull [-s[top]] [-noreload] [<checkout> ...]

Pull the specified checkouts from their remote repositories. Any problems will be (re)reported at the end.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/pulled". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each checkout named, retrieve changes from the corresponding remote repository (as described by the build description) and apply them (to the checkout), but *not* if a merge would be required.

> (For a VCS such as git, this actually means "not if a user-assisted merge would be required" - i.e., fast-forwards will be done.)

The value "_just_pulled" will be set to the labels of the checkouts whose working directories are altered by "muddle pull" or "muddle checkout" - i.e., those for which the "pull" or "checkout" operation did something tangible. One can then do "muddle rebuild _just_pulled" or "muddle distrebuild _just_pulled".

> (The value of _just_pulled is cleared at the start of "muddle pull" or "muddle checkout", and set at the end - the list of checkout labels is actually stored in the file .muddle/_just_pulled.)

Normally, "muddle pull" will attempt to pull all the chosen checkouts, re-reporting any problems at the end. If '-s' or '-stop' is given, then it will instead stop at the first problem.

Unexpected section title.

```
How build descriptions are treated specially
--------------------------------------------
```

If the build description is in the list of checkouts that should be pulled, either explicitly or after expanding one of _all and friends, then "muddle pull" will:

1. Remember exactly what the user asked for on the command line.
2. Pull the build description.
3. If the build description changed (i.e., was pulled), then reload it, and re-expand the labels from the command line - so, for instance, _all might change if the new build description has added or removed checkouts.
4. Remove the build description from this (new) list of checkouts, and pull any that are left.

Earlier versions of muddle (before v2.5.1) did not do this, which meant that one might do "muddle pull _all" and then have to do it again if the build description had changed. It was easy to forget to check for this, which could leave a build tree not as up-to-date as one might think.

If you *do* want the older, simpler mechanism, then:

```
muddle pull -noreload <arguments>
```

can be used, which will just pull the labels on the command line, without treating the build description specially.

Unexpected section title.

```
What about subdomains?
----------------------
```

If your build contains subdomains, then all of the subdomain build descriptions will be treated specially. Specifically, each requested build description is pulled in domain order, reloading the top-level build description and re-evaluating the command line each time.

Unexpected section title.

```
Other commands
--------------
```

Note that "muddle merge" and "muddle pull-upstream" do not behave in this manner, as it is believed that they are used in a more direct manner (with explicit labels).

allowed_switches = {'-stop': 'stop', '-noreload': 'noreload', '-s': 'stop'}

build_these_labels(*builder*, *labels*)

calc_build_descriptions(*builder*, *done=None*)
Calculate all the build descriptions in this build tree.

Remove any that have already been 'done'

Return a list of build description checkout labels, ordered by domain.

cmd_name = 'pull'

delete_pyc_files(*builder*, *co_label*)
Delete .pyc files in this checkout

handle_build_descriptions_first(*builder*, *labels*)
Pull our build descriptions before anything else.

Returns:

- the new top-level builder

- an amended list of the checkout labels still to pull.

label_names (*labels*)

pull (*builder*, *co_label*)
Do the work of pulling checkout 'co_label'

required_tag = 'pulled'

class muddled.commands.**PullUpstream**
Bases: *muddled.commands.UpstreamCommand*

Syntax muddle pull-upstream [<checkout> ...] -u[pstream] <name> ...

For each checkout, pull from the named upstream repositories.

Specifically, retrieve changes from the corresponding remote repository, and apply them (to the checkout), but *not* if a merge would be required.

(For a VCS such as git, this actually means "not if a user-assisted merge would be required" - i.e., fast-forwards will be done.)

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

The -u or -upstream switch is required, and must be followed by at least one upstream repository name. If a checkout does not have an upstream of that name, it will be ignored.

So, for instance:

```
pushd src/checkout1
muddle pull-upstream -u upstream1 upstream2
```

or:

```
muddle pull-upstream package:android{x86} -u upstream-android
```

Note that, unlike the normal "muddle pull" command, there is no -stop switch. Instead, we always stop at the first problem. Not finding an upstream with the right name does not count as a "problem" for this purpose.

Also, pull-upstream does not alter the meaning of "_just_pulled".

Use "muddle query upstream-repos [<checkout>]" to find out about the available upstream repositories.

allowed_switches = {}

cmd_name = 'pull-upstream'

direction = 'from'

do_our_verb (*builder*, *co_label*, *vcs_handler*, *upstream*, *repo*)

required_tag = 'checked_out'

verb = 'pull'

verbing = 'Pulling'

class muddled.commands.**Push**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle push [-s[top]] [<checkout> ...]

Push the specified checkouts to their remote repositories.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/changes_pushed". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

This updates the content of the remote repositories to match the local checkout.

If '-s' or '-stop' is given, then we'll stop at the first problem, otherwise an attempt will be made to process all the checkouts, and any problems will be re-reported at the end.

"muddle push" will refuse to push if the checkout is not on the expected branch, either an explicit branch from the build description, or the build description branch if we are "following" it, or "master".

allowed_switches = {'-stop': 'stop', '-s': 'stop'}

build_these_labels(*builder*, *labels*)

cmd_name = 'push'

required_tag = 'changes_pushed'

class muddled.commands.**PushUpstream**

Bases: *muddled.commands.UpstreamCommand*

Syntax muddle push-upstream [<checkout> ...] -u[pstream] <name> ...

For each checkout, push to the named upstream repositories.

This updates the content of the remote repositories to match the local checkout.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

The -u or -upstream switch is required, and must be followed by at least one upstream repository name. If a checkout does not have an upstream of that name, it will be ignored.

So, for instance:

```
pushd src/checkout1
muddle push-upstream -u upstream1 upstream2
```

or:

```
muddle push-upstream package:android{x86} -u upstream-android
```

Note that, unlike the normal "muddle push" command, there is no -stop switch. Instead, we always stop at the first problem. Not finding an upstream with the right name does not count as a "problem" for this purpose.

Use "muddle query upstream-repos [<checkout>]" to find out about the available upstream repositories.

allowed_switches = {}

cmd_name = 'push-upstream'

direction = 'to'

do_our_verb(*builder*, *co_label*, *vcs_handler*, *upstream*, *repo*)

required_tag = 'checked_out'

verb = 'push'

verbing = 'Pushing'

class muddled.commands.**QueryBuildDescBranch**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query build-desc-branch

Report the branch of the build description, and whether it is being used as the (default) branch for other checkouts.

If there are sub-domains in the build tree, then this reports the branch of the top-level build description, which is the only build description that can request checkouts to "follow" its branch.

cmd_name = 'query build-desc-branch'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutBranches**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-branches

Print the known checkouts and their branches.

For each checkout, reports its current branch, and the branch implied (or explicitly requested) by the build description, and the branch it is "following" (if the build description has set this).

For instance:

```
--------  --------------  ---------------  ----------------
Checkout  Current branch  Original branch  Branch to follow
--------  --------------  ---------------  ----------------
builds    master          <none>           <not following>
co1       master          <none>           <not following>
co2       <can't tell>    <none>           <not following>
```

In this example, both checkouts are in git (which is the only VCS for which muddle really supports branches), and on master.

The "original branches" are both <none>. The "builds" checkout contains the build description, and its original checkout is <none> because "muddle init" did not specify a branch. The original checkouts for "co1" and "co2" are <none> because the build description did not specify explicit branches for them. We can't tell what the current branch is for co2, which normally means that it has not yet been checked out.

Since the build description does not set "builder.follow_desc_build_branch = True", all the checkouts show as <not following>.

Here is a slightly more complicated case:

```
--------  --------------   ---------------  ----------------
Checkout  Current branch   Original branch  Branch to follow
--------  --------------   ---------------  ----------------
builds    test-v0.1        branch0          <it's own>
co1       test-v0.1        <none>           test-v0.1
co2       branch1          branch1          <none>
co3       <none>           <none>           <none>
co4       <none>           branch1          <none>
co5       <not supported>  ...              <not following>
```

This build tree was created using "muddle init -branch branch0", so the builds checkout shows "branch0" as its original branch. We can tell that the build description *does* have "builder.follow_desc_build_branch = True" because there are values in the "Branch to follow" column. The build description always follows itself.

"co1" doesn't specify a particular branch in the build description, so its "original branch" is <none>. However, it does follow the build description, so has "test-v0.1" in the "Branch to follow" column.

"co2" explicitly specifies "branch1" in the build description. It got checked out on "branch1", and is still on it. Having an explicit branch means it does not follow the build description.

"co3" explicitly specified a *revision" in the build description. This means that it got checked out on a detached HEAD, and thus its current branch is <none> - it really isn't on a branch.

Inline emphasis start-string without end-string.

"co4" explicitly specified a branch ("branch1") and a revision in the build description. The revision id specified didn't correspond to HEAD of the branch, so it too is on a detached HEAD. However, "branch1" still shows up as its original branch.

Finally, "co5" is not using git (it was actually using bzr), and thus muddle does not support branching it. However, it has set the "no_follow" VCS option in the build description, and thus the "Branch to follow" column shows as "<not following>" instead of "...".

(You can use "muddle query checkout-vcs" to see which VCS is being used for which checkout.)

cmd_name = 'query checkout-branches'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutDirs**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-dirs

Print the known checkouts and their checkout paths (relative to 'src/')

cmd_name = 'query checkout-dirs'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutId**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-id [<label>]

Report the VCS revision id (or equivalent) for the named checkout.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'checkout:'. If the label is a 'package:' label, and that package depends upon a single checkout, then report the id for that checkout.

If <label> is not given, and the current directory is within a checkout directory, then use that checkout.

The id returned is that which would be written to a stamp file as the checkout's revision id.

cmd_name = 'query checkout-id'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutLicenses**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-licenses

Print information including:

- the known checkouts and their licenses
- which checkouts (if any) have GPL licenses of some sort
- which checkouts are "implicitly" GPL licensed because of depending on a GPL-licensed checkout
- which packages have declared that they don't actually need to be "implicitly" GPL

•which checkouts have irreconcilable clashes between "implicit" GPL licenses and their actual license.

Note that "irreconcilable clashes" are only important if you intend to distribute the clashing items to third parties.

See also "muddle query role-licenses" for licenses applying to (packages in) each role.

cmd_name = 'query checkout-licenses'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutRepos**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-repos [-u[rl]]

Print the known checkouts and their checkout repositories

With '-u' or '-url', print the repository URL. Otherwise, print the full spec of the Repository instance that represents the repository.

So, for instance, the standard printout produces lines of the form:

```
checkout:kernel/* -> Repository('git', 'ssh://git@server/project99/src', 'kernel', prefix='linux
```

but with '-u' one would instead see:

```
checkout:kernel/* -> ssh://git@server/project99/src/linuxbase/kernel
```

allowed_switches = {'-u': 'url', '-url': 'url'}

cmd_name = 'query checkout-repos'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckoutVcs**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkout-vcs

Print the known checkouts and their version control systems. Also prints the VCS options for the checkout, if there are any.

cmd_name = 'query checkout-vcs'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCheckouts**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query checkouts [-j]

Print the names of all the checkouts described in the build description.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query checkouts'

with_build_tree (*builder*, *current_dir*, *args*)

class muddled.commands.**QueryCommand**
Bases: *muddled.commands.Command*

The base class for 'query' commands

get_label (*builder*, *args*)

get_label_from_fragment (*builder*, *args*, *default_type='package'*)

requires_build_tree()

class muddled.commands.**QueryDefaultDeployments**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query default-deployments [-j]

Print the names of the default deployments described in the build description (as defined using 'builder.by_default_deploy()').

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query default-deployments'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryDefaultRoles**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query default-roles [-j]

Print the names of the default roles described in the build description (as defined using 'builder.add_default_role()').

These are the roles that will be assumed for 'package:' label fragments.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query default-roles'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryDepend**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query dependencies <what>

Or muddle query dependencies <what> <label>

Print the current dependency sets.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

If no label is given, then all dependencies in the current build tree will be shown.

In order to show all dependency sets, even those where a given label does not actually depend on anything, <what> can be:

- system - Print synthetic dependencies produced by the system
- user - Print dependencies entered by the build description
- all - Print all dependencies

To show only those dependencies where there *is* a dependency, add '-short' (or '_short') to <what>, i.e.:

- system-short - Print synthetic dependencies produced by the system
- user-short - Print dependencies entered by the build description
- all-short - Print all dependencies

cmd_name = 'query dependencies'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryDeployments**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query deployments [-j]

Print the names of all the deployments described in the build description.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query deployments'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryDir**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query dir <label>

Print a directory:

- for checkout labels, the checkout directory
- for package labels, the install directory
- for deployment labels, the deployment directory

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

Typically used in a muddle Makefile, as for instance:

```
KBUS_INSTALLDIR:=$(shell $(MUDDLE) query dir package:kbus{*})
```

cmd_name = 'query dir'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryDistributions**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query distributions

List the names of the distributions defined by the build description, and the license categories that each distributes.

cmd_name = 'query distributions'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**QueryDomains**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query domains [-j]

Print the names of all the subdomains described in the build description (and recursively in the subdomain build descriptions).

Note that it does not report the '' (top level) domain, as that is assumed.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query domains'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryEnv**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query env <label>

Print the environment in which this label will be run.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query env'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryEnvs**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query all-env <label>

Print a list of the environments that will be merged to create the resulting environment for this label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query all-env'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryInstDetails**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query inst-details <label>

Print the list of actual instructions for this label, in the order in which they will be applied.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query inst-details'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryInstFiles**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query inst-files <label>

Print the list of currently registered instruction files, in the order in which they will be applied.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query inst-files'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryKernelver**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query kernelver <label>

Determine the Linux kernel version.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

<label> should be the package label for the kernel version. This command looks in <obj>/obj/include/linux/version.h (where <obj> is the directory returned by "muddle query objdir <label>") for the LINUX_VERSION_CODE definition, and attempts to decode that.

It prints out the Linux version, e.g.:

```
muddle query kernelver package:linux_kernel{boot}/built
2.6.29
```

cmd_name = 'query kernelver'

kernel_version(*builder*, *kernel_pkg*)
Given the label for the kernel, determine its version.

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryLicenses**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query licenses

Print the standard licenses we know about.

See "muddle query checkout-licenses" to find out about any licenses defined, and used, in the build description.

cmd_name = 'query licenses'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**QueryLocalRoot**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query localroot <label>

Print the "local root" directory for a label.

For a label representing a checkout, package or deployment in the top-level, prints out the normal root directory (as "muddle query root").

For a label in a subdomain, printes out the root directory for said subdomain (i.e., the directory containing its .muddle/ directory).

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query localroot'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryMakeEnv**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query make-env <label>

Print the environment in which "make" will be called for this label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

Specifically, print what muddle adds to the environment (so it leaves out anything that was already in the environment when muddle was called). Note that various things (lists of directories) only get set up when the directories actually exists - so, for instance, MUDDLE_INCLUDE_DIRS will only include directories for the packages depended on *that have already been built*. This means that this command shows the environment actually as would be used if one did `muddle buildlabel`, but not necessarily as it would be for `muddle build`, when the dependencies themselves would be built first. (It would be difficult to do otherwise, as the environment built is always as small as possible, and it is not until a package has been built that muddle can tell which directories will be present.

cmd_name = 'query make-env'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryMatch**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query match <label>

Print out any labels that match the label given. If the label is not wildcarded, this just reports if the label is known.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query match'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryName**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query name

Print the build name, as specified in the build description with:

```
builder.build_name = "Project32"
```

This prints just the name, so that one can use it in the shell - for instance in bash:

```
export PROJECT_NAME=$(muddle query name)
```

or in a muddle Makefile:

```
build_name:=$(shell $(MUDDLE) query name)
```

cmd_name = 'query name'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryNeededBy**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query needed-by <label>

Print what we need to build to build this label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query needed-by'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryNeeds**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query needs <label>

Print what this label is required to build.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query needs'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryObjdir**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query objdir <label>

Print the object directory for a label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

Typically used in a muddle Makefile, as for instance:

```
KBUS_OBJDIR:=$(shell $(MUDDLE) query objdir package:kbus{*})
```

cmd_name = 'query objdir'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryPackageRoles**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query package-roles [-j]

Print the names of all the packages, and their roles, as described in the build description.

Note that if there is a rule for a package with a wildcarded name, like "package:*{x86}/*", then '*' will be included in the names printed.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query package-roles'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryPackages**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query packages [-j]

Print the names of all the packages described in the build description.

Note that if there is a rule for a package with a wildcarded name, like "package:*{x86}/*", then '*' will be included in the names printed.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query packages'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryPreciseEnv**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query precise-env <label>

Print the environment pertaining to exactly this label (no fuzzy matches)

cmd_name = 'query precise-env'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryRelease**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query release [-labels]

Print information about this build as a release, including the release specification, and the "translation" of the special "_release" argument.

(That content, what is to be released, is defined in the build description, using 'builder.add_to_release_build()'.)

For instance:

```
$ muddle query release
This is a release build
Release spec:
  name        = simple
  version     = v1.0
  archive     = tar
  compression = gzip
  hash        = c7c10cf4d6da4519714ac334a983ab518c68c5d1
What to release (the meaning of "_release", before expansion):
  _default_deployments
  package:(subdomain2)second_pkg{x86}/*
```

or:

```
$ muddle query release
This is NOT a release build
Release spec:
  name        = None
  version     = None
  archive     = tar
  compression = gzip
  hash        = None
What to release (the meaning of "_release", before expansion):
  _default_deployments
  package:(subdomain2)second_pkg{x86}/*
```

If nothing has been designated for release, then that final clause will be replaced with:

```
What to release (the meaning of "_release"):
  <nothing defined>
```

With the '-labels' switch, just prints out that last list of "what to release":

```
$ muddle query release -labels
_default_deployments
package:(subdomain2)second_pkg{x86}/*
```

The '-labels' variant prints nothing out if nothing has been designated for release.

allowed_switches = {'-labels': 'labels'}

cmd_name = 'query release'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryRoleLicenses**

Bases: *muddled.commands.QueryCommand*

Syntax muddle query role-licenses [-no-clashes]

Print the known roles and the licenses used within them (i.e., by checkouts used by packages with those roles).

If -no-clashes is given, then don't report binary/private license clashes (which might cause problems when doing a "_by_license" distribution).

See also "muddle query checkout-licenses" for information on licenses with respect to checkouts.

allowed_switches = {'-no-clashes': 'no-clashes'}

cmd_name = 'query role-licenses'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryRoles**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query roles [-j]

Print the names of all the roles described in the build description.

With '-j', print them all on one line, separated by spaces.

allowed_switches = {'-j': 'join'}

cmd_name = 'query roles'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryRoot**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query root

Print the root path, the path of the directory containing the '.muddle/' directory.

For a build containing subdomains, this means the root directory of the top-level build.

The root is where "muddle where" will print "Root of the build tree".

cmd_name = 'query root'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryRules**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query rules <label>

Print the rules covering building this label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query rules'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryTargets**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query targets <label>

Print the targets that would be built by an attempt to build this label.

<label> is a label or label fragment (see "muddle help labels"). The default type is 'package:'.

cmd_name = 'query targets'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryUnused**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query unused [<label> [...]]

Report on labels that are defined in the build description, but are not "used" by the targets. With no arguments, the targets are the default deployables. The argument "_all" means all available deployables (not just the defaults). Otherwise, arguments are labels.

cmd_name = 'query unused'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryUpstreamRepos**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query upstream-repos [-u[rl]] [<co_label>]

Print information about upstream repositories.

If <co_label> is given then it should be a checkout label or label fragment (see "muddle help labels").

If a label or labels are given, then the repositories, and any upstream repositories, for those labels are reported. Otherwise, those repositories that have upstream repositories are reported.

With '-u' or '-url', print repository URLs. Otherwise, print the full spec of each Repository instance.

XXX Examples to be provided

allowed_switches = {'-u': 'url', '-url': 'url'}

cmd_name = 'query upstream-repos'

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**QueryVCS**
Bases: *muddled.commands.QueryCommand*

Syntax muddle query vcs

List the version control systems supported by this version of muddle, together with their VCS specifiers.

cmd_name = 'query vcs'

do_command()

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Rebuild**
Bases: *muddled.commands.PackageCommand*

Syntax muddle rebuild [<package> ...]

Rebuild packages. Just like build except that we clear any '/built' tags first (and their dependencies).

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/postinstalled". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

1. For each label, clear its '/built' tag, and then clear the tags for all the labels that depend on it. Note that this will include the same label with its '/installed' and '/postinstalled' tags.
2. For each label, build its '/postinstalled' tag (so essentially, do the equivalent of "muddle build").

build_these_labels(*builder*, *labels*)

cmd_name = 'rebuild'

class muddled.commands.**Reconfigure**
Bases: *muddled.commands.PackageCommand*

Syntax muddle reconfigure [<package> ...]

Reconfigure packages. Just like configure except that we clear any '/configured' tags first (and their dependencies).

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/configured". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

1. For each label, clear its '/configured' tag, and then clear the tags for all the labels that depend on it. Note that this will include the same label with its '/built', '/installed' and '/postinstalled' tags.
2. Do "muddle configure" for each label.

build_these_labels(*builder*, *labels*)

cmd_name = 'reconfigure'

required_tag = 'configured'

class muddled.commands.**Redeploy**
Bases: *muddled.commands.DeploymentCommand*

Syntax muddle redeploy [<deployment> ...]

Clean the named deployments (deleting their 'deploy/' directory), remove their '/deployed' tags, and then rebuild (deploy) them.

This is exactly equivalent to doing "muddle cleandeploy" for all the labels, followed by "muddle deploy" for them all.

<deployment> should be a label fragment specifying a deployment, or one of _all and friends, as for any deployment command. The <type> defaults to "deployment", and the deployment <tag> will be "/deployed". See "muddle help labels" for more information.

If no deployments are named, what we do depends on where we are in the build tree. See "muddle help labels".

build_these_labels(*builder*, *labels*)

cmd_name = 'redeploy'

class muddled.commands.**Reinstall**
Bases: *muddled.commands.PackageCommand*

Syntax muddle reinstall [<package> ...]

Reinstall packages (but don't rebuild them).

<package> should be a label fragment specifying a package, or one of _all and friends, as for any package command. The <type> defaults to "package", and the package <tag> will be "/postinstalled". See "muddle help labels" for more information.

If no packages are named, what we do depends on where we are in the build tree. See "muddle help labels".

1. For each label, clear its '/installed' tag, and then clear the tags for all the labels that depend on it. Note that this will include the same label with its '/postinstalled' tag.
2. For each label, build its '/postinstalled' tag (so essentially, do the equivalent of "muddle build").

build_these_labels(*builder*, *labels*)

cmd_name = 'reinstall'

class muddled.commands.**Release**
Bases: *muddled.commands.Command*

Produce a customer release from a release stamp file.

Syntax muddle release <release-file>

Or muddle release -test <release-file>

For example:

```
$ muddle release project99-1.2.3.release
```

This:

1. Checks the current directory is empty, and refuses to proceed if it is not.

   We always recommend doing `muddle init` or `muddle bootstrap` in an empty directory, but muddle insists that `muddle release` must be done in an empty directory.

2. Does `muddle unstamp <release-file>`,

3. Copies the release file to `.muddle/Release`.

   The existence of this file indicates that this is a release build tree, and "normal" muddle will refuse to build in it.

4. Copies the release specification to `.muddle/ReleaseSpec`.

5. Sets some extra environment variables, which can be used in the normal manner in muddle Makefiles:

   - `MUDDLE_RELEASE_NAME` is the release name, from the release file.
   - `MUDDLE_RELEASE_VERSION` is the release version, from the release file.
   - `MUDDLE_RELEASE_HASH` is the SHA1 hash of the release file

   "Normal" muddle will also create those environment variables, but they will be set to `(unset)`.

6. Does `mudddle build _release`.

   The meaning of "_release" is defined in the build description, using `builder.add_to_release_build()`. See:

   ```
   $ muddle doc mechanics.Builder.add_to_release_build
   ```

   for more information on that method, and "muddle query release" for the current setting.

   Note that, if youi have subdomains, only calls of `add_to_release_build()` in the top-level build description will be effective.

7. Creates the release directory, which will be called `<release-name>_<release-version>_<release-sha1>`. It copies the release file therein.

8. Calls the `release_from(builder, release_dir)` function in the build description, which is responsible for copying across whatever else needs to be put into the release directory.

   (Obviously it is an error if the build description does not have such a function.)

   Note that, if you have subdomains, only the `release_from()` function in the top-level build will be called.

9. Creates a compressed tarball of the release directory, using the compression mechanism specified in the release file. It will have the same basename as the release directory.

If the -test switch is given, then items 1..2 are not done. This allows testing a release build in the current build directory. The produce of such a test *must not* be treated as a proper release, as it has not involved a clean build of the build tree. Note that if you want to make your build tree back into a normal muddle build tree, then you will need to delete the .muddle/Release file yourself, by hand.

allowed_switches = {'-test': 'test'}

calc_tf_name(*release*, *release_dir*)
Work out the name and mode of the archive file we want to generate.

cmd_name = 'release'

do_release(*muddle_binary*, *current_dir*, *release_file*, *testing*)

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Reparent**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle reparent [-f[orce]] [<checkout> ...]

Re-associate the specified checkouts with their remote repositories.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/pulled". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

Some distributed VCSs (notably, Bazaar) can "forget" the remote repository for a checkout. In Bazaar, this typically means not remembering the "parent" repository, and thus not being able to pull. It appears to be possible to end up in this situation if network disconnection happens in an inopportune manner.

This command attempts to reassociate each checkout to the remote repository as named in the muddle build description. If '-force' is given, then this will be done even if the remote repository is already known, otherwise it will only be done if it is necessary.

For Bazaar: Reads and (maybe) edits .bzr/branch/branch.conf.

- If "parent_branch" is unset, sets it.
- With '-force', sets "parent_branch" regardless, and also unsets "push_branch".

allowed_in_release_build()

allowed_switches = {'-force': 'force', '-f': 'force'}

build_these_labels(*builder*, *labels*)

cmd_name = 'reparent'

required_tag = 'pulled'

class muddled.commands.**Retract**
Bases: *muddled.commands.AnyLabelCommand*

Syntax muddle retract <label> [<label> ...]

Retract the given labels and their consequents.

This unsets the tags specified in the given labels, and also the tags for all labels which each label depended on. For instance, if the label package:fred{x86}/built was given, then package:fred{x86}/configured would also be retracted, as /built (normally) depends on /configured for the same package.

This command is mostly for use by experts and scripts.

Each <label> is a label fragment, in the normal manner. The <type> defaults to "package:", and the <tag> defaults to the normal default <tag> for that type. Wildcards are expanded.

<label> may also be "_all", "_default_deployments", "_default_roles" or "_just_pulled".

See "muddle help labels" for more help on label fragments and the "_xxx" values.

build_these_labels(*builder*, *labels*)

cmd_name = 'retract'

class muddled.commands.**Retry**
Bases: *muddled.commands.AnyLabelCommand*

Syntax muddle retry <label> [<label> ...]

First this unsets the tags implied by the specified label(s), and only those tags. Then it rebuilds the labels.

Note that unsetting the tags *only* unsets exactly the tags named, and not any others.

This is sometimes useful when you're messing about with package rebuild rules.

Each <label> is a label fragment, in the normal manner. The <type> defaults to "package:", and the <tag> defaults to the normal default <tag> for that type. Wildcards are expanded.

<label> may also be "_all", "_default_deployments", "_default_roles" or "_just_pulled".

See "muddle help labels" for more help on label fragments and the "_xxx" values.

build_these_labels(*builder*, *labels*)

cmd_name = 'retry'

class muddled.commands.**RunIn**
Bases: *muddled.commands.Command*

Syntax muddle runin <label> <command> [...]

Run the command "<command> [...]" in the directory corresponding to every label matching <label>.

- Checkout labels are run in the directory corresponding to their checkout.
- Package labels are run in the directory corresponding to their object files.
- Deployment labels are run in the directory corresponding to their deployments.

We only ever run the command in any directory once.

<label> may be a label fragment, or one of the _xxx arguments If it is a label fragment, and the label type is not given, then "checkout:" is assumed. If it is an _xxx argument, then it may not be _all, since it would not be clear what type of label to expand it to). See "muddle help label _all" for more information on these values.

In practice, it is often simplest to use a shell script for <command>, rather than trying to work out the appropriate quoting rules for whatever command is actually wanted.

cmd_name = 'runin'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**StampDiff**
Bases: *muddled.commands.Command*

Syntax muddle stamp diff [<style>] <path1> <path2> [<output_file>]

Compare two builds, as version stamps.

Each of <path1> and <path2> may be an existing stamp file, or the top-level directory of a muddle build tree (i.e., the directory that contains the '.muddle' and 'src' directories).

If <output_file> is given, then the results of the comparison will be written to it, otherwise they will be written to standard output.

<style> specifies the way the comparison is done:

- -u, -unified - output a unified difference between stamp files.
- -c, -context - output a context difference between stamp files. This uses a "before/after" style of presentation.
- -n, -ndiff - use Python's "ndiff" to output the difference between stamp files. This is normally a more human-friendly set of differences, but outputs the parts of the files that match as well as those that do not.
- -h, -html - output the difference between stamp files as an HTML page, displaying the files in two columns, with differences highlighted by colour.
- -d, -direct - output the difference between two VersionStamp datastructures (this is the datastructure used to hold a stamp file internally within muddle). This is the default.

NOTE that at the moment '-d' only compares checkout information, not repository and domain information. It also ignores any "problems" in the stamp file.

For textual comparisons between stamp files, "muddle stamp diff" will first generate a temporary stamp file, if necessary (i.e., if <path1> or <path2> is a build tree), using the equivalent of "muddle stamp save".

For direct ('-d') comparison, a VersionStamp will be created from the build tree or read from the stamp file, as appropriate.

cmd_name = 'stamp diff'

compare_stamps(*muddle_binary*, *args*)

diff(*path1*, *path2*, *file1*, *file2*, *diff_style='unified'*, *fd=<open file '<stdout>', mode 'w'>*)
Output a comparison of two stamp files

diff_direct(*path1_is_build*, *path2_is_build*, *path1*, *path2*, *stamp1*, *stamp2*, *fd*)
Output comparison using VersionStamp instances.

Currently, only compares the checkouts.

XXX TODO It *should* compare everything (including any problems!)

print_syntax()

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**StampPull**
Bases: *muddled.commands.Command*

Syntax muddle stamp pull [<repository_url>]

This performs a VCS "pull" operation for the "versions/" directory. This assumes that the versions repository is defined in `.muddle/VersionsRepository`.

If a <repository_url> is given, then that is used as the remote repository for the pull, and also saved as the "current" remote repository in `.muddle/VersionsRepository`.

(If the VCS being used is Subversion, then <repository> is ignored by the actual "pull", but will still be used to update the VersionsRepository file. So be careful.)

If a <repository_url> is not given, then the repository URL named in `.muddle/VersionsRepository` is used. If there is no repository specified there, then the operation will fail.

See 'unstamp' for restoring from stamp files.

allowed_in_release_build()

cmd_name = 'stamp pull'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**StampPush**
Bases: *muddled.commands.Command*

Syntax muddle stamp push [<repository_url>]

This performs a VCS "push" operation for the "versions/" directory. This assumes that the versions repository is defined in `.muddle/VersionsRepository`.

If a <repository_url> is given, then that is used as the remote repository for the push, and also saved as the "current" remote repository in `.muddle/VersionsRepository`.

(If the VCS being used is Subversion, then <repository> is ignored by the actual "push", but will still be used to update the VersionsRepository file. So be careful.)

If a <repository_url> is not given, then the repository URL named in `.muddle/VersionsRepository` is used. If there is no repository specified there, then the operation will fail.

'stamp push' does not (re)create a stamp file in the "versions/" directory - use 'stamp version' to do that separately.

See 'unstamp' for restoring from stamp files.

allowed_in_release_build()

cmd_name = 'stamp push'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**StampRelease**
Bases: *muddled.commands.Command*

Syntax muddle stamp release [<switches>] <release-name> <release-version>

Syntax muddle stamp release [<switches>] <release-name> -next

Or muddle stamp release [<switches>] -template

This is similar to "stamp version", but saves a release stamp file - a stamp file that describes a release of the build tree.

The release stamp file written will be called:

```
versions/<release_name>_<release_version>.release
```

The "versions/" directory is at the build root (i.e., it is a sibling of the ".muddle/" and "src/" directories). If it does not exist, it will be created.

If the VersionsRepository is set (in the .muddle/ directory), and it is a distributed VCS (e.g., git or bzr) then `git init` (or `bzr init`, or the equivalent) will be done in the directory if necessary, and then the file will be added to the local working set in that directory. For subversion, the file adding will be done, but no attempt will be made to initialise the directory.

If the `-next` option is used, then the version number will be guessed. Muddle will look in the "versions/" directory for all the ".release" files whose names start with `<release_name>_v`, and will work out the last

version number (as <major>.minor>) present (not that 1.01 is the same as 1.1). It will then use 0.0 if it didn't find anything, or will use the next <minor> value. So if the user asked for `muddle stamp release Fred -next` and the files in "versions/" were:

```
Fred_v1.1.release
Fred_v3.02.release
Graham_v9.9.release
```

then the next version number would be guessed as v3.3.

If the `-template` option is used, then the file created will be called:

```
versions/this-is-not-a-file-name.release
```

and both the release name and release version values in the file will be set to `<REPLACE THIS>`. The user will have to rename the file, and edit both of those to sensible values, before using it (well, we don't enforce renaming the file, but...).

<switches> may be:

- -archive <name>

  This specifies how the release will be archived. At the moment the only permitted value is "tar".

- -compression <name>

  This specifies how the archive will be compressed. The default is "gzip", and at the moment the only other alternative is "bzip2".

See "muddle release" for using release files to build a release.

Note that release files are also valid stamp files, so "muddle unstamp" can be used to retore a build tree from them.

allowed_in_release_build()

cmd_name = 'stamp release'

guess_next_version_number(*version_dir*, *name*)
Return our best guess as to the next (minor) version number.

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**StampSave**
Bases: *muddled.commands.Command*

Syntax muddle stamp save [<switches>] [<filename>]

Go through each checkout, and save its remote repository and current brach/revision id/number to a file.

This is intended to be enough information to allow reconstruction of the entire build tree, as-is.

<switches> may be:

- -before <when> - use the (last) revision id at or before <when>
- -f, -force - "force" a revision id
- -h, -head - use HEAD for all checkouts
- -v <version>, -version <version> - specify the version of stamp file

These are explained more below. Switches may occur before or after <filename>.

If a <filename> is specified, then output will be written to a file called either <filename>.stamp or <filename>.partial. If <filename> already ended in '.stamp' or '.partial', then the old extension will be removed before deciding on whether to use '.stamp' or '.partial'.

If a <filename> is not specified, then a file called <sha1-hash>.stamp or <sha1-hash>.partial will be used, where <sha1-hash> is a hexstring representation of the hash of the content of the file.

The '.partial' extension will be used if it was not possible to write a full stamp file (revisions could not be determined for all checkouts, and neither '-force' nor '-head' was specified). An attempt will be made to give useful information about what the problems are.

If a file already exists with the name ultimately chosen, that file will be overwritten.

If '-before' is specified, then use the (last) revision id at or before that date and time. <when> is left a bit unspecified at the moment, and thus this feature is experimental.

> XXX At the moment '-before' is only supported for git and bzr, and
> XXX thus any form of date/time/revision id that git and/or will accept
> XXX may be used for <when>. The simple for "yyyy-mm-dd hh:mm:ss" seems
> XXX acceptabl to both.

For instance:

```
muddle stamp save -before "2012-06-26 23:00:00"
```

If '-f' or '-force' is specified, then attempt to "force" a revision id, even if it is not necessarily correct. For instance, if a local working directory contains uncommitted changes, then ignore this and use the revision id of the committed data. If it is actually impossible to determine a sensible revision id, then use the revision specified by the build description (which defaults to HEAD). For really serious problems, this may refuse to guess a revision id, in which case the 'stamp save' process should stop with the relevant checkout.

> (Typical use of '-f' is expected to be when a 'stamp save' reports problems in particular checkouts, but inspection shows that these are artefacts that may be ignored, such as an executable built in the source directory.)

Note that if '-before' is specified, '-force' will be ignored.

If '-h' or '-head' is specified, then HEAD will be used for all checkouts. In this case, the repository specified in the build description is used, and the revision id and status of each checkout is not checked.

By default, a version 2 stamp file will be created. This is equivalent to specifying '-version 2'. If '-version 1' is specified, then a version 1 stamp file will be created instead. This is the version of stamp file understood by muddle before it was able to create version 2 stamp files (see 'muddle help stamp save' to see if this is the case for a particular version of muddle or not). Note that the version 1 stamp file created by muddle 2.3 and above is not absolutely guaranteed to be correct.

See "muddle unstamp" for restoring from stamp files.

cmd_name = 'stamp save'

decide_stamp_filename(*hash*, *basename=None*, *partial=False*)

> Return filename, given a SHA1 hash hexstring, and maybe a basename.
>
> If 'partial', then the returned filename will have extension '.partial', otherwise '.stamp'.
>
> If the basename is not given, then the main part of the filename will be <hash>.
>
> If the basename is given, then if it ends with '.stamp' or '.partial' then that will be removed before it is used.

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**StampVersion**
Bases: *muddled.commands.Command*

Syntax muddle stamp version [-f[orce]|-v[ersion] <version>]

This is similar to "stamp save", but using a pre-determined stamp filename.

Specifically, the stamp file written will be called:

versions/<build_name>.stamp

or, if the build description has asked for checkouts to follow its branch with `builder.follow_build_desc_branch = True`:

```
versions/<build_name>.<branch_name>.stamp
```

The "versions/" directory is at the build root (i.e., it is a sibling of the ".muddle/" and "src/" directories). If it does not exist, it will be created.

If the VersionsRepository is set (in the .muddle/ directory), and it is a distributed VCS (e.g., git or bzr) then `git init` (or `bzr init`, or the equivalent) will be done in the directory if necessary, and then the file will be added to the local working set in that directory. For subversion, the file adding will be done, but no attempt will be made to initialise the directory.

<build_name> is the name of this build, as specified by the build description (by setting `builder.build_name`). If the build description does not set the build name, then the name will be taken from the build description file name. You can use "muddle query name" to find the build name for a particular build.

If a full stamp file cannot be written (i.e., if the result would have extension ".partial"), then the version stamp file will not be written.

Note that '-f' is supported (although perhaps not recommended), but '-h' is not.

By default, a version 2 stamp file will be created. This is equivalent to specifying '-version 2'. If '-version 1' is specified, then a version 1 stamp file will be created instead. This is the version of stamp file understood by muddle before it was able to create version 2 stamp files (see 'muddle help stamp version' to see if this is the case for a particular version of muddle or not). Note that the version 1 stamp file created by muddle 2.3 and above is not absolutely guaranteed to be correct.

See "muddle unstamp" for restoring from stamp files.

allowed_in_release_build()

cmd_name = 'stamp version'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Status**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle status [-v] [-j] [-quick] [<checkout> ...]

Report on the status of checkouts that need attention.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/pulled". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

If '-v' is given, report each checkout label as it is checked (allowing a sense of progress if there are many bazaar checkouts, for instance).

Runs the equivalent of `git status` or `bzr status` on each repository, and tries to only report those which have significant status.

> Note: For subversion and bazaar, the (remote) repository is queried, which may be slow. For git, the HEAD of the remote may be queried.

Be aware that "muddle status" will report on the currently checked out checkouts. "muddle status _all" will (attempt to) report on *all* the checkouts described by the build, even if they have not yet been checked out. This will fail on the first checkout directory it can't "cd" into (i.e., the first checkout that isn't there yet).

This muddle command exits with status 0 if all checkouts appear alright, and with status 1 if no checkouts were specified, if an exception occurred, or if some checkouts need attention.

At the end, if any checkouts need attention, their names are reported. With '-j', print them all on one line, separated by spaces.

The '-quick' switch tells muddle not to make any (potentially slow) queries across the network, and is only supported for "git". It will only look at the local information it already has, which means that the information it can gives depends upon whatever was last fetched into the local repository. It can typically inform you if there are local updates to be pushed, but will not (cannot) warn you if there are commits to be pulled.

allowed_in_release_build()

allowed_switches = {'-quick': 'quick', '-v': 'verbose', '-j': 'join'}

build_these_labels(*builder*, *labels*)

cmd_name = 'status'

required_tag = 'checked_out'

class muddled.commands.**Subst**

Bases: *muddled.commands.Command*

> **Syntax** muddle subst <src_file> [<xml_file>] <output_file>

Reads in <src_file>, and replaces any strings of the form "${..}" with values from the XML file (if any) or from the environment.

For the examples, I'm assuming we're building a release build, using "muddle release", and thus the MUDDLE_RELEASE_xxx environment variables are set.

Unexpected section title.

```
Without an XML file
-------------------
```

So, in the two argument form, we might run:

```
$ muddle subst version.h.in version.h
```

on version.h.in:

```
#ifndef PROJECT99_VERSION_FILE
#define PROJECT99_VERSION_FILE
#define BUILD_VERSION "${MUDDLE_RELEASE_NAME}: $(MUDDLE_RELEASE_VERSION}"
#endif
```

to produce:

```
#ifndef PROJECT99_VERSION_FILE
#define PROJECT99_VERSION_FILE
#define BUILD_VERSION "simple: v1.0"
#endif
```

Unexpected section title.

```
With an XML file
----------------
```

In the three argument form, values will first be looked up in the XML file, and then, if they're not found, in the environment. So given values.xml:

```
<?xml version="1.0" ?>
<values>
    <version>Kynesim version 99</version>
    <more>
        <value1>This is value 1</value1>
        <value2>This is value 2</value2>
    </more>
</values>
```

and values.h.in:

```
#ifndef KYNESIM_VALUES
#define KYNESIM_VALUES
#define KYNESIM_VERSION "${/values/version}"
#define RELEASE_VERSION "Release version ${MUDDLE_RELEASE_VERSION}"
#endif
```

then running:

```
$ muddle subst values.h values.xml values.h.in
```

would give us values.h:

```
#ifndef KYNESIM_VALUES
#define KYNESIM_VALUES
#define KYNESIM_VERSION "Kynesim version 99"
#define RELEASE_VERSION "Release version v1.0"
#endif
```

XML queries are used in the "${..}" to extract particular values from the XML. These look a bit like XPath queries - "/elem/elem/elem...", so for instance:

```
${/values/more/value2}
```

would be replaced by:

```
This is value 2
```

You can escape a "${ .. }" by passing "$${ .. }", so:

```
$${/values/more/value1}
```

becomes:

```
${/values/more/value1}
```

Both ${/version} and ${"/version"} give the same result.

You can also nest evaluations. With the environment variable THING set to "/values/version", then:

```
${ ${THING} }
```

will evaluate to:

```
Kynesim version 99
```

You can call functions with "${fn: .. }". Parameters can be surrounded by matching double quotes - these will be stripped before the parameter is evaluated. The available functions are:

- "${fn:val(something)}"

  This expands to the value of 'something' as a query (either as an environment variable or XPath)

- "${fn:ifeq(something,b)c}"

  If ${something} evaluates to b, then this expands to c. Both b and c may contain "${..}" sequences.

  Note that 'something' is expanded without you needing to specify such, but b and c are not.

  It is allowed to do things like:

```
${fn:ifeq(/values/version,"Kynesim version 99")
    def missing_function(a):
        # 'Version ${/values/version} of the software does not provide
        # this function, so we do so here
        <implementation code>
 }
```

- "${fn:ifneq(something,b)c}"

  The same, but you get c if evaluating 'something' does not give b.

- "${fn:echo(a,b,c,...)}"

  Evaluates each parameter (a, b, c, ...) in turn. Spaces between parameters are ignored. So:

```
${fn:echo(a, " space ", ${/values/more/value1}}
```

  would give:

```
a space This is value 1
```

cmd_name = 'subst'

do_subst(*args*)

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Sync**

Bases: *muddled.commands.CheckoutCommand*

Syntax muddle sync [<checkout> ...]

Or muddle sync [-v[erbose]] [<checkout> ...]

Or muddle sync [-show] [<checkout> ...]

"Synchronise" each checkout onto the branch it should be on...

Less succinctly, for each checkout, do the first applicable of the following:

- If this is the top-level build description, then:
  - if it has "builder.follow_build_desc_branch = True", then nothing needs to be done, as we're already there.

– if it does not have "builder.follow_build_desc_branch = True", but a branch was specified for it (i.e., via "muddle init -branch"), then go to that branch.

– if it does not have "builder.follow_build_desc_branch = True", and no branch was specified (at "muddle init"), then go to "master".

- If the build description specifies a revision for this checkout, go to that revision.
- If the build description specifies a branch for this checkout, and the checkout VCS supports going to a specific branch, go to that branch
- If the build description specifies that this checkout should not follow the build description (both Subversion and Bazaar support the "no_follow" option), then go to "master".
- If the build description specifies that this checkout is shallow, then give up.
- If the checkout's VCS does not support lightweight branching, then give up (the following choices require this).
- If the build description has "builder.follow_build_desc_branch = True", then go to the same branch as the build description.
- Otherwise, go to "master".

With '-v' or '-verbose', report in detail on what the "sync" operation is doing, and why.

With '-show', report on its decision making process, but don't actually do anything.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/changes_committed". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

allowed_switches = {'-v': 'verbose', '-verbose': 'verbose', '-show': 'show'}

build_these_labels (*builder*, *labels*)

cmd_name = 'sync'

required_tag = 'changes_committed'

class muddled.commands.**UnCheckout**

Bases: *muddled.commands.CheckoutCommand*

Syntax muddle uncheckout [<checkout> ...]

Tell muddle that the given checkouts no longer exist in the 'src/' directory hierarchy, and will need to be checked out again before they can be used.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each label, unset the '/checked_out' tag for that label, and the tags of any labels that depend on it. Note that this will include all the tags for any packages that directly depend on this checkout. However, it will not perform any "clean" or "distclean" actions for those packages.

Note that muddle itself does not check whether the checkout directory has been deleted or not. Attempting to do "muddle checkout" for a checkout directory that (still) exists will generally fail.

build_these_labels (*builder*, *labels*)

cmd_name = 'uncheckout'

class muddled.commands.**UnStamp**

Bases: *muddled.commands.Command*

To create a build tree from a stamp file:

Syntax muddle unstamp <file>

Or muddle unstamp <url>

Or muddle unstamp <vcs>+<url>

Or muddle unstamp <vcs>+<repo_url> <version_desc>

To update a build tree from a stamp file:

Syntax muddle unstamp -u[pdate] <file>

Unexpected section title.

```
Creating a build tree from a stamp file
---------------------------------------
```

The normal "unstamp" command reads the contents of a "stamp" file, as produced by the "muddle stamp" command, and:

1. Retrieves each checkout mentioned
2. Reconstructs the corresponding muddle directory structure
3. Confirms that the muddle build description is compatible with the checkouts.

This form of the command cannot be used within an existing muddle build tree, as its intent is to create a new build tree.

The file may be specified as:

- The local path to a stamp file.

  For instance:

```
muddle stamp  thing.stamp
mkdir /tmp/thing
cp thing.stamp /tmp/thing
cd /tmp/thing
muddle unstamp  thing.stamp
```

- The URL for a stamp file. In this case, the file will first be copied to the current directory.

  For instance:

```
muddle unstamp  http://some.url/some/path/thing.stamp
```

  which would first copy "thing.stamp" to the current directory, and then use it. If the file already exists, it will be overwritten.

- The "revision control specific" URL for a stamp file. This names the VCS to use as part of the URL - for instance:

```
muddle unstamp  bzr+ssh://kynesim.co.uk/repo/thing.stamp
```

  This also copies the stamp file to the current directory before using it. Note that not all VCS mechanisms support this (at time of writing, muddle's git support does not). If the file already exists, it will be overwritten.

- The "revision control specific" URL for a repository, and the path to the version stamp file therein.

  For instance:

```
muddle unstamp  bzr+ssh://kynesim.co.uk/repo  versions/ProjectThing.stamp
```

  This is intended to act somewhat similarly to "muddle init", in that it will checkout:

```
bzr+ssh://kynesim.co.uk/repo/versions
```

  and then unstamp the ProjectThing.stamp file therein.

Unexpected section title.

```
Updating a build tree from a stamp file
---------------------------------------
```

The "-update" form ("unstamp -update" or "unstamp -u") also reads the contents of a "stamp" file, but it then tries to amend the current build tree to match the stamp file.

This form of the command must be used within an existing muddle build tree, as its intent is to alter it.

The stamp file must be specified as a local path - the URL forms are not supported.

The command looks up each checkout described in the stamp file. If it already exists, then it sets it to the correct revision, using "muddle pull". This last means that the value "_just_pulled" will be set to those checkouts which have been pulled, so one can do, for instance, "muddle distrebuild _just_pulled".

> XXX Future versions of this command will also be able to change the branch of a checkout. This is not yet supported.

If the checkout does not exist, then it will be cloned, using "muddle checkout". Newly cloned checkouts will not be represented in "_just_pulled".

In the simplest case, the "unstamp -update" operation may just involve choosing different revisions on some checkouts.

Before using this form of the command, it is probably worth using:

```
muddle stamp diff . <file>
```

to determine what changes will be made.

After using this form of the command, it is highly recommended to use:

```
muddle veryclean
```

to delete the directories built from the checkout sources.

allowed_in_release_build()

allowed_switches = {'-u': 'update', '-update': 'update'}

check_build(*current_dir*, *checkouts*, *muddle_binary*)
Check that the build tree we now have on disk looks a bit like what we want...

cmd_name = 'unstamp'

print_syntax()

requires_build_tree()

restore_stamp(*builder*, *current_dir*, *domains*, *checkouts*)
Given the information from our stamp file, restore things.

unstamp_from_file(*muddle_binary*, *current_dir*, *thing*)
Unstamp from a file (local, over the network, or from a repository)

unstamp_from_repo(*muddle_binary*, *current_dir*, *repo*, *version_path*)
Unstamp from a repository and version path.

unstamp_from_stamp(*muddle_binary*, *current_dir*, *stamp*, *versions_repo=None*)
Given a stamp file, do our work.

update_from_file(*builder*, *filename*)
Update our build from the given stamp file.

update_from_stamp(*builder*, *domains*, *checkouts*)
Given the information from our stamp file, update the current build.

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

class muddled.commands.**Unimport**
Bases: *muddled.commands.CheckoutCommand*

Syntax muddle unimport [<checkout> ...]

Assert that the given checkouts haven't been checked out and must therefore be checked out.

<checkout> should be a label fragment specifying a checkout, or one of _all and friends, as for any checkout command. The <type> defaults to "checkout", and the checkout <tag> will be "/checked_out". See "muddle help labels" for more information.

If no checkouts are named, what we do depends on where we are in the build tree. See "muddle help labels".

For each label, unset the '/checked_out' tag for that label.

This command does not do anything to labels that depend on the given labels - if you want that, see "muddle removed".

Note that muddle itself does not check whether the checkout directory has been deleted or not. Attempting to do "muddle checkout" for a checkout directory that (still) exists will generally fail.

build_these_labels(*builder*, *labels*)

cmd_name = 'unimport'

class muddled.commands.**UpstreamCommand**
Bases: *muddled.commands.CheckoutCommand*

The parent class for the push/pull-upstream commands.

allowed_switches = {}

build_these_labels(*builder*, *labels*, *upstream_names*, *no_op*)

direction = 'to'

do_our_verb(*builder*, *co_label*, *vcs_handler*, *upstream*, *repo*)
Each subclass needs to implement this.

handle_label(*builder*, *co_label*, *upstream_name*, *repo*)

required_tag = 'checked_out'

verb = 'push'

verbing = 'Pushing'

with_build_tree(*builder*, *current_dir*, *args*)
Our command line is somewhat differently shaped.

So we have to handle it ourselves.

class muddled.commands.**VeryClean**
Bases: *muddled.commands.Command*

Syntax muddle veryclean

Sets the muddle build tree back to just checked out sources.

1. Delete the obj/, install/ and deploy/ directories.
2. Removes all the package tags, so muddle thinks that packages have not had anything done to them.
3. Removes all the deployment tags, so muddle thinks that all deployments have not been deployed.

For a build tree without subdomains, this is equivalent to:

```
$ rm -rf obj install deploy
$ rm -rf .muddle/tags/package
$ rm -rf .muddle/tags/deployment
```

It's a bit more complicated if there are any subdomains. If there is a 'domains/' directory, then this command will recurse down into it and perform the same operation for each subdomain it finds. This does not depend on whether the subdomain is defined in the build description - it is done purely on the basis of what directories are actually present.

As usual, 'muddle -n veryclean' will report on what it would do, without actually doing it.

Using this helps prevent unwanted built/installed software "building up" in the obj and install (and, to a lesser extent, deploy) infrastructure. Note that it does *not* remove checkouts that are no longer in use, nor can it do anything about any build artefacts inside checkout directories.

cmd_name = 'veryclean'

requires_build_tree()

with_build_tree(*builder*, *current_dir*, *args*)

class muddled.commands.**Whereami**
Bases: *muddled.commands.Command*

Syntax muddle where [-detail]

Or muddle whereami [-detail]

Looks at the current directory and tries to identify where it is within the enclosing muddle build tree. If it can calculate a label corresponding to the location, it will also report that (as <name> and, if appropriate, <role>).

For instance:

```
$ muddle where
Root of the build tree
$ cd src; muddle where
Checkout directory
$ cd main_co; muddle where
Checkout directory for checkout:main_co/*
$ cd ../../obj/main_pkg; muddle where
Package object directory for package:main_pkg{*}/*
```

If you're not in a muddle build tree, it will say so:

```
You are here. Here is not in a muddle build tree.
```

If the '-detail' switch is given, output suitable for parsing is output, in the form:

<what> <label> <domain>

i.e., a space-separated triple of items that don't themselves contain whitespace. For instance:

```
$ muddle where
Checkout directory for checkout:screen-4.0.3/*
$ muddle where -detail
Checkout checkout:screen-4.0.3/* None
```

cmd_name = 'where'

requires_build_tree()

want_detail(*args*)

with_build_tree(*builder*, *current_dir*, *args*)

without_build_tree(*muddle_binary*, *current_dir*, *args*)

muddled.commands.**build_a_kill_b**(*builder*, *labels*, *build_this*, *kill_this*)
For every label in labels, build the label formed by replacing tag in label with build_this and then kill the tag in label with kill_this.

We have to interleave these operations so an error doesn't lead to too much or too little of a kill.

muddled.commands.**build_labels**(*builder*, *to_build*)

muddled.commands.**command**(*command_name*, *category*, *aliases=None*)
A simple decorator to remmember a class by its command name.

'category' indicates which type of command this is

muddled.commands.**in_category**(*command_name*, *category*)

muddled.commands.**kill_labels**(*builder*, *to_kill*)

muddled.commands.**subcommand**(*main_command*, *sub_command*, *category*, *aliases=None*)
Remember the class for <main_command> <subcommand>.

17.2.3 muddled.cpiofile

Note: *Provides CPIO file support*

Utilities to write cpio archives.

There is apparently no standard way to do this from python. Ugh.

class muddled.cpiofile.**Archive**
Bases: object

Represents a CPIO archive.

Files are represented by their header information and the name of a file in the filesystem where they live. When render() is called, the appropriate data is written out to disc.

add_file(*a_file*)
DANGER WILL ROBINSON! You need to add files in the right order here or cpio will get very confused.

Todo

Reorder files in render() so that we get them in the right order, and remember to create intermediate directories.

add_files(*in_files*)

render(*to_file*, *logProgress=False*)
Render a CPIO archive to the given file.

class muddled.cpiofile.**CpioFileDataProvider**(*hierarchy*)
Bases: *muddled.filespec.FileSpecDataProvider*

Given a file map like that returned from files_from_fs() and a root name, create a filespec data provider.

name is the root of the effective hierarchy.

abs_match(*filespec*, *vroot=None*)
Return a list of the file object for each file that matches filespec.

list_files_under(*dir*, *recursively=False*, *vroot=None*)
Return a list of the files under dir.

class muddled.cpiofile.**File**
Bases: object

Represents a file in a CPIO archive.

Create a new, empty file. Some more or less sensible defaults are set up so that if you do try to synthesise a cpio archive from a default-constructed File you don't get utter rubbish. No guarantees you get a valid archive either though ..

- key_name is the name of the key under which this file is stored in the parent hierarchy. It's a complete hack, but essential for finding a file in the key map quickly without which deletion becomes an O(n^2) operation (and N = number of files in the root fs, so it's quite high).
- self.name - is the name of the file in the target archive.
- self.fs_name - is the name of the file in the underlying filesystem.
- self.orig_file - is the name of the file from which the data in this file object comes.

S_BLK = 24576

S_CHAR = 8192

S_DIR = 16384

S_LINK = 40960

S_REG = 32768

S_SGID = 1024

S_SOCKET = 49152

S_STICKY = 512

S_SUID = 2048

as_str(*fs_relative=None*)
Report ourself, but present the file system name relative to 'fs_relative'

delete_child_with_name(*in_name*)

rename(*name*)

set_contents(*data*)

set_contents_from_file(*file_name*)

class muddled.cpiofile.**Hierarchy**(*map*, *roots*)
Bases: object

•self.map - maps names in the target archive to file objects.

•self.roots - is a subset of self.map that just maps the root objects.

as_str(*fs_relative=None*)
Report ourself, but present the file system names relative to 'fs_relative'

erase_target(*file_name*)
Recursively remove file_name and all its descendants from the hierarchy.

merge(*other*)
Merge other with self.

We need to keep the hierarchy sensibly updated.

We merge the maps. We then kill all children and iterate over everything in the resulting map, finding a parent to add it to. This fails to preserve the order of files in directories, but the result is correct in every other way.

Anything in the result that does not have a parent is a root.

normalise()
Normalise the hierarchy into one with a single root.

We do this by taking each root in turn and removing a component, creating a directory in the process. If the resulting root is in roots, we add it to the children of that root and eliminate it from the map.

Iterate until there is only one root left.

parent_from_key(*key_name*)

put_target_file(*name*, *obj*)
Put a file into the archive. The directory for it must already exist.

•name - The name of the file in the target archive.

•obj - The file object to insert.

render(*to_file*, *logProgress=False*)

muddled.cpiofile.**file_for_dir**(*name*)
Create a vague attempt at a directory entry.

muddled.cpiofile.**file_from_data**(*name*, *data*)
Creates a File object from some explicit data you give it.

muddled.cpiofile.**file_from_fs**(*orig_file*, *new_name=None*)
Create a file object from a file on disc. You'll want to rename() it, unless you meant the file in the CPIO archive to have the same name as the one you passed in.

muddled.cpiofile.**hierarchy_from_fs**(*name*, *base_name*)
Create a hierarchy of files from a named object in the filesystem.

The files will be named with 'base_name' substituted for 'name'.

Returns a Hierarchy with everything filled in.

muddled.cpiofile.**merge_maps**(*dest*, *src*)
Merge src into dest. This needs special handling because we need to keep the hierarchy intact

We merge dest and src and then just rebuild the entire hierarchy - it's the easiest way, frankly.

For everything in the merged list, zap its children.

Now iterate over everything, os.path.split() it to find its parent and add it to its parents' child list.

If its parent doesn't exist, it's a root - mark and ignore it.

muddled.cpiofile.**trace_files**(*file_list*, *root*)
Given a File, add it and all its children, top-down, into file_list. Used as a utility routine by Hierarchy.

17.2.4 muddled.db

Note: *Handles the content of the .muddle directory*

Contains code which maintains the muddle database, held in root/.muddle

class muddled.db.**CheckoutData**(*vcs_handler*, *repo*, *co_dir*, *co_leaf*)
Bases: object

•location - The directory the checkout is in, relative to the root of the build tree. For instance:

```
src/builds
domains/subdomain1/src/first_co
```

•dir and leaf - The same information, as it was originally specified in the build description. This is primarily of use in version stamping. The dir may be None, and the leaf defaults to the checkout labels name.

We expect to take the repository described in 'repo' and check it out into:

–src/<co_leaf> or

–src/<co_dir>/<co_leaf>

depending on whether <co_dir> is None.

•repo - A Repository instance, representing where the checkout is checked out from. For example (eliding the actual URL):

```
Repository('git', 'http://.../main', 'builds')
Repository('git', 'http://.../subdomain1', 'first_co')
```

This the Repository as defined in the build description.

•vcs_handler - A VCS handler, which knows how to do version control operations for this checkout.

•options - Any specialised optons needed by the VCS handler. At the moment, the only option available is whether a git checkout is shallow or not. This is a dictionary.

move_to_subdomain(*other_domain_name*)

set_option(*name*, *value*)
Add/replace the named VCS option 'name'.

For reasons mostly to do with how stamping/unstamping works, we require option values to be either boolean, integer or string.

Also, only those option names that are explicitly allowed for a particular VCS may be used.

class muddled.db.**Database**(*root_path*)

Bases: object

Represents the muddle database

Historically, this class represented the muddle database as stored in the .muddle directory (on disk). Since we expect the user (and code) to edit these files frequently, we deliberately do not cache their values (other than, well, as themselves in the .muddle directory).

Since then however, we have also gained some dictionaries linking checkout labels to particular quantities.

It's useful to have a single place for most such dictionaries because when we do subdomain manipulation (i.e., taking a build description and including its build tree into another as a subdomain) we need to change all the labels in the new subdomain to reflect that fact. The fewer places we have to worry about that, the better.

So, we remember:

- root_path - The path to the root of the build tree.

Various PathFile instances:

- RootRepository_pathfile - for the '.muddle/RootRepository' file
- Description_pathfile - for the '.muddle/Description' file
- VersionsRepository_pathfile - for the '.muddle/VersionsRepository' file
- DescriptionBranch_pathfile - for the '.muddle/DescriptionBranch' file

which describe what the user requested via the original "muddle init".

and:

- local_labels - Transient labels which are "asserted", via 'set_tag()', and queried via 'is_tag()'. This functionality is used inside the Builder's "build_label()" mechanism, and is only intended for use within muddle itself.

Also, a variety of dictionaries that take (mostly) checkout labels as keys. Note that:

1. All the keys are "normalised" to have an unset label tag.
2. Thus it is assumed that the dictionaries will only be accessed via the methods supplied for this purpose.
3. The existence of an entry does not necessarily imply that the particular checkout is actually used, as the need for it may have gone away during a `builder.unify()` operation.

and, perhaps most importantly, the user should treat all of these as READ ONLY, since muddle itself maintains their content.

The dictionaries we use are:

- checkout_data - This maps checkout labels to the information we need to do checkout actions.
- checkout_vcs - This is a cache remembering the VCS for a checkout. It is not intended for direct access.
- checkout_licenses - This maps a checkout label to a License instance, representing the source code license under which this checkout's source code is being used. For instance:

```
checkout:builds/*               -> License('MPL 1.1', 'open-source')
checkout:(subdomain1)first_co/* -> License('LGPL v3', 'gpl')
```

In the case of a checkout that has multiple licenses, the license that is being "used" should be indicated.

Note that not all checkouts will necessarily have licenses associated with them.

- checkout_license_files - Some licenses require distribution of a license file from within the checkout, even in binary distributions. In such cases, this dictionary maps the checkout label to the name of the license file, relative to the checkout directory.

- license_not_affected_by is a dictionary of the form:

  { package_label : set(gpl_checkout_labels) }

  Each gpl_checkout_label is a label whose license (typically GPL) might be expected to "propagate" to any package built against that checkout. This dictionary is used to tell the system that, whilst 'package_label' may depend upon any of the 'gpl_checkout_labels', it doesn't build against them in a way that actually causes that propagation.

  So, for instance, if we have LGPL checkout label which our package links to as a dynamic library, we'd want to tell muddle that the package *depends* on the checkout, but doesn't get affected by the GPL nature of its license.

  This sort of thing is necessary because muddle itself has no way of telling.

  Note that ALL labels in this dictionary and its constituent sets should have their tags set to '*', so it is expected that this dictionary will be accessed using set_license_not_affected_by() and get_license_not_affected_by().

- nothing_builds_against is a set of checkout labels, each of which is a checkout that (presumably) has a GPL license, but against which no package links in a way that will cause GPL license "propagation".

- upstream_repositories is a dictionary of the form:

  { repo : { repo, set(names) } }

  That is, the key is a Repository instance (normally expected to be the same as one of the values in the checkout_repos dictionary), and the value is a dictionary whose keys are other repositories ("upstream" repositories) and some names associated with them.

  The same names may be associated with more than one upstream repository. It is also conceivable that an upstream repository might also act as a key, if it in turn has upstream repositories (whether this is strictly necessary is unclear - XXX still to decide whether to support this).

  > XXX *Note that if a checkout is branched because it is following the build description branch, the checkout Repository in checkot_repositories will get a new .branch value, at which point it will not necessarily compare equal to the Repository we're using as a key. This uncertainty MAY be a problem, and it may be better if add_upstream_repo() takes a copy of the Repository object before using it as a key, to force the issue.*

- domain_build_desc_label is a dictionary of the form:

  { domain : build-desc-label }

  where domain is the domain from a label (so None or a suitable string), and build-desc-label is the (normalised) checkout label for the build description checkout for that domain.

  Clearly, there is always at least one entry, with key None, for the top-level build description.

Initialise a muddle database with the given root_path.

add_upstream_repo(*orig_repo*, *upstream_repo*, *names*)

Add an upstream repo to 'orig_repo'.

- 'orig_repo' is the original Repository that we are adding an upstream for.
- 'upstream_repo' is the upstream Repository. It is an error if that repository is already an upstream of 'orig_repo'.

- 'names' is a either a single string, or a sequence of strings, that can be used to select this (and possibly other) upstream repositories.

Upstream repository names must be formed of A-Z, a-z, 0-9 and underscore or hyphen.

build_desc_file_name()
Return the filename of the build description.

checkout_has_license(*checkout_label*)
Return True if the named checkout has a license registered

clear_all_instructions(*domain=None*)
Clear all instructions - essentially only ever called from the command line.

clear_tag(*label*)

commit()
Commit changes to the db back to disc.

Remember to call this function when anything of note happens - don't assume you aren't about to hit an exception.

db_file_name(*rel*)
The full path name of the given relative filename in the current build tree.

dump_checkout_licenses(*just_name=False*)
Report on the licenses associated with our checkouts.

If 'just_name' is true, then report the licenses name, otherwise report the full License definition.

dump_checkout_paths()

dump_checkout_repos(*just_url=False*)
Report on the repositories associated with our checkouts.

If 'just_url' is true, then report the repository URL, otherwise report the full Repository definition (which shows branch and revision as well).

dump_checkout_vcs()
Report on the version control systems associated with our checkouts, and any VCS options.

dump_domain_build_desc_labels()

dump_upstream_repos(*just_url=False*)
Report on the upstream repositories associated "default" repositories

If 'just_url' is true, then report the repository URL, otherwise report the full Repository definition (which shows branch and revision as well).

get_checkout_data(*checkout_label*)

get_checkout_dir_and_leaf(*checkout_label*)

get_checkout_license(*checkout_label*, *absent_is_None=False*)
Returns the License instance for this checkout label

If 'absent_is_None' is true, then if 'checkout_label' does not have an entry in the licenses dictionary, None will be returned. Otherwise, an appropriate GiveUp exception will be raised.

get_checkout_license_file(*checkout_label*, *absent_is_None=False*)
Returns the License file for this checkout label

If 'absent_is_None' is true, then if 'checkout_label' does not have an entry in the license files dictionary, None will be returned. Otherwise, an appropriate GiveUp exception will be raised.

get_checkout_location (*checkout_label*)
'checkout_label' is a "checkout:" Label, or None

If it is None, then "src" is returned.

Otherwise, the path to the checkout directory for this label, relative to the root of the build tree, is calculated and returned.

If you want the full path to the checkout directory, then use get_checkout_path().

get_checkout_path (*checkout_label*)
'checkout_label' is a "checkout:" Label, or None

If it is None, then "<root path>/src" is returned.

Otherwise, the path to the checkout directory for this label is calculated and returned.

If you want the path *relative* to the root of the build tree (i.e., a path starting "src/"), then use get_checkout_location().

get_checkout_repo (*checkout_label*)
Returns the Repository instance for this checkout label

get_checkout_vcs (*checkout_label*)
'checkout_label' is a "checkout:" Label.

Returns the VCS handler for the given checkout.

Raises GiveUp (containing an explanatory message) if we cannot find that checkout label.

get_checkout_vcs_options (*checkout_label*)
'checkout_label' is a "checkout:" Label.

Returns the options for the given checkout, as a (possibly empty) dictionary.

Since most checkouts will not have options, and will thus have no entry for such, cannot return an error if there is no such checkout in the build.

get_domain_build_desc_label (*domain*)
'domain' is a domain as taken from a label, so None or a string

If it is None, then the checkout label for the top-level build description is returned.

Otherwise, the checkout label for the build description for that domain is returned.

In either case, the checkout label will be normalised (so its tag will be '*')

Raises GiveUp with an appropriate message if 'domain' is not recognised.

get_license_not_affected_by (*this_label*)
Find what is registered as not affecting this label's license

That is, the things on which this package depends, that appear to be GPL and propagate, but against which we have been told we do not actually build, so the license is not, in fact, propagated.

Returns a (possibly empty) set of checkout labels, each with tag '*'.

get_nothing_builds_against (*co_label*)
Return True if this label is in the "not linked against" set.

get_subdomain_info (*domain_name*)
Return the root repository and build description for a subdomain.

Reads the RootRepository and Description files in the (sub)domain's ".muddle" directory.

get_upstream_repos(*orig_repo*, *names=None*)
Retrieve the upstream repositories for 'orig_repo'

If 'names' is given, it must be a sequence of strings, in which case only those upstream repositories annotated with any of the names will be returned.

Returns a list of tuples of the form:

(upstream repositories, matching names)

This will be empty if there are no upstream repositories for 'orig_repo', or none with any of the names in 'names' (if given).

In the case of 'names' being empty, 'matching names' will contain the names registered for that upstream repository.

NB: 'matching names' is a tuple with the names sorted, and the list returned is also sorted.

include_domain(*other_builder*, *other_domain_name*)
Include data from other_builder, built in other_domain_name

This method is the main reason why this class gets to hold so much information - it gives us a single place to concentrate much of the knowledge about including subdomains.

Note we rely upon all the labels in the other domain already having been altered to reflect their subdomainness

This should only be called by muddle itself.

instruction_file_dir(*domain=None*)
Return the name of the directory in which we keep the instruction files

instruction_file_name(*label*)
If this label were to be associated with a database file containing the (absolute) filename of an instruction file to use for this package and role, what would it be?

is_tag(*label*)
Is this label asserted?

print_upstream_repo_info(*orig_repo*, *co_labels*, *just_url*)
Print upstream repository information.

'orig_repo' is the "main" repository, the one that is not upstream

'co_labels' is a sequence of 0 or more checkout labels, which are associated with that repository.

If 'just_url' is true, then report the repository URL, otherwise report the full Repository definition (which shows branch and revision as well).

scan_instructions(*lbl*)
Returns a list of pairs (label, filename) indicating the list of instruction files matching lbl. It's up to you to load and sort them (but load_instructions() will help with that).

set_checkout_data(*checkout_label*, *co_data*)

set_checkout_license(*checkout_label*, *license*)

set_checkout_license_file(*checkout_label*, *license_file*)
Set the license file for this checkout.

set_checkout_vcs_option(*checkout_label*, *option_name*, *option_value*)
'checkout_label' is a "checkout:" Label.

set_domain_build_desc_label(*checkout_label*)
This should only be called by muddle itself.

set_domain_marker(*domain_name*)
Mark this as a (sub)domain

In a (sub)domain, we have a file called `.muddle/am_subdomain`, which acts as a useful flag that we *are* a (sub)domain.

set_instructions(*label*, *instr_file*)
Set the name of a file containing instructions for the deployment mechanism.

- label -
- instr_file - The InstructionFile object to set.

If instr_file is None, we unset the instructions.

set_license_not_affected_by(*this_label*, *co_label*)
Asserts that the license for 'co_label' does not affect 'pkg_label'

We assume that:

1. 'this_label' is a package that depends (perhaps indirectly) on 'co_label', or is a checkout directly required by such a package.
2. 'co_label' is a checkout with a "propagating" license (i.e., some form of GPL license).
3. Thus by default the "GPL"ness would propagate from 'co_label' to 'this_label' (and, if it is a package, to the checkouts it is (directly) built from).

However, this function asserts that, in fact, this is not true. Our checkout is (or our checkouts are) not built in such a way as to cause the license for 'co_label' to propagate.

Or, putting it another way, for a normal GPL license, we're not linking with anything from 'co_label', or using its header files, or copying GPL'ed files from it, and so on.

If 'co_label' is under LGPL, then that would reduce to saying we're not static linking against 'co_label' (or anything else not allowed by the LGPL).

Note that we may be called before 'co_label' has registered its license, so we cannot actually check that 'co_label' has a propagating license (or, indeed, that it exists or is depended upon by 'pkg_label').

set_nothing_builds_against(*co_label*)
Indicate that no-one links against this checkout.

...or, at least, not in a way to cause GPL license "propagation".

set_tag(*label*)
Assert this label.

setup(*repo_location*, *build_desc*, *versions_repo=None*, *branch=None*)
Set values for the files in .muddle that describe our intial state.

- 'repo_location' is written to .muddle/RootRepository
- 'build_desc' is written to .muddle/Description
- If 'versions_repo' is not None, it is written to .muddle/VersionsRepository. Note that "not None" means that a value of '' *will* be written to the file.

  If 'versions_repo' is None, and 'repo_location' is not a centralised VCS (i.e., subversion), then it will be written to .muddle/VersionsRepository instead.
- If 'branch' is not None, then it will be written to .muddle/DescriptionBranch

This method should only be called by muddle itself.

tag_file_name(*label*)
If this file exists, the given label is asserted.

To make life a bit easier, we group labels.

upstream_name_re = <_sre.SRE_Pattern object>

class muddled.db.**Instruction**
Bases: object

Something stored in an InstructionFile.

Subtypes of this type are mainly defined in the instr.py module.

clone_from_xml(*xmlNode*)
Given an XML node, create a clone of yourself, initialised from that XML or raise an error.

equal(*other*)
Return True iff self and other represent the same instruction.

Not __eq__() because we want the python identity to be object identity as always.

outer_elem_name()
What's the outer element name for this instructiont type?

to_xml(*doc*)
Given an XML document, return a node which represents this instruction

class muddled.db.**InstructionFactory**
Bases: object

An instruction factory.

from_xml(*xmlNode*)
Given an xmlNode, manufacture an Instruction from it or return None if none could be built

class muddled.db.**InstructionFile**(*file_name*, *factory*)
Bases: object

An XML file containing a sequence of instructions for deployments. Each instruction is a subtype of Instruction.

file_name Where this file is stored values A list of instructions. Note that instructions are ordered.

add(*instr*)
Add an instruction.

clear()

commit(*file_name*)
Commit an instruction list file back to disc.

equal(*other*)
Return True iff self and other represent the same set of instructions. False if they don't.

get()
Retrieve the value of this instruction file.

get_xml()
Return an XML representation of this set of instructions as a string.

read()
Read our instructions from disc. The XML file in question looks like:

```
<?xml version="1.0"?>
<instructions priority=100>
 <instr-name>
   <stuff .. />
 </instr-name>
</instructions>
```

The priority is used by deployments when deciding in what order to apply instructions. Higher priorities get applied last (which is the logical way around, if you think about it).

save_as(*file_name*)

class muddled.db.**JustPulledFile**(*file_name*)
Bases: object

Our memory of the checkouts that have just been pulled.

Set the path to the _just_pulled file.

add(*label*)
Add the label to our local memory.

The label is not added to the _just_pulled file until commit() is called.

clear()
Clear the contents of the _just_pulled file, and our local memory.

If the _just_pulled files does not exist, does nothing

commit()
Commit our local memory to the _just_pulled file.

The labels are sorted before being written to the file.

Leaves the local memory intact after writing (it does not clear it).

get_from_disk()
Retrieve the contents of the _just_pulled file as a list of labels.

First clears the local memory, then reads the labels in the _just_pulled file into local memory, then returns that set as a sorted list.

is_pulled(*label*)

class muddled.db.**PathFile**(*file_name*)
Bases: object

Manipulates a file containing a single path name.

Create a PathFile object with the given filename.

commit()
Write the value of the PathFile to disc.

from_disc()
Retrieve the current value of the PathFile, directly from disc.

Returns None if there is a problem reading the PathFile.

Caches the value if there was one.

get()
Retrieve the current value of the PathFile, or None if there isn't one.

Uses the cached value if that is believed valid.

get_if_it_exists()
Retrieve the current value of the PathFile, if it exists (on disk).

This variant does not try to cache the value.

set(*val*)
Set the value of the PathFile (possibly to None).

class muddled.db.**TagFile**(*file_name*)
Bases: object

An XML file containing a set of tags (statements).

clear(*tag_value*)
Clear the relevant tag value.

commit()
Commit an XML tagfile back to a file.

erase()
Erase this tag file.

get()
Retrieve the value of this tagfile.

read()
Read data in from the disc.

The XML file in question looks a bit like:

```
<?xml version="1.0"?>
<tags>
  <X />
  <Y />
</tags>
```

set(*tag_value*)
Set the relevant tag value.

muddled.db.**load_instruction_helper**(*x*, *y*)
Given two triples (l,f,i), compare i.prio followed by f.

muddled.db.**load_instructions**(*in_instructions*, *a_factory*)
Given a list of pairs (label, filename) and a factory, load each instruction file, sort the result by priority and filename (the filename just to ensure that the sort is stable across fs operations), and return a list of triples (label, filename, instructionfile).

- in_instructions -
- a_factory - An instruction factory - typically instr.factory.

Returns a list of triples (label, filename, instructionfile object)

17.2.5 muddled.depend

Note: *The core of the dependency system, and Labels*

Dependency sets and dependency management

class muddled.depend.**Action**
Bases: object

Represents an object you can call to "build" a tag.

build_label(*builder*, *label*)
Build the given label. Your dependencies have been satisfied.

•in_deps - Is the set whose dependencies have been satisified.

Returns True on success, False or throw otherwise.

class muddled.depend.**Label**(*type*, *name*, *role=None*, *tag='*'*, *transient=False*, *system=False*, *domain=None*)
Bases: object

A label denotes an entity in muddle's dependency hierarchy.

A label is structured as:

```
<type>:<name>{<role>}/<tag>[<flags>]
```

or:

```
<type>:(<domain>)<name>{<role>}/<tag>[<flags>]
```

The <type>, <name>, <role> and <tag> parts are composed of the characters [A-Za-z0-9-+_], or the wildcard character '*'.

The <domain> name is composed of the same plus '(' and ')'.

The domain, role and flags are all optional.

Note: The label strings "type:name/tag" and "type:name{}/tag[]" are identical, although the former is the more usual form.)

The '+' is allowed in label parts to allow for names like "g++".

Domains are used when relating sub-builds to each other, and are not necessary when relating labels within the same build. It is not allowed to specify the empty domain as "()" - just omit the parentheses.

If necessary "sub-domains" are specified using nested domains – for instance:

```
(outer)
(outer(inner))
(outer(inner(even.innerer)))
```

This is intended to be unambiguous rather than pretty.

Note that wildcarding of a domain name currently only supports one level (i.e., the top "(*)"), and not wildcarding of nested domains.

If you do find yourself using multi-level domains, we would strongly suggest reconsidering your overall build design.

The "core" part of a label is the `<name>{<role>}` or `(<domain>)<name>{<role>}`. The <type> and <tag> can (typically) be thought of as tracking the progress of the "core" entity through its lifecycle (build sequence, etc.).

Names beginning with an underscore are reserved by muddle, so do not use them for other purposes.

(Why is the 'domain' argument at the end of the argument list? Because it was added after the other arguments were already well-established, and some uses of Label use positional arguments.)

Label instances are treated as immutable by the muddle system, although the implementation does not currently enforce this. Please don't try to abuse this, as Bad Things will happen.

Note: The *flags* on a label are not immutable, and are regarded as transient annotations.

Note: When a domain is included as a subdomain, all of its labels are "adjusted" to have the new, appropriate domain name. This is clearly a special meaning of the word "immutable". However, it should only be the muddle system itself doing this.

Because of this (potential change in content of a label), the domain name does not contribute to a label's hash value. Thus a label that whose domain name is changed will continue to work as the same key in a dictionary (for instance).

Type What kind of label this is. The standard muddle values are "checkout", "package" and "deployment". These values are defined programmatically via muddled.utils.LabelType. Thus the 'type' is conventionally used to indicate what general "stage" of the build process a label belongs to.

Name The name of this checkout/package/whatever. This should be a useful mnemonic for the labels purpose.

Role The role for this checkout/package/whatever. A role might delimit the target architecture of the labels it is used in (roles such as "x86", "arm", "beagleboard"), or the sort of purpose ("role" in the more traditionale sense, such as "boot", "firmware", "packages"), or some other useful delineation of a partition in the general label namespace (thinking of labels as points in an N-dimensional space).

Tag A tag indicating more precisely what stage the label belongs to within each 'type'. There are different conventional values according to the 'type' of the label (for instance, "checked_out", "built", "installed", etc.). These values are defined programmatically via muddled.utils.LabelTag.

Transient If true, changes to this tag will not be persistent in the muddle database. 'transient' is used to denote something which will go away when muddle is terminated - e.g. environment variables.

System If true, marks this label as a system label and not to be reported (by 'muddle depend') unless asked for. System labels are labels "invented" by muddle itself to satisfy implicit dependencies, or to allow the build system as a whole to work.

Domain The domain is used to specify which build or sub-build this label corresponds to. Nested "tail recursive" parenthesised components may be used to specify sub-domains (but this is not recommended). The domain defaults to the current build.

The role may be None, indicating (for instance) that roles are not relevant to this particular label.

The domain may be None, indicating that the label belongs to the current build. If the domain is given as '' (empty string) then this is equivalent to None, and is stored as such. Do not specify domains unless you need to.

The kind, name, role and tag may be wildcarded, by being set to '*'. When evaluating dependencies between labels, for instance, a wildcard indicates "for any value of this part of the label".

Domains can be wildcarded, and that probably means the obvious (that the label applies across all domains), but this may not yet be implemented. Wildcarding of sub-domains may never be supported.

Note that label flags (including specifically 'transient' and 'system') are not equality-preserving properties of a label - two labels are not made unequal just because they have different flags.

(In fact, no two labels should ever have different values for transience, for obvious reasons, and the system flag is intended only to limit over-reporting of information.)

For instance:

```
>>> Label('package', 'busybox')
Label('package', 'busybox', role=None, tag='*')
>>> str(_)
'package:busybox/*'
>>> Label('package', 'busybox', tag='installed')
Label('package', 'busybox', role=None, tag='installed')
>>> str(_)
'package:busybox/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed')
Label('package', 'busybox', role='rootfs', tag='installed')
>>> str(_)
'package:busybox{rootfs}/installed'
>>> Label('package', 'busybox', 'rootfs', 'installed')
Label('package', 'busybox', role='rootfs', tag='installed')
>>> str(_)
'package:busybox{rootfs}/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed', domain="arm.helloworld")
Label('package', 'busybox', role='rootfs', tag='installed', domain='arm.helloworld')
>>> str(_)
'package:(arm.helloworld)busybox{rootfs}/installed'
>>> Label('package', 'busybox', role='rootfs', tag='installed', domain="arm(helloworld)")
Label('package', 'busybox', role='rootfs', tag='installed', domain='arm(helloworld)')
>>> str(_)
'package:(arm(helloworld))busybox{rootfs}/installed'
```

FLAG_DOMAIN_SWEEP = 'D'

FLAG_SYSTEM = 'S'

FLAG_TRANSIENT = 'T'

copy()
Return a copy of this label.

copy_and_unify_with(*target*)
Return a copy of ourserlves, unified with the target.

All the non-wildcard parts of 'target' are copied, to overwrite the equivalent parts of the new label.

copy_with_domain(*new_domain*)
Return a copy of self, with the domain changed to new_domain.

Note that if 'new_domain' is given as "" (empty string) then that is treated as if it were given as None.

copy_with_role(*new_role*)
Return a copy of self, with the role changed to new_role.

copy_with_tag(*new_tag*, *system=None*, *transient=None*)
Return a copy of self, with the tag changed to new_tag.

domain

domain_part = '[()A-Za-z0-9._+-]+|*'

domain_part_re = <_sre.SRE_Pattern object>

fragment_re = <_sre.SRE_Pattern object at 0x20eddd0>

static from_fragment (*fragment*, *default_type*, *default_role=None*, *default_domain=None*)
Given a string containing a label fragment, return a Label.

The caller indicates the default type, role and domain.

The fragment must contain a <name>, but otherwise *may* contain any of:

- <type>: - if this is not given, the default is used
- (<domain>) - if this is not given, the default is used.
- {<role>} - if this is not given, the default is used.
- /<tag> - if this is not given, a tag appropriate to the <type> is chosen (checked_out, postinstalled or deployed)

Any of the default_xx values may be None.

static from_string (*label_string*)
Construct a Label from its string representation.

The string should be of the correct form:

- <type>:<name>/<tag>
- <type>:<name>{<role>}/<tag>
- <type>:<name>/<tag>[<flags>]
- <type>:<name>{<role>}/<tag>[<flags>]
- <type>:(<domain>)<name>/<tag>
- <type>:(<domain>)<name>{<role>}/<tag>
- <type>:(<domain>)<name>/<tag>[<flags>]
- <type>:(<domain>)<name>{<role>}/<tag>[<flags>]

See the docstring for Label itself for the meaning of the various parts of a label.

<flags> is a set of individual characters indicated as flags. There are two flags that will be recognised and used, 'T' for Transience and 'S' for System. Any other flag characters will be ignored.

If the label string is valid, a corresponding Label will be returned, otherwise a utils.GiveUp exception will be raised.

```
>>> Label.from_string('package:busybox/installed')
Label('package', 'busybox', role=None, tag='installed')
>>> Label.from_string('package:busybox{firmware}/installed[ABT]')
Label('package', 'busybox', role='firmware', tag='installed', transient=True)
>>> Label.from_string('package:(arm.hello)busybox{firmware}/installed[ABT]')
Label('package', 'busybox', role='firmware', tag='installed', transient=True, domain='arm.he
>>> Label.from_string('*:(*)*{*}/*')
Label('*', '*', role='*', tag='*', domain='*')
>>> Label.from_string('*:*{*}/*')
Label('*', '*', role='*', tag='*')
>>> Label.from_string('foo:bar{baz}/wombat[T]')
Label('foo', 'bar', role='baz', tag='wombat', transient=True)
>>> Label.from_string('foo:(ick)bar{baz}/wombat[T]')
Label('foo', 'bar', role='baz', tag='wombat', transient=True, domain='ick')
>>> Label.from_string('foo:(ick(ack))bar{baz}/wombat[T]')
Label('foo', 'bar', role='baz', tag='wombat', transient=True, domain='ick(ack)')
```

A tag must be supplied:

```
>>> Label.from_string('package:busybox')
Traceback (most recent call last):
...
GiveUp: Label string 'package:busybox' is not a valid Label
```

If you specify a domain, it may not be "empty":

```
>>> Label.from_string('package:()busybox/*')
Traceback (most recent call last):
...
GiveUp: Label string 'package:()busybox/*' is not a valid Label
```

is_definite()
Return True iff this label contains no wildcards

is_wildcard()
Return True iff this label contains at least one wildcard.

This is the dual of is_definite(), but is provided so whichever seems more appropriate to the task at hand can be chosen.

just_match(*other*)
Return True if the labels match, False if they do not

label_part = '[A-Za-z0-9._+-]+|*'

label_part_re = <_sre.SRE_Pattern object>

label_string_re = <_sre.SRE_Pattern object at 0x20ec900>

match(*other*)
Return an integer indicating the match specicifity - which we do by counting '*' s and subtracting from 0.

Returns the match specicifity, None if there wasn't one.

match_without_tag(*other*)
Returns True if other matches self without the tag, False otherwise

Specifically, tests whether the two Labels have identical type, domain, name and role.

middle()
Return the "middle" portion of our name, between type and tag.

That is, the domain, name and role (as appropriate).

For instance:

```
>>> checkout('fred').middle()
'fred'
>>> checkout('jim', domain='a(b)', tag='*').middle()
'(a(b))jim'
>>> package('fred', role='bob', domain='a').middle()
'(a)fred{bob}'
```

name

role

static split_domain(*value*)
Split a domain into its parts and check that it is valid.

Note that the 'value' should *not* include the outermost parentheses (see the examples below).

Raises a utils.GiveUp exception if it's Bad.

For instance:

```
>>> Label.split_domain('fred')
['fred']
>>> Label.split_domain('fred(jim)')
['fred', 'jim']
>>> Label.split_domain('fred(jim(bob))')
['fred', 'jim', 'bob']
>>> Label.split_domain('')
Traceback (most recent call last):
...
GiveUp: Label domain '()' is not allowed
>>> Label.split_domain('()')
Traceback (most recent call last):
...
GiveUp: Label domain '(())' starts with zero length domain, '()', i.e. '(('
>>> Label.split_domain('(')
Traceback (most recent call last):
...
GiveUp: Label domain part '(()' has unbalanced parentheses, '('
>>> Label.split_domain(')')
Traceback (most recent call last):
...
GiveUp: Label domain '())' has unbalanced parentheses, ')'
>>> Label.split_domain('fred(jim')
Traceback (most recent call last):
...
GiveUp: Label domain part '(fred(jim)' has unbalanced parentheses, 'fred(jim'
>>> Label.split_domain('fred((jim(bob)))')
Traceback (most recent call last):
...
GiveUp: Label domain '(fred((jim(bob))))' starts with zero length domain, '((jim(bob))', i.e
```

split_domains()
Returns a list of the domains for this Label, in order.

If there are no subdomains, then a zero length list is returned.

Raises a utils.GiveUp exception if the parentheses do not match up (the check is only fairly crude), or if there are two adjacent opening parentheses.

This is similar to utils.split_domain, but behaves somewhat differently.

tag

type

unifies(*other*)
Returns True if and only if every field in self is either equal to a field in other , or if other is a wildcard. Wildcards in self do not match anything but a wildcard in other.

class muddled.depend.**Rule**(*target_dep*, *action*)
Bases: object

A rule or "dependency set".

Every Rule has:

- a target Label (its desired result),
- an optional Action object (to do the work to produce that result),

- and a set of Labels on which the target depends (which must have been satisfied before this Rule can be triggered).

In other words, once all the dependency Labels are satisfied, the object can be called to 'build' the target Label.

(And if there is no object, the target is automatically satisfied.)

Note that the "set of Labels" is indeed a set, so adding the same Label more than once will not have any effect (caveat: adding a label with different flags from a previous label may have an effect, but it's not something that should be relied on).

Note: The actual "satisfying" of labels is done in muddled.mechanics. For instance, Builder.build_label() "builds" a label in the context of the rest of its environment, and uses 'action' to "build" the label.

- *target_dep* is the Label this Rule intends to "make".
- *action* is None or an Action, which will be used to "make" the *target_dep*.

add (*label*)
Add a dependency on the given Label.

catenate_and_merge (*other_rule*, *complainOnDuplicate=False*, *replaceOnDuplicate=True*)
Merge ourselves with the given rule.

If replaceOnDuplicate is true, other_rule get priority - this is the target for a unify() and makes the source build instructions go away.

depend_checkout (*co_name*, *tag*)
Add a dependency on label "checkout:<co_name>/<tag>".

depend_deploy (*dep_name*, *tag*)
Add a dependency on label "deployment:<dep_name>/tag".

depend_pkg (*pkg*, *role*, *tag*)
Add a dependency on label "package:<pkg>{<role>}/tag".

merge (*deps*)
Merge another Rule with this one.

Adds all the dependency labels from *deps* to this Rule.

If *deps.action* is not None, replaces our *action* with the one from *deps*.

replace_target (*new_t*)

to_string (*showSystem=True*, *showUser=True*)
Return a string representing this dependency set.

If *showSystem* is true, include dependency labels with the System tag (i.e., dependencies inserted by the muddle system itself), otherwise ignore such.

If *showUser* is true, include dependency labels without the System tag (i.e., "normal" dependencies, explicitly added by the user), otherwise ignore such.

The default is to show all of the dependencies.

For instance (not a very realistic example):

```
>>> tgt = Label.from_string('package:fred{jim}/*')
>>> r = Rule(tgt,None)
>>> r.to_string()
'package:fred{jim}/* <- [ ]'
```

```
>>> r.add(Label.from_string('package:bob{bob}/built'))
>>> r.depend_checkout('fred','jim')
>>> r.depend_pkg('albert','jim','built')
>>> r.depend_deploy('hall','deployed')
>>> r.to_string()
'package:fred{jim}/* <- [ checkout:fred/jim, deployment:hall/deployed, package:albert{jim}/b
```

The "<-" is to be read "depends on".

Note that the order of the dependencies in the output is sorted by label.

unify_dependencies (*source*, *target*)
Whenever source appears in our dependencies, replace it with source.unify(target)

class muddled.depend.**RuleSet**
Bases: object

A collection of rules that encapsulate how you can get from A to B.

Formally, this is just a mapping of labels to Rules. Duplicate targets are merged - it's assumed that the objects will be the same.

Informally, we cache some of the label look-ups for a major improvement in build time; the half billion lookups were taking over a hundred seconds to do!

CAVEAT: Be aware that new rules (when added) can be merged into existing rules. Since we don't *copy* rules when we add them, this could be a cause of unexpected side effects...

add (*rule*)
Add the Rule 'rule'.

Specifically, if we already have a rule for this rule's target label, merge the new rule into the old (see Rule.merge).

If this rule is for a new target, just remember it.

merge (*other_deps*)
Merge another RuleSet into this one.

Simply adds each rule from the other RuleSet to this one (see the 'RuleSet.add' method)

rule_for_target (*target*, *createIfNotPresent=False*)
Return the rule for this target - this contains all the labels that need to be asserted in order to build the target.

If createIfNotPresent is true, and there is no rule for this target, then we will create (and add to our internal map) an empty Rule for this target.

Otherwise, if there is no rule for this target, we return None

rules_for_target (*label*, *useTags=True*, *useMatch=True*)
Return the set of rules for any target(s) matching the given label.

- If useTags is true, then we should take account of tags when matching, otherwise we should ignore them. If useMatch is true, then useTags is ignored.
- If useMatch is true, then we allow wildcards in 'label', otherwise we do not.

Returns the set of Rules found, or an empty set if none were found.

rules_which_depend_on (*label*, *useTags=True*, *useMatch=True*)
Given a label, return a set of the rules which have it as one of their dependencies.

If there are no rules which have this label as one of their dependencies, we return the empty set.

- If useTags is true, then we should take account of tags when matching, otherwise we should ignore them. If useMatch is true, then useTags is ignored.
- If useMatch is true, then we allow wildcards in 'label', otherwise we do not.

targets_match(*target*, *useMatch=True*)

Return the set of targets matching the given 'target' label.

If useMatch is true, allow wildcards in 'target' (in which case more than one result may be obtained). If useMatch is false, then at most one match can be found ('target' itself).

Returns a set of suitable targets, or an empty set if there are none.

to_string(*matchLabel=None*, *showUser=True*, *showSystem=True*, *ignore_empty=False*)

Return a string representing this rule set.

If *showSystem* is true, include dependency labels with the System tag (i.e., dependencies inserted by the muddle system itself), otherwise ignore such.

If *showUser* is true, include dependency labels without the System tag (i.e., "normal" dependencies, explicitly added by the user), otherwise ignore such.

The default is to show all of the dependencies.

For instance (not a very realistic example):

```
>>> l = Label.from_string('package:fred{bob}/initial')
>>> r = RuleSet()
>>> depend_chain(None, l, ['built', 'bamboozled'], r)
>>> print str(r)
-----
package:fred{bob}/bamboozled <- [ package:fred{bob}/built ]
package:fred{bob}/built <- [ package:fred{bob}/initial ]
package:fred{bob}/initial <- [ ]
-----
```

The "<-" is to be read "depends on".

Note that the order of the rules in the output is sorted by target label, and is thus reproducible.

unify(*source*, *target*)

Merge source into target.

This is a pain, and depends heavily on CatenatedObject

wrap_actions(*generator*, *label*)

class muddled.depend.**SequentialAction**(*a*, *b*)

Bases: object

Invoke two actions in turn

build_label(*builder*, *label*)

muddled.depend.**checkout**(*name*, *tag='*'*, *domain=None*)

A simple convenience function to return a checkout label

- 'name' is the checkout name
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
    >>> l = checkout('fred')
    >>> l == Label.from_string('checkout:fred/*')
    True
```

muddled.depend.**depend_chain**(*action*, *label*, *tags*, *ruleset*)
Add a chain of dependencies to the given ruleset.

This is perhaps best explained with an example:

```
    >>> l = Label.from_string('package:fred{bob}/initial')
    >>> r = RuleSet()
    >>> depend_chain(None, l, ['built', 'bamboozled'], r)
    >>> print str(r)
    -----
    package:fred{bob}/bamboozled <- [ package:fred{bob}/built ]
    package:fred{bob}/built <- [ package:fred{bob}/initial ]
    package:fred{bob}/initial <- [ ]
    -----
```

muddled.depend.**depend_empty**(*action*, *label*)
Create a dependency set with no prerequisites - simply signals that a tag is available to be built at any time.

muddled.depend.**depend_none**(*action*, *label*)
Quick rule that makes label depend on nothing.

muddled.depend.**depend_one**(*action*, *label*, *dep_label*)
Quick rule that makes label depend only on dep_label.

muddled.depend.**depend_self**(*action*, *label*, *old_tag*)
Make a quick dependency set that depends just on you. Used by some of the standard package and checkout classes to quickly build standard dependency sets.

muddled.depend.**deployment**(*name*, *tag='*'*, *domain=None*)
A simple convenience function to return a deployment label

- 'name' is the deployment name
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
    >>> l = deployment('fred')
    >>> l == Label.from_string('deployment:fred/*')
    True
```

muddled.depend.**label_from_string**(*str*)
Do not use this!!! Can you say "deprecated"?

This function was originally removed, since it is replaced by Label.from_string. However, too many old builds still attempt to import it, which can cause problems at the "muddle init" stage, and also with "muddle unstamp".

Please do not use this function in new builds.

muddled.depend.**label_list_to_string**(*labels*, *join_with=' '*)

muddled.depend.**label_set_to_string**(*label_set*, *start_with='['*, *end_with=']'*, *join_with=', '*)
Utility function to convert a label set to a string.

muddled.depend.**needed_to_build**(*ruleset*, *target*, *useTags=True*, *useMatch=False*)
Given a rule set and a target, return a complete list of the rules needed to build the target.

- If useTags is true, then we should take account of tags when looking for the rules for this 'target', otherwise we should ignore them.
- If useMatch is true, then we allow wildcards in 'target', otherwise we do not.

Returns a list of rules.

muddled.depend.**normalise_checkout_label**(*label*, *tag='*'*)
Given a checkout label with random "other" fields, normalise it.

Returns a normalised checkout label, with the role unset and the tag set to 'tag' (normally, "*"). This may be the same label (if it was already normalised), or it may be a new label. No guarantee is given of either.

Raise a MuddleBug exception if the label is not a checkout label.

A normalised checkout label:

1. Has the given tag (normally '*', sometimes LabelTag.CheckedOut)
2. Does not have a role (checkout labels do not use the role)
3. Does not have the system or transient flags set
4. Has the same name and (if present) domain

muddled.depend.**package**(*name*, *role*, *tag='*'*, *domain=None*)
A simple convenience function to return a package label

- 'name' is the package name
- 'role' is the package role
- 'tag' is the label tag, defaulting to "*"
- 'domain' is the label domain name, defaulting to None

For instance:

```
>>> l = package('fred', 'jim')
>>> l == Label.from_string('package:fred{jim}/*')
True
```

muddled.depend.**required_by**(*ruleset*, *label*, *useTags=True*, *useMatch=True*)
Given a ruleset and a label, form the list of labels that (directly or indirectly) depend on label. We deliberately do not give you the associated rules since you will want to call needed_to_build() individually to ensure that other prerequisites are satisfied.

The order in which we give you the labels gives you a hint as to a logical order to rebuild in (i.e. one the user will vaguely understand).

- useMatch - If True, do wildcard matches, else do an exact comparison.
- **useTags - If False, we discount the value of a tag - this effectively** results in a wildcard tag search.

Returns a set of labels to build.

muddled.depend.**retag_label_list**(*labels*, *new_tag*)
Does what it says on the tin, returning the new label list.

That is, returns a list formed by copying each Label in 'labels' and setting its tag to the given 'new_tag'.

muddled.depend.**rule_list_to_string**(*rule_list*)
Utility function to convert a rule list to a string.

muddled.depend.**rule_target_str**(*rule*)
Take a rule and return its target as a string. Mainly used as an argument for map so we can print lists of rules sensibly.

muddled.depend.**rule_with_least_dependencies**(*rules*)
Given a (Python) set of rules, find the 'best' one to use.

This is actually impossible by any rational metric, so you usually only expect to call this function with a set of size 1, in which case our metric really doesn't matter.

However, in a vague attempt to be somewhat intelligent, we return the element with the fewest direct dependencies.

17.2.6 muddled.deployment

Note: *The base of deployment support*

Common rules for deployments - basically just the clean rules.

class muddled.deployment.**CleanDeploymentBuilder**
Bases: *muddled.depend.Action*

build_label(*builder*, *label*)

muddled.deployment.**deployment_depends_on_deployment**(*builder*, *what*, *depends_on*, *domain=None*)
Inter-deployment dependencies. Aren't you glad we have a general purpose dependency solver?

muddled.deployment.**deployment_depends_on_roles**(*builder*, *deployment*, *roles*, *domain=None*)
Make the deployment of the deployment with the given name depend on the installation of every package in the given role

muddled.deployment.**deployment_rule_from_name**(*builder*, *name*)
Return the rule for target label "deployment:<name>{}/deployed".

Raises an exception if there is more than one such rule.

muddled.deployment.**inform_deployment_path**(*builder*, *name*, *deployment*, *roles*, *domain=None*)
Sets an environment variable to tell the given roles about the location of the given deployment.

Useful when e.g. some tools need to run other tools and therefore want to know where they are at build (rather than run)time.

muddled.deployment.**pkg_depends_on_deployment**(*builder*, *pkg*, *roles*, *deployment*, *domain=None*)
Make this package depend on the given deployment

Specifically, given each role 'r' in 'roles', make the label "package:<pkg>{<r>}/preconfig" depend on the label "deployment:<deployment>/deployed".

If 'domain' is given, this is (currently) just used for the deploymet label.

muddled.deployment.**register_cleanup**(*builder*, *deployment*)
Register the rule you need to clean a deployment.

Cleaning a deployment basically means we remove the directory and its deployed tag.

muddled.deployment.**role_depends_on_deployment**(*builder*, *role*, *deployment*, *domain=None*)
Make every package in the given role depend on the given deployment

muddled.deployment.**set_env**(*builder*, *deployment*, *name*, *value*)
Set NAME=VALUE in the environment for this deployment.

17.2.7 muddled.distribute

Note: *Distribution support - generating subsets of the build tree for distribution to others.*

Actions and mechanisms relating to distributing build trees

class muddled.distribute.**DistributeAction**(*name*, *data*)
Bases: *muddled.depend.Action*

An action that distributes a something-or-other.

Intended as a base class for actions that know what they're doing.

We contain one thing: a dictionary of {name : distribution information}.

'name' is the name of a distribution that this action supports.

'data' is the data that describes how we do the distribution - this will differ according to whether we are distributing a checkout or package.

add_distribution(*name*, *copy_vcs=None*, *private_files=None*)
Add a new named distribution.

It is an error if there is already a distribution of this name.

build_label(*builder*, *label*)
Override this to do the actual task of distribution.

distribution_names()
Return the distribution names we know about.

does_distribution(*name*)
Return True if we know this distribution name, False if not.

get_distribution(*name*)
Return the data for distribution 'name', or raise MuddleBug

merge_names(*other*)
Merge in the distribution names dictionary from another action.

Any names that the other action has that we don't will be copied over.

Any names we already have will be ignored.

class muddled.distribute.**DistributeBuildDescription**(*name*, *copy_vcs=None*, *private_files=None*, *replacement_build_desc=None*)
Bases: *muddled.distribute.DistributeAction*

This is a bit like DistributeCheckoutAction, but without 'just'.

'name' is the name of a DistributionContext. When created, we are told which DistributionContext we can be distributed by. Later on, other names may be added...

If 'copy_vcs' is False, then don't copy any VCS "special" files (['.git', '.gitignore', ...] or ['.bzr'], etc., depending on the VCS used for this checkout).

If 'copy_vcs' is True, then always copy such files.

If 'copy_vcs' is None, then do whatever "muddle distribute" indicates.

Note: *Note that copy_vcs may only distinguish true and false for the moment, so None may be equivalvent to False.*

If 'private_files' is given, it is a sequence of Python files, relative to the build description checkout directory, that must be replaced by empty files when the distribution is done. It is always copied (as a set). If it is None, then an empty set will be used.

If 'replacement_build_desc' is given, then it is a file (relative to the checkout directory) to be used instead of all the rest of the content of the build description checkout. It will be named using the appropriate name (as in `.muddle/Description`) when it is distributed. If a replacement build description is named, then 'copy_vcs' will be ignored, and no VCS will be copied. Similarly, 'private_files' will be ignored.

add_distribution(*name*, *copy_vcs=None*, *private_files=None*, *replacement_build_desc=None*)
Add a new named distribution.

It is an error if there is already a distribution of this name.

add_private_files(*name*, *private_files*)
Add some specific private files to distribution 'name'.

Distribution 'name' must already be present on this action.

Does nothing if the files are already added

build_label(*builder*, *label*)

copying_vcs(*name*)
Are we distributing the VCS directory?

set_copy_vcs(*name*, *copy_vcs*)
Change the value of copy_vcs for distribution 'name'.

set_replacement_build_desc(*name*, *replacement_build_desc*)
Change the value of replacement_build_desc for distribution 'name'.

Note that 'None' is a perfectly sensible value.

class muddled.distribute.**DistributeCheckout**(*name*, *copy_vcs=False*, *just=None*)
Bases: *muddled.distribute.DistributeAction*

An action that distributes a checkout.

By default it copies the whole of the checkout source directory, not including any VCS files (.git/, etc.)

A mechanism for only copying *some* files is also included. This is typically used by "binary" packages that want to copy (for instance) the muddle Makefile for a checkout, but not all the source code.

Each checkout distribution is associated with a data tuple of the form:

(copy_vcs, specific_files)

where specific_files is None or a set of specific files for distibution.

'name' is the name of a DistributuionContext. When created, we are told which DistributionContext we can be distributed by. Later on, other names may be added...

If 'copy_vcs' is false, then don't copy any VCS "special" files ['.git', '.gitignore', ...] or ['.bzr'], etc., depending on the VCS used for this checkout).

If 'just' is None, then we wish to distribute all the source files in the checkout (apart from VCS files, for which see 'copy_vcs').

If 'just' is not None, then it must be a sequence of source paths, relative to the checkout directory, which are the specific files to be distributed for this checkout.

Note that we distinguish between 'just=None' and 'just=[]': the former instructs us to distribute all source files, the latter instructs us to distribute no source files.

add_distribution(*name*, *copy_vcs=None*, *just=None*)
Add a new named distribution.

The arguments are interpreted as when creating an instance.

It is an error if there is already a distribution of this name.

add_source_files(*name*, *source_files*)
Add some specific source files to distribution 'name'.

If we're already distributing the whole checkout, then this does nothing, as we're already outputting all the source files.

Don't try to use this to add the VCS directory to a distribution of all source files that was instantiated with copy_vcs, as the clause above will make that fail...

build_label(*builder*, *label*)

copying_all_source_files(*name*)
Are we distributing all the soruce files?

copying_vcs(*name*)
Are we distributing the VCS directory?

override(*name*, *copy_vcs=None*, *just=None*)
Override the definition of an existing named distribution.

request_all_source_files(*name*)
Request that distribution 'name' distribute all source files.

This is a tidy way of undoing any selection of specific files.

set_copy_vcs(*name*, *copy_vcs*)
Change the value of copy_vcs for distribution 'name'.

class muddled.distribute.**DistributePackage**(*name*, *obj=True*, *install=True*)
Bases: *muddled.distribute.DistributeAction*

An action that distributes a package.

If the package is being distributed as binary, then this action copies the obj/ and install/ directories for the package, as well as any instructions (and anything else I haven't yet thought of).

If the package is being distributed as source, then this action copies the source directory for each checkout that is *directly* used by the package.

In either case, associated and appropriate muddle tags are also copied.

Destination directories that do not exist are created as necessary.

'name' is the name of a DistributuionContext. When created, we are told which DistributionContext we can be distributed by. Later on, other names may be added...

If 'obj' is true, then the obj/ directory (and the associated muddle tags) should be copied.

If 'install' is true, then the install/ directory (and the associated muddle tags) should be copied

Notes:

1. We don't forbid having any particular combinations of 'obj' and 'install', although both False is not terribly useful.

add_distribution(*name*, *obj=True*, *install=True*)
Add a new named distribution.

It is an error if there is already a distribution of this name.

add_or_set_distribution(*name*, *obj=True*, *install=True*)
Add a distribution if it's not there, or replace it if it is.

build_label(*builder*, *label*)

muddled.distribute.**distribute**(*builder*, *name*, *target_dir*, *with_versions_dir=False*, *with_vcs=False*, *no_muddle_makefile=False*, *no_op=False*, *package_labels=None*, *checkout_labels=None*)
Distribute using distribution context 'name', to 'target_dir'.

The DistributeContext called 'name' must exist.

All distributions described in that DistributeContext will be made.

'name' is the name of the distribution to, erm, distribute. The special names:

- _source_release (all checkout source directories)
- _binary_release (all install directories, maybe plus extras)
- _for_gpl (just GPL and GPL-propagated source directories)
- _all_open (all open licensed source directories)
- _by_license (source or install directories by license, nothing private)

are always recognised. See the code or "muddle help distribute" for a more complete description of these.

'target_dir' is where to put the distribution. It will be created if necessary.

If 'with_versions_dir' is true, then any stamp "versions/" directory will also be distributed.

If "with_vcs" is true, then the VCS directory (.git/ for git, etc.) will be copied for:

- the build description(s)
- the "versions/" directory (if it is distributed)
- all checkouts in a "_source_release" distribution

If 'no_muddle_makefile' is true, then the appropriate muddle Makefile (in the appropriate checkout) will *not* be distributed with a package.

If 'no_op' is true, then we just report on what we would do - this lists the labels that would be distributed, and the action that would be used to do so.

If 'package_labels' and/or 'checkout_labels' is not None, then the labels selected for distribution will be "filtered" through those sequences, and only labels that occur in one or the other will be added to the distribution. Note that this filtering is done before adding in build descriptions. Passing them both as empty sets is likely to give a very small distribution...

NB: We assume that each package label in 'package_labels' has had the checkouts it directly depends upon added to 'checkout_labels' by the caller. Also, all labels must have their tag as '*'.

muddled.distribute.**distribute_build_desc**(*builder*, *name*, *label*, *copy_vcs=False*)
Request the distribution of the given build description checkout.

- 'name' is the name of the distribution we're adding this build description to.

  Note that this function is normally used by muddle itself, and it does not support any wildcarding of 'name'.

- 'label' must be a checkout label, but the tag is not important.

- **'copy_vcs' says whether we should copy VCS "special" files (so, for** git this includes at least the '.git' directory, and any '.gitignore' or '.gitmodules' files). The default is not to do so.

Notes:

1. If there was already a DistributeBuildDescription action defined for this build description's checkout, then we will amend it to look as if we created it (but leaving any "private" file requests untouched).
2. If there was already a DistributeBuildCheckout action defined for this build description's checkout, then we will replace it with a DistributeBuildDescription action. All we'll copy over from the older action is the distribution names.

_distribution/<name>.py files

If the build description checkout contains a file called `_distribution/<name>.py`, where `<name>` is the 'name' of the distribution we're building, then that file will be distributed as the build description (using the appropriate name found from the `.muddle/Description` file), and all other files in the build description checkout will be ignored. Note that this also means that in this case any calls of `'set_private_build_files()` will be ignored.

Since the name of the file is specifically tied to the distribution name, no license checking is done in this case - if you are doing a "_for_gpl" distribution, and provide a `_distribution/_for_gpl.py` file, then it is assumed that this was deliberate, whatever license the main build description may have.

Also, 'copy_vcs' will be ignored in this situation, and any VCS data will not be copied.

`muddled.distribute.`**`distribute_checkout`**(*builder*, *name*, *label*, *copy_vcs=False*)

Request the distribution of the specified checkout(s).

- 'name' is the name of the distribution we're adding this checkout to, or a "shell pattern" matching existing (already named) distributions. In that case:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

- 'label' must be

  1. a checkout label, in which case that checkout will be distributed
  2. a package label, in which case all the checkouts directly used by the package will be distributed (this is identical to calling 'distribute_checkout' on each of them in turn). Note that in this case the same value of 'copy_vcs' will be used for all the checkouts. Either the package name or package role may be wildcarded, in which case the checkouts directly used by each matching label will be distributed.

  In either case, the label tag is ignored.

- **'copy_vcs' says whether we should copy VCS "special" files (so, for** git this includes at least the '.git' directory, and any '.gitignore' or '.gitmodules' files). The default is not to do so.

All files and directories within the specified checkout(s) will be distributed, except for the VCS "special" files, whose distribution depends on 'copy_vcs'.

Notes:

1. If we already described a distribution called 'name' for a given checkout label, then this will silently overwrite it.

muddled.distribute.**distribute_checkout_files**(*builder*, *name*, *label*, *source_files*)

Request the distribution of extra files from a particular checkout.

- 'name' is the name of the distribution we're adding this checkout to, or a "shell pattern" matching existing (already named) distributions. In that case:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

- 'label' must be a checkout label. The label tag is not important.
- 'specified_files' is a sequence of file paths, relative to the checkout directory.

The intent of this function is to allow adding a small number of source files from a checkout to a binary package distribution, typically so that the necessary Makefiles and other build infrastructure is distributed. So, for instance:

```
label = Label.from_string
distribute_package(builder, 'marmalade', label('package:binapp{x86}/*'),
                   obj=True, install=True, with_muddle_makefile=True)
distribute_checkout(builder, 'marmalade', label('checkout:binapp-1.2/*'),
                    ['Makefile', 'src/Makefile', 'src/rules'])
```

Notes:

1. If we already described a distribution called 'name' for a given checkout label, then this will, if necessary, add the given source files to that distribution.
2. However, if that previous distribution was distributing "all files" (i.e., created with 'distribute_checkout()'), then we will not alter the action. This means that this call may not be used to override the 'copy_vcs' choice by trying to specify the VCS directory as an extra source path...

muddled.distribute.**distribute_package**(*builder*, *name*, *label*, *obj=False*, *install=True*, *with_muddle_makefile=True*)

Request the distribution of the given package.

- 'name' is the name of the distribution we're adding this checkout to, or a "shell pattern" matching existing (already named) distributions. In that case:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

- 'label' must be a package label. Either the name or the role may be wildcarded, in which case this function will be called on each matching label. The label tag is ignored.
- If 'obj' is true, then the obj/ directory (and the associated muddle tags) should be copied.
- If 'install' is true, then the install/ directory (and the associated muddle tags) should be copied
- If 'with_muddle_makefile' is true, then the muddle Makefile associated with building this package will also be distributed.

  This is implemented by looking up the MakeBuilder action used to build the package, finding the checkout and Makefile name from that, and then calling 'distribute_checkout_files()' to add that file in that checkout to the distribution.

For most distributions with obj=False, install=True, this is probably a useful option.

The 'with_muddle_makefile=True' mechanism is a fair attempt at allowing the distributed obj/ and install/ directory contents to be built, but doesn't support things like calling a different makefile or including other files directly in the muddle Makefile.

If you need to specify extra files, that can be done with additional calls to 'distribute_checkout_files()'.

Notes:

1. We don't forbid having any particular combinations of 'obj' and 'install', although both False is not terribly useful.
2. If we already described a distribution called 'name' for 'label', then this will silently overwrite it.

muddled.distribute.**get_distribution_names**(*builder=None*)
Return the known distribution names.

Note that 'builder' is optional.

muddled.distribute.**get_distributions_by_category**(*builder*)
Return a dictionary of distribution names according to license category.

The dictionary returned has license category names as keys, and sets of distribution names as the values.

muddled.distribute.**get_distributions_for**(*builder*, *categories*)
Return distributions that distribute all the given 'categories'

That is, for each distribution, look and see if the license categories it distributes for include all the values in 'categories', and if it does, add its name to the result.

For instance, we know we should always have at least two distributions that work for category 'binary':

```
>>> dists = get_distributions_for(None, ['binary'])
>>> '_by_license' in dists
True
>>> '_binary_release' in dists
True
>>> '_for_gpl' in dists
False
```

'builder' is ignored at the moment, but should be the build tree "builder" if available.

muddled.distribute.**get_distributions_not_for**(*builder*, *categories*)
Return distributions that distribute none of the given 'categories'

That is, for each distribution, look and see if the license categories it distributes for include any of the values in 'categories', and if it does not, add its name to the result.

For instance, we know that we have at least one distribution that is not for 'binary' and 'secure':

```
>>> dists = get_distributions_not_for(None, ['binary', 'secure'])
>>> '_for_gpl' in dists
True
>>> '_source_release' in dists
False
```

Asking for distributions that don't do anything should hopefully return an empty list:

```
>>> get_distributions_not_for(None, ALL_LICENSE_CATEGORIES)
[]
```

'builder' is ignored at the moment, but should be the build tree "builder" if available.

muddled.distribute.**get_used_distribution_names**(*builder*)
Return a set of all the distribution names that are actually in use

"in use" is taken to mean that some rule in the dependency tree has an action that is defined for that particular distribution name.

muddled.distribute.**name_distribution**(*builder*, *name*, *categories=None*)
Declare that a distribution called 'name' exists.

Also specify which license categories are distributed.

If 'categories' is None, then all license categories are distributed.

Otherwise 'categories' must be a sequence of category names, taken from 'gpl', 'open-source', binary' and 'private'.

The user may assume that the standard distributions (see "muddle help distribute") already exist, but otherwise must name a distribution before it is used.

It is not an error to name a distribution more than once (although it won't have any effect), but the categories named must be identical.

```
>>> name_distribution(None, '_all_open', ['gpl', 'open-source'])  # same categories
>>> name_distribution(None, '_all_open', ['open-source']) # different categories
Traceback (most recent call last):
...
GiveUp: Attempt to name distribution "_all_open" with categories "open-source" but it already ha
```

It is an error to try to use a distribution before it has been named. This includes adding checkouts and packages to distributions. Wildcard operations will only take account of the distributions that have already been named.

Distribution names that start with an underscore are reserved by muddle to define as it wishes, although we don't stop you naming a distribution that starts with an underscore (just remember muddle may take the name later without warning).

muddled.distribute.**select_all_binary_nonprivate_packages**(*builder*, *name*, *with_muddle_makefile*, *just_from=None*)
Select all packages with a "binary" license for distribution.

'name' is the name of our distribution, for error reporting.

If 'with_muddle_makefile' is true, then we'll make an attempt to add distribution information for each package's muddle Makefile (in the appropriate checkout)

We do *not* want "private" packages, and as such this function checks to see if any "private" packages may be present in the install/ directories that we are proposing to distribute.

If 'just_from' is given, then we'll only consider the (package) labels therein.

muddled.distribute.**select_all_gpl_checkouts**(*builder*, *name*, *with_vcs*, *just_from=None*)
Select all checkouts with some sort of "gpl" license for distribution

(or any checkout that has had "gpl"-ness propagated to it)

(or any checkout that one of those depends on)

'name' is the name of our distribution.

'with_vcs' is true if we want VCS "special" files in our distributed checkouts.

If 'just_from' is given, then we'll only consider the labels therein.

muddled.distribute.**select_all_open_checkouts**(*builder*, *name*, *with_vcs*, *just_from=None*)
Select all checkouts with an "open" license for distribution.

This includes all "gpl" checkouts, and all checkouts made implicitly "gpl".

'name' is the name of our distribution.

'with_vcs' is true if we want VCS "special" files in our distributed checkouts.

If 'just_from' is given, then we'll only consider the labels therein.

muddled.distribute.**select_all_prop_source_checkouts**(*builder*, *name*, *with_vcs*, *just_from=None*)

Select all checkouts with a "prop-source" license for distribution.

'name' is the name of our distribution.

'with_vcs' is true if we want VCS "special" files in our distributed checkouts.

If 'just_from' is given, then we'll only consider the labels therein.

muddled.distribute.**set_private_build_files**(*builder*, *name*, *private_files*)

Set some private build files for the (current) build description.

These are files within the build description directory that will replaced by dummy files when doing the distribution.

- 'name' is the name of the distribution we're adding this checkout to, or a "shell pattern" matching existing (already named) distributions. In that case:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any char not in seq
```

- 'private_files' is the list of the files that must be distributed as dummy files. They are relative to the build description checkout directory.

The "original" private files must exist and must work by providing a function with signature:

```
def describe_private(builder, *args, **kwargs):
    ...
```

The dummy files will also containg such a function, but its body will be `pass`.

17.2.8 muddled.env_store

Note: *Internal environment handling*

An environment store holds a set of instructions for manipulating a set of environment variables

Sometimes we need to generate these for C. This is particularly evil because C neither has good environment variable lookup nor good string handling support.

See the boilerplate in `resources/c_env.c` for how we handle this. It's not pretty ..

class muddled.env_store.**EnvBuilder**(*external=False*)

Bases: object

Represents a way of building an environment variable value from a series of instructions.

- prepend_list - List of paths to prepend to the value
- retain_old_value - Retain the old value?

- append_list - List of things to append to the old value.
- env_type - Type of this environment variable
- erased - Have we been erased?
- external - This variable is defined externally.

All paths are now of EnvExprs.

append(*val*)

append_expr(*val*)
Append val to this environment value.

copy()

dependencies()
Return a set of the environment variables that this value depends on.

empty()
Is this environment builder empty? i.e. does it have an empty value?

ensure_appended(*val*)

ensure_appended_expr(*val*)
Make sure val is appended to the value or append it.

Returns True if we added the value, False if it was already there

ensure_prepended(*val*)

ensure_prepended_expr(*val*)
Make sure val is part of the value or prepend it. What you usually want for paths.

Returns True if we added the value, False if it was already there.

erase()

get(*inOldValue*, *language*)

get_c(*var*, *prefix*, *variable_name*)
Return a string containing C code which leaves the value of this builder in 'var'.

The string does *not* declare var, - that's the caller's job.

variable_name is the variable name whose value we're processing - it's needed so we can refer to its previous value.

get_py(*inOldValue*, *env_name='os.environ'*)
Like get, but in python syntax.

- inOldValue - A python expression which gives the old value.

get_sh(*inOldValue*, *doQuote*)
The old value of this variable was inOldValue; what is its new value?

- doQuote - if doQuote is true, we'll use shell quoting. We never quote inOldValue since it's probably $PATH or something else that shouldn't be quoted.

get_value(*inOldValue*, *env={}*)

merge(*other*)
Merge another environment builder with this one.

prepend(*val*)

prepend_expr(*val*)
Prepend val to this environment value.

set(*val*)

set_expr(*val*)
Set val to this environment value.

set_external(*external=True*)

set_type(*type*)

class muddled.env_store.**EnvExpr**(*type*, *val=None*)
Bases: object

An environment variable expression. This allows us to symbolically represent things like catenating one variable value with another.

CatType = 'Cat'

RefType = 'Ref'

StringType = 'String'

append(*other*)

append_ref(*ref*)

append_str(*str*)

augment_dependency_set(*a_set*)
Add the environment variables this expression depends on to a_set.

same_as(*other*)
Decide if two EnvExprs will produce the same value on output.

to_c(*var*, *prefix*)
Returns a list of strings you can join together to make C code for constructing the value of this expression

- var - The name of the C variable we're creating.
- prefix - The name of the prefix to prepend to function calls.

to_py(*env_var*)
Return a list of expressions you can put into a literal python list after `"".join()` to write the correct value for this variable.

to_sh(*doQuote*)

to_value(*env*)

class muddled.env_store.**EnvLanguage**
Bases: object

Languages in which we can generate setenv files.

C = 3

Python = 1

Sh = 0

Value = 2

class muddled.env_store.**EnvMode**
Bases: object

Ways of manipulating environment variables.

Append = 0

Prepend = 2

Replace = 1

class muddled.env_store.**EnvType**
Bases: object

Types of environment variable.

SimpleValue is just a value (the default)

Path colon-separated path

Path = 1

SimpleValue = 0

class muddled.env_store.**Store**
Bases: object

Maintains a store of environment variables and allows us to apply them to any given environment dictionary.

append(*name*, *value*)
Append a value to a variable.

append_expr(*name*, *expr*)
Append an EnvExpr to a variable.

apply(*in_env*)
Apply the modifications here to the environment in dict.

builder_for_name(*name*)
Return a builder for the given variable, inventing one if there isn't already one

copy()

dependency_sort()
Sort self.vars.items() in as close to dependency order as you can.

empty(*name*)
Return True iff name has a builder with an empty value, False otherwise.

(i.e. if it's likely to actually generate an environment variable setting)

ensure_appended(*name*, *value*)

ensure_prepended(*name*, *value*)

erase(*name*)
Explicitly erase a variable.

external(*name*)

get_c_subst_var(*prefix*)
Returns the block of C to use as a substitute for body_impl in resources/c_env.c

get_setvars_c(*builder*, *prefix*)

get_setvars_py(*name*)
Write some statements that will set the relevant environment variables in python.

Returns a string containing the relevant python code.

get_setvars_script(*builder*, *name*, *lang*)
Write a setvars script in the chosen language.

get_setvars_sh(*name*)
Write a setvars script.

Returns a string containing the script.

merge(*other*)
Merge another environment store into this one. Instructions from the new store will override or augment those in self.

op(*name*, *mode*, *value*)
Perform mode (an EnvMode) on name with value.

prepend(*name*, *value*)
Prepend a value to a variable.

prepend_expr(*name*, *expr*)
Prepend an EnvExpr to a variable.

set(*name*, *value*)
Set a value for a name.

set_expr(*name*, *expr*)
Set a variable to an EnvExpr.

set_external(*name*)

set_type(*name*, *type*)

muddled.env_store.**add_install_dir_env**(*env*, *var_name*)
Add an install directory, whose base is held in var_name, to:

- PATH
- LD_LIBRARY_PATH
- PKG_CONFIG_PATH

muddled.env_store.**append_expr**(*var*, *str*)
Create an environment expression consisting of the given string appended to the given variable name.

muddled.env_store.**prepend_expr**(*astr*, *var*)

Create an environment expression consisting of a string prepended to a variable

muddled.env_store.**print_deps**(*deps*)
Given a dictionary mapping environment variable names to sets of dependencies, return a string representing the map.

muddled.env_store.**set_expr**(*var*)
Create an environment expression consisting of a reference to the given variable name

muddled.env_store.**string_expr**(*var*)
Create an environment expression consisting of a literal string.

17.2.9 muddled.filespec

Note: *Internal handling of file specifications, for deployment*

FileSpecs: a cheap and cheerful way to specify files for the purposes of deployment instructions

class muddled.filespec.**FSFileSpecDataProvider**(*base_dir*)
Bases: object

A FileSpecDataProvider rooted at a particular point in the filesystem

abs_match(*filespec*)
Match the filespec to this data provider and return a list of actual absolute filenames on which to operate

list_files_under(*dir*, *recursively=False*, *vroot=None*)

class muddled.filespec.**FileSpec**(*root*, *spec*, *allUnder=False*, *allRegex=False*)
Bases: object

Represents a (possibly recursive) file specification. Filespecs are essentially python regular expressions with a recursion flag.

Matching for these objects is slightly special, since they need to apply to objects in the filesystem. Filespecs contain a root, which bounds the spec, a specifier, which is a regular expression indicating which files in that spec should match, and two recursion flags:

- all_under - This filespec applies to all files under any directories that match the base filespec.
- all_regex - The specifier applies as a regex to all files under the root. This can be very slow if there are many files under the filespec root.

clone_from_xml(*xmlNode*)
Clone a filespec from some XML like:

```
<filespec>
 <root>..</root>
 <spec> ..</spec>
 <all-under />
 <all-regex />
</filespec>
```

equal(*other*)

is_filespec_node(*inXmlNode*)
Given an XML node, decide if this is likely to be a filespec (really just checks if it's an element of the right name). Useful for parsing documents that may contain filespecs.

match(*data_provider*, *vroot=None*)
Match this filespec with a data provider, returning a set of file- and directory- names upon which to operate.

Since we have no idea what the root path of the data provider might be, you probably need to stitch together the filenames yourself.

`FSFileSpecDataProvider.abs_match()` is probably your friend - if you're using the filesystem to provide data for a filespec, call it, not us.

vroot is a 'virtual root' - the data provider transparently returns the subset of files that would be present if 'vroot' in the data provider were the root. It is used by remappings in the cpio deployment, among others.

outer_elem_name()

to_xml(*doc*)
Create some XML from this filespec.

class muddled.filespec.**FileSpecDataProvider**
Bases: object

Provides data to a filespec so it can decide what it matches.

list_files_under(*dir*, *recursively=False*, *vroot=None*)
Return a list of the files under dir. If dir is not a directory, returns an empty list.

The files are returned without 'dir', so:

```
list_files_under("/fred/wombat", False)
```

gives:

```
[ "a", "b", "c" ]
```

not:

```
[ "/fred/wombat/a" .. ]
```

whilst:

```
list_files_under("/fred", True)
```

gives:

```
[ "wombat", "wombat/a", "wombat/b", "wombat/c" ]
```

class muddled.filespec.**ListFileSpecDataProvider**(*file_list*)
Bases: object

A FileSpecDataProvider that uses a file list. Used to test the FileSpec matching code.

list_files_under(*dir*, *recursively=False*, *vroot=None*)

17.2.10 muddled.instr

Note: *Internal handling of instruction files, for deployment*

Routines and classes which cope with instructions

class muddled.instr.**BuiltinInstructionFactory**
Bases: *muddled.db.InstructionFactory*

An instruction factory that can build all the built-in instructions. You can extend or augment this class to generate a factory which builds your favourite add-on instructions.

(though note that your favourite deployment will need to understand them in order to to obey them)

instr_map Maps instruction names to prototype classes, which can then be cloned

from_xml(*xmlNode*)

register(*name*, *instruction*)

class muddled.instr.**ChangeModeInstruction**(*filespec*, *new_mode*, *name*)
Bases: *muddled.db.Instruction*

Change the mode of a filespec (chown).

clone_from_xml(*node*)

equal(*other*)

outer_elem_name()

to_xml(*doc*)

class muddled.instr.**ChangeUserInstruction**(*filespec*, *new_user*, *new_group*, *name*)
Bases: *muddled.db.Instruction*

An instruction that takes a username, groupname and filespec.

This is the base class for chown and chgrp.

clone_from_xml(*node*)

equal(*other*)

outer_elem_name()

to_xml(*doc*)

class muddled.instr.**MakeDeviceInstruction**
Bases: *muddled.db.Instruction*

Create a device file - this is essentially `mknod`.

clone_from_xml(*node*)

equal(*other*)

outer_elem_name()

to_xml(*doc*)

validate()

muddled.instr.**sanitise_filename**(*name*)
Sanitise a <name>filename</name>.

We want to make sure that the name is relative to the muddle directories. Specifically, we want to make sure that if the filename is <name>/etc/passwd</name> then we do not try to access the host system's `/etc/passwd` file, but rather a local `.../etc/passwd`.

It turns out the simplest thing to do is to remove any initial "/", rendering the name relative...

17.2.11 muddled.licenses

Note: *License annotation - indicating the type of license that applies to checkouts, for use in making distributions.*

Matters relating to attributing licenses to checkouts

class muddled.licenses.**License**(*name*, *category*, *version=None*)
Bases: object

The representation of a source license.

License instances should be:

1.Singular

2.Immutable

but I don't particularly propose to work hard to enforce those...

Use the subclasses to create your actual License instance, so that you can use any appropriate extra methods...

Initialise a new License.

The 'name' is the name of this license, as it is normally recognised.

'category' is meant to be a broad categorisation of the type of the license. Currently that is one of:

- 'gpl' - some sort of GPL license, which propagate the need to distribute source code to other "adjacent" entities
- 'open-source' - an open source license, anything that is not 'gpl'. Source code may, but need not be, distributed.
- 'prop-source' - a proprietary source license, not an open source license. This might, for instance, be used for /etc files, which are distributed as "source code" (i.e., text), but are not in fact licensed under an open source license.
- 'binary' - a binary license, indicating that the source code is not to be distributed, but binary (the contents of the "install" directory) may be.
- 'private' - a marker that the checkout should not be distributed at all.

'version' may be a version string. If it is None, then it will not be shown in the str() or repr() for a license.

copy_with_version(*version*)
Make a copy of this license with a different version string.

distribute_as_source()
Returns True if we this license is for source distribution.

Currently, equivalent to having a category of open-source, gpl or prop-source.

is_binary()
Is this a binary-distribution-only license?

is_gpl()
Returns True if this is some sort of GPL license.

is_lgpl()
Returns True if this is some sort of LGPL license.

This *only* works for the LicenseLGpl class (and any subclasses of it, of course).

is_open()
Returns True if this is some sort of open-source license.

Note: this includes GPL and LGPL licenses.

is_open_not_gpl()
Returns True if this license is 'open-source' but not 'gpl'.

is_private()
Is this a private-do-not-distribute license?

is_proprietary_source()
Returns True if this is some sort of propetary source license.

(i.e., has category 'prop-source')

Note: this does *not* include 'open-source' or 'gpl'.

propagates()
Does this license "propagate" to other checkouts?

In other words, if checkout A has this license, and checkout B depends on checkout A, does the license have an effect on what you can do with checkout B?

For non-GPL licenses, the answer is assumed "no", and we thus return False.

For GPL licenses with a linking exception (e.g., the GCC runtime library, or some Java libraries with CLASSPATH exceptions), the answer is also "no", and we return False.

However, for most GPL licenses (and this includes LGPL), the answer if "yes", there is some form of propagation (remember, LGPL allows dynamic linking, and use of header files, but not static linking), and we return True.

If we return True, it is then up to the user to decide if this means anything in this particular case - muddle doesn't know *why* one checkout depends on another.

class muddled.licenses.**LicenseBinary**(*name*, *version=None*)
Bases: *muddled.licenses.License*

A binary license - we distribute binary only, not source code

class muddled.licenses.**LicenseGPL**(*name*, *version=None*, *with_exception=False*)
Bases: *muddled.licenses.License*

Some sort of GPL license.

(Why LicenseGPL rather than GPLLicense? Because I find the later more confusing with the adjacent 'L's, and I want to keep GPL uppercase...)

Initialise a new GPL License.

The 'name' is the name of this license, as it is normally recognised.

Some GNU libraries provide a linking exception, which allows software to "link" (depending on the exception) to the library, without needing to be GPL-compatible themselves. One example of this (more or less) is the LGPL, for which we have a separate class. Another example is the GCC Runtime Library.

is_gpl()
Returns True if this is some sort of GPL license.

propagates()
Does this license "propagate" to other checkouts?

In other words, if checkout A has this license, and checkout B depends on checkout A, does the license have an effect on what you can do with checkout B?

For non-GPL licenses, the answer is assumed "no", and we thus return False.

For GPL licenses with a linking exception (e.g., the GCC runtime library, or some Java libraries with CLASSPATH exceptions), the answer is also "no", and we return False.

However, for most GPL licenses (and this includes LGPL), the answer if "yes", there is some form of propagation (remember, LGPL allows dynamic linking, and use of header files, but not static linking), and we return True.

If we return True, it is then up to the user to decide if this means anything in this particular case - muddle doesn't know *why* one checkout depends on another.

class muddled.licenses.**LicenseLGPL**(*name*, *version=None*, *with_exception=False*)
Bases: *muddled.licenses.LicenseGPL*

Some sort of Lesser GPL (LGPL) license.

The lesser GPL implies that it is OK to link to this checkout as a shared library, or to include its header files, but not link statically. We don't treat that as a "with_exception" case specifically, since it is up to the user to decide if an individual checkout that depends on a checkout with this license is affected by our GPL-ness.

Initialise a new LGPL License.

The 'name' is the name of this license, as it is normally recognised.

is_lgpl()
Returns True if this is some sort of LGPL license.

class muddled.licenses.**LicenseOpen**(*name*, *version=None*)
Bases: *muddled.licenses.License*

Some non-GPL open source license.

This should probably be named "LicenseOpenSource", but that is rather long.

class muddled.licenses.**LicensePrivate**(*name*, *version=None*)
Bases: *muddled.licenses.License*

A "private" license - we do not want to distribute anything

class muddled.licenses.**LicenseProprietarySource**(*name*, *version=None*)
Bases: *muddled.licenses.License*

A source license, but not open source.

This is separate from the "open" licenses mainly because it is not, in fact, representing an open license, so even if it were to be treated identically in all matters, it would still be wrong.

The class name is rather long, but it is hard to think of a shorter name that explains what it is for.

muddled.licenses.**checkout_license_allowed**(*builder*, *co_label*, *categories*)
Does this checkout have a license in the given categories?

Returns True if the checkout has a license that is in any of the given categories, or if it does not have a license.

Returns False if it is licensed, but its license is not in any of the given categories.

muddled.licenses.**get_binary_checkouts**(*builder*)
Return a set of all the "binary" licensed checkouts.

muddled.licenses.**get_gpl_checkouts**(*builder*)
Return a set of all the GPL licensed checkouts.

That's checkouts with any sort of GPL license.

muddled.licenses.**get_implicit_gpl_checkouts**(*builder*)
Find all the checkouts to which GPL-ness propagates.

Returns a tuple, (result, because), where:

- 'result' is a set of the checkout labels that are implicitly made "GPL" by propagation, and
- **'because' is a dictionary linking each such label to a set of strings** explaining the reason for the labels inclusion

muddled.licenses.**get_license**(*builder*, *co_label*, *absent_is_None=True*)
Get the license for a checkout.

If 'absent_is_None' is true, then if 'co_label' does not have an entry in the licenses dictionary, None will be returned. Otherwise, an appropriate GiveUp exception will be raised.

This is a simple wrapper around builder.db.get_checkout_license.

muddled.licenses.**get_license_clashes**(*builder*, *implicit_gpl_checkouts*)
Return clashes between actual license and "implicit GPL" licensing.

`get_implicit_gpl_checkouts()` returns those checkouts that are implicitly "made" GPL by propagation. However, if the checkouts concerned were already licensed with either "binary" or "private" licenses, then it is likely that the caller would like to know about it, as it is probably a mistake (or at best an infelicity).

This function returns two sets, (bad_binary, bad_private), of checkouts named in `implicit_gpl_checkouts` that have an explicit "binary" or "private" license.

muddled.licenses.**get_license_clashes_in_role**(*builder*, *role*)
Find license clashes in the install/ directory of 'role'.

Returns two dictionaries (binary_items, private_items)

'binary_items' is a dictionary of {checkout_label : binary_license}

'private_items' is a dictionary of {checkout_label : private_license}

If private_items has content, then there is a licensing clash in the given role, as one cannot do a binary distribution of both "binary" and "private" licensed content in the same "install" directory.

muddled.licenses.**get_not_licensed_checkouts**(*builder*)
Return the set of all checkouts which do not have a license.

(Actually, a set of checkout labels, with the label tag "/checked_out").

muddled.licenses.**get_open_checkouts**(*builder*)
Return a set of all the open licensed checkouts.

That's checkouts with any sort of GPL or open license.

muddled.licenses.**get_open_not_gpl_checkouts**(*builder*)
Return a set of all the open licensed checkouts that are not GPL.

Note sure why anyone would want this, but it's easy to provide.

muddled.licenses.**get_private_checkouts**(*builder*)
Return a set of all the "private" licensed checkouts.

muddled.licenses.**get_prop_source_checkouts**(*builder*)
Return a set of all the "proprietary source" licensed checkouts.

muddled.licenses.**licenses_in_role**(*builder*, *role*)
Given a role, what licenses are used by the packages (checkouts) therein?

Returns a set of License instances. May also include None in the values in the set, if some of the checkouts are not licensed.

muddled.licenses.**print_standard_licenses**()

muddled.licenses.**report_license_clashes**(*builder*, *report_binary=True*, *report_private=True*, *just_for=None*)
Report any license clashes.

This wraps get_implicit_gpl_checkouts() and check_for_license_clashes(), plus some appropriate text reporting any problems.

It returns True if there were any clashes, False if there were not.

It reports clashes with "binary" licenses if 'report_binary' is True.

It reports clashes with "private" licenses if 'report_private' is True.

If both are False, it is silent.

If 'just_for' is None, it looks at all the implicit GPL checkouts. Otherwise, it only considers those labels in 'just_for' that are also implicitly GPL.

muddled.licenses.**report_license_clashes_in_role**(*builder*, *role*, *just_report_private=True*)
Report license clashes in the install/ directory of 'role'.

Basically, this function allows us to be unhappy if there are a mixture of "binary" and "private" things being put into the same "install/" directory.

If 'just_report_private' is true, then we will only talk about the private entities, otherwise we'll report the "binary" licensed packages that end up there as well.

If there was a clash reported, we return True, and otherwise we return False.

`muddled.licenses.`**`set_license`**(*builder*, *co_label*, *license*, *license_file=None*, *not_built_against=False*)

Set the license for a checkout.

'co_label' is either a checkout label, or the name of a checkout.

(Specifying a checkout label allows a domain name to be specified as well. The tag of the checkout label is ignored.)

'license' must either be a License instance, or the mnemonic for one of the standard licenses.

Some licenses (for instance, 'BSD-3-clause') require inclusion of their license file in binary distributions. 'license_file' allows the relevant file to be named (relative to the root of the checkout directory), and implies that said file should be included in all distributions.

If 'not_built_against' is True, then it will be noted that nothing "builds against" this checkout. See 'set_nothing_builds_against()' for more information - this parameter is a useful convenience to save an extra call.

`muddled.licenses.`**`set_license_for_names`**(*builder*, *co_names*, *license*)

A convenience function to set one license for several checkout names.

Since this uses checkout names rather than labels, it is not domain aware.

It calls 'set_license()' for each checkout name, passing it a checkout label constructed from the checkout name, with no domain.

`muddled.licenses.`**`set_license_not_affected_by`**(*builder*, *this_label*, *co_label*)

Asserts that the license for co_label does not affect this_label

We assume that:

1. 'this_label' is a package that depends (perhaps indirectly) on 'co_label', or is a checkout directly required by such a package.
2. 'co_label' is a checkout with a "propagating" license (i.e., some form of GPL license).
3. Thus by default the "GPL"ness would propagate from 'co_label' to 'this_label' (and, if it is a package, to the checkouts it is (directly) built from).

However, this function asserts that, in fact, this is not true. Our checkout is (or our checkouts are) not built in such a way as to cause the license for 'co_label' to propagate.

Or, putting it another way, for a normal GPL license, we're not linking with anything from 'co_label', or using its header files, or copying GPL'ed files from it, and so on.

If 'co_label' is under LGPL, then that would reduce to saying we're not static linking against 'co_label' (or anything else not allowed by the LGPL).

Note that we may be called before 'co_label' has registered its license, so we cannot actually check that 'co_label' has a propagating license (or, indeed, that it exists or is depended upon by 'pkg_label').

This is a simple wrapper around builder.db.set_license_not_affected_by.

`muddled.licenses.`**`set_nothing_builds_against`**(*builder*, *co_label*)

Asserts that no packages "build against" this checkout.

We assume that `co_label` is a checkout with a "propagating" license (i.e., some form of GPL license).

This function tells the distribution/licensing system that there are no packages that build against (link against) this checkout, in a way which would cause GPL license "propagation".

Typically used to mark checkouts that (just) provide or build:

- an application (a program)
- a kernel module
- a text file (e.g., something to be placed in /etc)

An example might be busybox, which is GPL-2 licensed, but which builds a set of independent programs.

17.2.12 muddled.mechanics

Note: *The main machinery that makes muddle go - the Builder class.*

Contains the mechanics of muddle.

class muddled.mechanics.**BuildDescriptionAction**(*file_name*, *build_co*)
Bases: *muddled.depend.Action*

Load the build description.

This action is used to read a build description. As such, it does not need to be domain aware - it is only ever done in the (current) top-level domain.

build_label(*builder*, *label*)
Actually load the build description.

Note that the Python path (sys.path) will have the build description checkout directory added to its start, so that the release_from() function itself can import things therefrom.

class muddled.mechanics.**Builder**(*root_path*, *muddle_binary*, *domain_params=None*, *default_domain=None*)
Bases: object

A builder does stuff following rules derived from a build description.

Don't construct a Builder directly, always use the 'load_builder()' function, or 'minimal_build_tree()' if that is more appropriate.

- self.db - The metadata database for this project.
- self.ruleset - The rules describing this build
- self.env - A dictionary of label to environment
- self.default_roles - The roles to build when you don't specify any. These will also be used for "guessing" a role for a package when one is not specified. '_default_roles' is calculated from this.
- self.default_deployment_labels - The deployments to deploy when you don't specify any specific roles. This is the meaning of '_default_deployments'.

There are then a series of values used in managing subdomains:

- self.banned_roles - An array of pairs of the form (role, domain) which aren't allowed to share libraries.
- self.domain_params - Maps domain names to dictionaries storing parameters that other domains can retrieve. This is used to communicate values from a build to its subdomains.
- self.unifications - This is a list of tuples of the form (source-label, target-label), where one "replaces" the other in the build tree.

And:

- self.what_to_release - a set of entities to build for a release build. It may contain labels and also "special" names, such as '_default_deployments' or even '_all'. You may not include '_release' (did you need to ask?). "_just_pulled" is not allowed either.

Construct a fresh Builder with a .muddle directory at the given 'root_path'.

'muddle_binary' is the full path to our muddle "binary" - the program that is muddle. This is needed when running muddle Makefiles that invoke $(MUDDLE).

'domain_params' is the set of domain parameters in effect when this builder is loaded. It's used to communicate values down to sub-domains.

Note that you MUST NOT set 'domain_params' Null unless you are the top-level domain - it MUST come from the enclosing domain's builder or modifications made by the subdomain's buidler will be lost, and as this is the only way to communicate values to a parent domain, this would be bad. Ugh.

'default_domain' is the default domain value to add to anything in local_pkgs , etc - it's used to make sure that if you're cd'd into a domain subdirectory, we build the right labels.

add_default_deployment_label(*label*)
Set the label that's built when you call muddle from the root directory

add_default_role(*role*)
Add role to the list of roles built when you don't ask for something different.

Returns False if we didn't actually add the role (it was already there), True if we did.

add_default_roles(*roles*)
Add the given roles to the list of default roles for this build.

add_to_release_build(*thing*)
Add a thing to the set of entities to build for a release build.

'thing' must be:

- a Label
- one of the "special" names, "_all", "_default_deployments", "_default_roles".
- a sequence of either/both

It may not be "_release" (!) or "_just_pulled".

Special names are expanded after all build descriptions have been read.

The special name "_release" corresponds to this set.

all_checkout_labels(*tag=None*)
Return a set of the labels of all the checkouts in our rule set.

Note that if 'tag' is None then all the labels will be of the form:

checkout:<co_name>/*

otherwise 'tag' will be used as the checkout label tag:

checkout:<co_name>/<tag>

all_checkout_rules()
Returns a set of the labels of all the checkouts in our rule set.

Specifically, all the rules for labels of the form:

```
checkout:*{*}/checked_out
checkout:(*)*{*}/checked_out
```

Returns a set of labels, thus allowing one to know the domain of each checkout as well as its name.

all_checkouts()
Return a set of the names of all the checkouts in our rule set.

Returns a set of strings.

This is not domain aware. Consider using all_checkout_labels(), which is.

all_deployment_labels(*required_tag*)
Return a set of all the deployment labels in our ruleset.

If 'required_tag' is given, then the labels returned will all have that tag (this, of course, may result in a smaller set of labels).

all_deployments()
Return a set of the names of all the deployments in our rule set.

Returns a set of strings.

all_domains()
Return a set of the names of all the domains in our rule set.

Returns a set of strings. The 'None' domain (the unnamed, top-level domain) is returned as the empty string, "".

all_package_labels()
Return a set of the labels of all the packages in our rule set.

all_packages()
Return a set of the names of all the packages in our rule set.

Returns a set of strings.

Note that if '*' is one of the package "names" in the ruleset, then it will be included in the names returned.

all_packages_with_roles()
Return a set of the names of all the packages/roles in our rule set.

Returns a set of strings.

Note that if '*' is one of the package "names" in the ruleset, then it will be included in the names returned.

However, any labels with role '*' will be ignored.

all_roles()
Return a set of the names of all the roles in our rule set.

Returns a set of strings.

apply_unifications(*source*)

build_co_and_path()
Return a pair (build_co, build_path).

build_desc_repo
The Repository for our build description.

This used to be a simple value, but is now a shim around looking it up in builder.db.checkout_data - so that we only have the information stored in one place.

Returns None if there is no Repository registered yet

build_label(*label*, *silent=False*)
The fundamental operation of a builder - build this label.

build_label_with_options(*label*, *useDepends=True*, *useTags=True*, *silent=False*)
The fundamental operation of a builder - build this label.

•useDepends - Use dependencies?

build_name
The build name is meant to be a short description of the purpose of a build. It might thus be something like "ProjectBlue_intel_STB" or "XWing-minimal".

The name may only contain alphanumerics, underlines and hyphens - this is to facilitate its use in version stamp filenames. Also, it is a superset of the allowed characters in a Python module name, which means that the build description filename (excluding its ".py") will be a legal build name (so we can use that as a default).

by_default_deploy(*deployment*)
Add a deployment label to the deployments to build by default.

by_default_deploy_list(*deployments*)
Now we've got a list of default labels, we can just add them ..

checkout_label_exists(*label*)
Return True if this checkout label is in any rules (i.e., is used).

Note that this method does not understand wildcards, so the match must be exact.

checkout_path(*label*)
Return the path in which the given checkout resides.

This is a simple wrapper around builder.db.get_checkout_path(), provided for use in scripts and build descriptions, since it "matches" builder.package_obj_path, builder.deploy_path, and so on.

checkouts_for_package(*pkg_label*)
Return a set of the checkouts that the given package depends upon

This only looks at *direct* dependencies (so if a package depends on something that in turn depends on a checkout that it does not directly depend on, then that indirect checkout will not be returned).

It does, however, expand wildcards.

deploy_path(*label*)
Where should deployment 'label' deploy to?

diagnose_unused_labels(*labels*, *arg*, *required_type=None*, *required_tag='*'*)
Concoct a useful report on why none of 'labels' is used.

We rely on 'labels' having been generated by our label_from_fragment() method, which means that all the labels will have the same type

We assume quite a lot of knowledge about how that method works.

effective_environment_for(*label*)
Return an environment which embodies the settings that should be used for the given label. It's the in-order merge of the output of `list_environments_for()`.

expand_release()
Expand the command line argument "_release"

expand_underscore_arg(*word*, *type_for_all=None*)
Given a command line argument ('word') that starts with an underscore, try to expand it to a list of labels.

If the argument is _all, then if 'type_for_all' is given, expand it to all labels of that type, and otherwise reject it.

Raises a GiveUp exception if the argument is not recognised.

expand_wildcards(*label*, *default_to_obvious_tag=True*)
Given a label which may contain wildcards, return a set of labels that match.

As per the normal definition of labels, the <type>, <name>, <role> and <tag> parts of the label may be wildcarded.

If default_to_obvious_tag is true, then if label has a tag of '*', it will be replaced by the "obvious" (final) tag for this label type, before any searching (so for a checkout: label, /checked_out would be used).

find_local_package_labels(*dir*, *tag*)
This is slightly horrible because if you're in a source checkout (as you normally will be), there could be several packages.

Returns a list of the package labels involved. Uses the given tag for the labels.

find_location_in_tree(*dir*)
Find the directory type and name of subdirectory in a repository. This is used by the find_local_package_labels method to work out which packages to rebuild

- dir - The directory to analyse

If nothing sensible can be determined, we return None. Otherwise we return a tuple of the form:

(DirType, label, domain_name)

where:

- 'DirType' is a utils.DirType value,
- 'label' is None or a label describing our location,
- 'domain_name' None or the subdomain we are in and

If 'label' and 'domain_name' are both given, they will name the same domain.

follow_build_desc_branch

follows_build_desc_branch

get_all_checkout_labels_below(*dir*)
Get the labels of all the checkouts in or below directory 'dir'

NOTE that this will not work if you are in a subdirectory of a checkout. It's not meant to. Consider using find_location_in_tree() to determine that, before calling this method.

get_build_desc_branch(*verbose=False*)
Return the current branch of the top-level build description.

(Returns None if the build description is not on a branch, or if its VCS does not support this operation.)

get_default_domain()

get_dependent_package_dirs(*label*)
Find all the dependent packages for label and return a set of the object directories for each. Mainly used as a helper function by `set_default_variables()`.

get_distribution()
Retrieve the current distribution name and target directory.

Raises GiveUp if there is no current distribution set.

get_domain_parameter(*domain*, *name*)

get_domain_parameters(*domain*)

get_environment_for(*label*)
Return the environment store for the given label, inventing one if necessary.

get_labels_in_default_roles()
Return a list of the package labels in the default roles.

get_parameter(*name*)
Returns the given domain parameter, or None if it wasn't defined.

include_domain(*domain_builder*, *domain_name*)
Import the builder domain_builder into the current builder, giving it domain_name.

We first import the db, then we rename None to domain_name in banned_roles

instruct(*pkg*, *role*, *instruction_file*, *domain=None*)
Register the existence or non-existence of an instruction file. If instruction_file is None, we unregister the instruction file.

•instruction_file - A db.InstructionFile object to save.

is_release_build()
Are we a release build (i.e., a build tree created by "muddle release")?

We look to see if there is a file called .muddle/Release

kill_label(*label*, *useTags=True*, *useMatch=True*)
Kill everything that matches the given label and all its consequents.

label_from_fragment(*fragment*, *default_type*)
A variant of Label.from_fragment that understands types and wildcards

In particular, it knows that:

1.packages have roles, but checkouts and deployments do not.

2.wildcards expand to their appropriate values

Returns a list of labels. This method does not check that all of the labels returned actually exist as targets in the dependency tree.

labels_for_role(*kind*, *role*, *tag*, *domain=None*)
Find all the target labels with the specified kind, role and tag and return them in a set.

If 'domain' is specified, also require the domain to match.

list_environments_for(*label*)
Return a list of environments that contribute to the environment for the given label.

Returns a list of triples (match level, label, environment), in order.

load_instructions(*label*)
Load the instructions which apply to the given label (usually a wildcard on a role, from a deployment) and return a list of triples (label, filename, instructionfile).

map_unifications(*source_list*)

mark_domain(*domain_name*)
Write a file that marks this directory as a domain so we don't mistake it for the root.

note_unification(*source*, *target*)

package_install_path(*label*)
Where should pkg install itself, by default?

package_obj_path(*label*)
Where should the package with this label build its object files?

packages_for_deployment(*dep_label*)
Return a set of the packages that the given deployment depends upon

This only looks at *direct* dependencies (so if a deployment depends on something that in turn depends on a package that it does not directly depend on, then that indirect package will not be returned).

It does, however, expand wildcards.

packages_using_checkout (*co_label*)
Return a set of the packages which directly use a checkout (this does not include dependencies)

print_banned_roles ()

resource_body (*file_name*)
Return the body of a resource as a string.

resource_file_name (*file_name*)

role_combination_acceptable_for_lib (*r1*, *r2*, *domain1=None*, *domain2=None*)
True only if (r1,r2) does not appear in the list of banned roles.

role_install_path (*role*, *domain=None*)
Where should this role find its install to deploy?

roles_do_not_share_libraries (*r1*, *r2*, *domain1=None*, *domain2=None*)
Assert that roles a and b do not share libraries

Either a or b may be * to mean wildcard

Add (r1,r2) to the list of role pairs that do not share their libraries.

set_default_variables (*label*, *store*)
Muddle defines a variety of environment variables which are available whilst a label is being built. The particular variables provided depend on the type of label being built, or the type of build.

Package labels are associated with muddle Makefiles, so any environment variable specific to a package label will be available within a muddle Makefile (i.e., commands such as "muddle build" work on package labels).

Unexpected section title.

```
All labels
----------
```

MUDDLE The muddle executable itself. This can be used in muddle Makefiles, for instance:

```
fred_objdir = $(shell $(MUDDLE) query objdir package:fred{base})
```

MUDDLE_ROOT The absolute path to the root of the build tree (where the '.muddle' and 'src' directories are).

MUDDLE_LABEL The label currently being built.

MUDDLE_KIND, **MUDDLE_NAME**, **MUDDLE_ROLE**, **MUDDLE_TAG**, **MUDDLE_DOMAIN** Broken-down bits of the label being built. Values will not exist if the label does not contain them (so if 'label' is a checkout label, `MUDDLE_ROLE` will not be set).

MUDDLE_OBJ Where we should build object files for this label - the `obj` directory for packages, the `src` directory for checkouts, and the `deploy` directory for deployments. See "muddle query objdir".

Unexpected section title.

```
Package labels
--------------
```

For package labels, we also set:

MUDDLE_OBJ_OBJ, MUDDLE_OBJ_INCLUDE, MUDDLE_OBJ_LIB, MUDDLE_OBJ_BIN
$(MUDDLE_OBJ)/obj, $(MUDDLE_OBJ)/include, $(MUDDLE_OBJ)/lib, $(MUDDLE_OBJ)/bin, respectively. Note that we do not *create* these directories - it is up to the muddle Makefile to do so.

MUDDLE_INSTALL Where we should install package files to.

MUDDLE_INSTRUCT A shortcut to the 'muddle instruct' command for this package. Essentially "$(MUDDLE) instruct $(MUDDLE_LABEL)"

MUDDLE_UNINSTRUCT A shortcut to the 'muddle uninstruct' command for this package. Essentially "$(MUDDLE) uninstruct $(MUDDLE_LABEL)"

MUDDLE_PKGCONFIG_DIRS A path suitable for passing to pkg-config to tell it to look only at packages this label is declared to be dependent on. It will be empty if the label doesn't have any dependencies.

MUDDLE_PKGCONFIG_DIRS_AS_PATH The same as MUDDLE_PKGCONFIG_DIRS, for historical reasons.

MUDDLE_INCLUDE_DIRS A space separated list of include directories, constructed from the packages that this label depends on. Names will have been intelligently escaped for the shell. Only directories that actually exist will be included.

Typically used in a muddle Makefile as:

```
CFLAGS += $(MUDDLE_INCLUDE_DIRS:%=-I%)
```

MUDDLE_LIB_DIRS A space separated list of library directories, constructed from the packages that this label depends on. Names will have been intelligently escaped for the shell. Only directories that actually exist will be included.

Typically used in a muddle Makefile as:

```
LDFLAGS += $(MUDDLE_LIB_DIRS:%=-L%)
```

MUDDLE_LD_LIBRARY_PATH The same values as in MUDDLE_LIB_DIRS, but with items separated by colons. This is useful for passing (as LD_LIBRARY_PATH) to configure scripts that try to look for libraries when linking test programs.

MUDDLE_KERNEL_DIR If any of the packages that this label depends on has a directory called kerneldir in its obj dir (so, in its own terms, $(MUDDLE_OBJ)/kerneldir), then we set this value to that directory. Otherwise it is not set. If there is more than one candidate, then the last found is used (but the order of search is not defined, so this would be confusing).

If the build tree is building a Linux kernel, it can be useful to build the kernel into a directory of this name.

MUDDLE_KERNEL_SOURCE_DIR Like MUDDLE_KERNEL_DIR, but it looks for a directory called kernelsource. The same comments apply.

Unexpected section title.

```
Deployment labels
-----------------
```

For deployment labels we also set:

MUDDLE_DEPLOY_FROM Where we should deploy from (probably just MUDDLE_INSTALL with the last component removed)

MUDDLE_DEPLOY_TO Where we should deploy to, if we're a deployment.

Unexpected section title.

```
Release build values
--------------------
```

When building in a release tree (typically by use of "muddle release") extra environment variables are set to allow the build to know useful information about the release. In a non-release build, these will all be set to "(unset)".

MUDDLE_RELEASE_NAME The release name.

MUDDLE_RELEASE_VERSION The release version.

MUDDLE_RELEASE_HASH The hash of the release stamp file. This acts as a useful unique identifier for a particular release, as it is calculated from the stamp file information describing all the release checkouts.

Two releases with the same name and version, but with different checkout information, will have different release hashes.

set_distribution(*name*, *target_dir*)
Set the current distribution name and target directory.

set_domain_parameter(*domain*, *name*, *value*)

set_parameter(*name*, *value*)

Set a domain parameter: Danger Will Robinson! This is a very odd thing to do - domain parameters are typically set by their enclosing domains. Setting your own is an odd idea and liable to get you into trouble. It is, however, the only way of communicating values back from a domain to its parent (and you shouldn't really be doing that either!)

setup_environment(*label*, *src_env*)
Modify src_env to reflect the environments which apply to label, in match order.

target_label_exists(*label*)
Return True if this label is a target.

If it is not, then we are not going to be able to build it.

Note that this method does not understand wildcards, so the match must be exact.

unify_environments(*source*, *target*)
Given a source label and a target label, find all the environments which might apply to source and make them also apply to target.

This is (slightly) easier than one might imagine ..

unify_labels(*source*, *target*)
Unify the 'source' label with/into the 'target' label.

Given a dependency tree containing rules to build both 'source' and 'target', this edits the tree such that the any occurrences of 'source' are replaced by 'target', and dependencies are merged as appropriate.

Free variables (i.e. wildcards in the labels) are untouched - if you need to understand that, see depend.py for quite how this works.

Why is it called "unify" rather than "replace"? Mainly because it does more than replacement, as it has to merge the rules/dependencies together. In retrospect, though, some variation on "merge" might have been easier to remember (if also still inaccurate).

uninstruct_all()

exception muddled.mechanics.**ErrorInBuildDescription**(*message=None*, *retcode=1*)
Bases: *muddled.utils.GiveUp*

We want to be able to distinguish this exception *in this module*

We don't expect anyone outside this module to care.

muddled.mechanics.**build_co_and_path_from_str**(*str*)
Turn a BuildDescription text into checkout name and inner path.

That is, we assume the string we're given (which was presumably read from a BuildDescription) is of the form:

<checkout-name>/<path-to-build-desc>

For instance:

```
>>> build_co_and_path_from_str('builds/01.py')
('builds', '01.py')
>>> build_co_and_path_from_str('strawberry/jam/toast.py')
('strawberry', 'jam/toast.py')
```

muddled.mechanics.**check_build_name**(*name*)
Check a build name for legality.

Raises a GiveUp exception if the name is not allowed.

muddled.mechanics.**dynamic_load_build_desc**(*builder*)
Dynamically load the build description for this builder.

Specifically, load the top-level build description. At the moment we do not provide support for (re)loading subdomain build descriptions.

(When each build description is first read, it is always the top-level build description of its own build.)

Note that the Python path (sys.path) will have the build description checkout directory added to its start, so that the build description can import things therefrom.

Returns the apropriate module

muddled.mechanics.**include_domain**(*builder*, *domain_name*, *domain_repo*, *domain_desc*)
Include the named domain as a sub-build of this builder.

- 'domain_name' is the name of the domain
- 'domain_repo' is the string defining the repository for the domain's build.
- 'domain_build_desc' is then the path to the domain's build description, within that.

If the domain has not yet been retrieved from 'domain_repo' (more specifically, if `domains/<domain_name>/.muddle/` doesn't yet exist), then it will be retrieved. This will normally happen when `muddle init` is done.

Note that, as a short-hand convenience, sub-domains are marked as such by having a `.muddle/am_subdomain` file. This will be created by `include_domain()` if necessary.

muddled.mechanics.**load_builder**(*root_path*, *muddle_binary*, *params=None*, *default_domain=None*)
Load a builder from the given root path.

This should only ever be called internally within muddle itself.

- 'root_path' is the path to the root of our build tree
- 'muddle_binary' is the path to our muddle binary. This is used when doing $(MUDDLE) in muddle Makefiles, and is not needed otherwise.

- If given, 'params' specifies the domain parameters to pass down to the new Builder instance we're creating - it maps domain names to dictionaries of parameters so that other domains can retrieve them. This is used within the calltree of include_domain().
- If given, 'default_domain' is the default domain name for this (sub) build tree. This is used by the "muddle unstamp" command, and the muddle_patch.py script.

muddled.mechanics.**minimal_build_tree**(*muddle_binary*, *root_path*, *repo_location*, *build_desc*, *desc_branch=None*, *versions_repo=None*)

Setup the very minimum of a build tree.

This should give a .muddle directory with its main files, but will not actually try to retrieve any checkouts.

muddled.mechanics.**run_release_from**(*builder*, *release_dir*)

Run the build descriptions "release_from()" function.

'release_dir' is the path to the directory where the release is being assembled, which will become the final release archive/tarball.

The function is called as:

```
release_from(builder, release_dir)
```

We only run the "release_from()" in the top-level build description.

Note that the Python path (sys.path) will have the build description checkout directory added to its start, so that the release_from() function itself can import things therefrom.

17.2.13 muddled.pkg

Note: *Package actions and other low level setup*

Routines for manipulating packages and checkouts.

class muddled.pkg.**ArchSpecificAction**(*underlying*, *arch*)

Bases: object

Allow an action to be invoked if and only if you're on the right architecture

build_label(*builder*, *label*)

class muddled.pkg.**ArchSpecificActionGenerator**(*arch*)

Bases: object

generate(*underlying*)

class muddled.pkg.**Deployment**

Bases: *muddled.depend.Action*

Represents a deployment. Deployments (typically) package code into release packages

build_label(*builder*, *tag*)

Whatever's needed to build the relevant tag for this deployment.

class muddled.pkg.**NoAction**

Bases: *muddled.depend.Action*

An action which does nothing - used largely for testing.

build_label(*builder*, *label*)

class muddled.pkg.**NullPackageBuilder**(*name*, *role*)
Bases: *muddled.pkg.PackageBuilder*

A package that does nothing.

This can be useful when a build wants to force some checkouts to be present (and checked out), but there is nothing to build in them. Examples include documentation and meta-information that is just being kept in the build tree so that it doesn't get lost.

Use the 'null_package' function to construct a useful instance.

Construct a package.

self.name The name of this package

self.deps The dependency set for this package. The dependency set contains mappings from role to (package, role). A role of '*' indicates a wildcard.

build_label(*builder*, *label*)

class muddled.pkg.**PackageBuilder**(*name*, *role*)
Bases: *muddled.depend.Action*

Describes a package.

Construct a package.

self.name The name of this package

self.deps The dependency set for this package. The dependency set contains mappings from role to (package, role). A role of '*' indicates a wildcard.

build_label(*builder*, *label*)

class muddled.pkg.**Profile**(*name*, *role*)
Bases: object

A profile ties together a role, a deployment and an installation directory. Profiles aren't actions - they modify the builder.

There are two things you can do to a profile: you can `assume()` it, in which case you build that profile, or you can `use()` it, in which case that profile's build results (if any) become available to you.

assume(*builder*)

use(*builder*)

class muddled.pkg.**VcsCheckoutBuilder**(*vcs*)
Bases: *muddled.depend.Action*

This class represents the actions available on a checkout.

'self.vcs' is the VCS handler which knows how to do version control operations on a checkout.

build_label(*builder*, *co_label*)

must_pull_before_commit(*builder*, *co_label*)
Must we update in order to commit? Only the VCS handler knows ..

muddled.pkg.**add_checkout_rules**(*builder*, *co_label*, *action*)
Add the standard checkout rules to a ruleset for a checkout with name co_label. 'action' should be an instance of VcsCheckoutBuilder, which knows how to build a checkout: label, depending on its tag.

muddled.pkg.**add_package_rules**(*ruleset*, *pkg_name*, *role_name*, *action*)
Add the standard package rules to a ruleset.

muddled.pkg.**append_env_for_package**(*builder*, *pkg_name*, *pkg_roles*, *name*, *value*, *domain=None*, *type=None*)

Set the environment variable name to value in the given package built in the given roles. Useful for customising package behaviour in particular roles in the build description.

muddled.pkg.**depend_across_roles**(*ruleset*, *pkg_name*, *role_names*, *depends_on_pkgs*, *depends_on_role*)

Register that pkg_name{role_name}'s preconfig depends on depends_on_pkg{depends_on_role} having been postinstalled.

muddled.pkg.**do_depend**(*builder*, *pkg_name*, *role_names*, *deps*)

Make pkg_name in role_names depend on the contents of deps.

deps is a list of 2-tuples (pkg_name, role_name)

If the role name is None, we depend on the pkg name in the role we're currently using, so `do_depend(a, ['b', 'c'], [ ('d', None) ])` leads to `a{b}` depending on `d{b}` and `a{c}` depending on `d{c}`.

If role_names is a string, we will implicitly convert it into the singleton list [role_names].

muddled.pkg.**do_depend_label**(*builder*, *pkg_name*, *role_names*, *label*)

Make pkg_name in role_names depend on the given label

If role_names is a string, we will implicitly convert it into the singleton list [role_names].

muddled.pkg.**null_package**(*builder*, *name*, *role*)

Create a Null package, a package that does nothing.

Uses NullPackageBuilder to construct our package, and then calls add_package_rules() to add the standard rules for a package.

Returns the new package instance.

Use like this:

```
# We have documentation in this checkout
checkouts.simple.relative(builder, co_name='docs')

# And we'd like it always to be checked out
# For this, we use a Null package that doesn't build itself
null_pkg = null_package(builder, name='docs', role='meta')
pkg.package_depends_on_checkout(builder.ruleset,
                                pkg_name='docs', role_name='meta',
                                co_name='docs')

# And add that to our default roles
builder.add_default_role('meta')
```

muddled.pkg.**package_depends_on_checkout**(*ruleset*, *pkg_name*, *role_name*, *co_name*, *action=None*)

Make the given package depend on the given checkout

- ruleset - The ruleset to use - builder.ruleset, for example.
- pkg_name - The package which depends.
- role_name - The role which depends. Can be '*' for a wildcard.
- co_name - The checkout which this package and role depends on.
- action - If non-None, specifies an Action to be invoked to get from the checkout to the package preconfig. If you are a normal (outside muddle itself) caller, then you will normally leave this None unless you are doing something deeply weird.

muddled.pkg.**package_depends_on_packages**(*ruleset*, *pkg_name*, *role*, *tag_name*, *deps*)
Make pkg_name depend on the list in deps.

pkg_name's tag_name tag ends up depending on the deps having been installed - this can be PreConfig or Built, depending on whether you need that dependency to configure yourself or not.

muddled.pkg.**prepend_env_for_package**(*builder*, *pkg_name*, *pkg_roles*, *name*, *value*, *domain=None*, *type=None*)
Set the environment variable name to value in the given package built in the given roles. Useful for customising package behaviour in particular roles in the build description.

muddled.pkg.**set_checkout_vcs_option**(*builder*, *co_label*, ***kwargs*)
Sets extra VCS options for a checkout (identified by its label).

For reasons mostly to do with how stamping/unstamping works, we require option values to be either boolean, integer or string.

For example:

```
pkg.set_checkout_vcs_option(builder, depend.checkout('kernel-source'),
                            shallow_checkout=True, something_else=99)
```

This is a wrapper around:

```
builder.db.set_checkout_vcs_option(depend.checkout('kernel-source'),
                                   'shallow', True)
builder.db.set_checkout_vcs_option(depend.checkout('kernel-source'),
                                   'something_else', 99)
```

"muddle help vcs <name>" should document the available options for the version control system <name> (see "muddle help vcs" for the supported version control systems).

muddled.pkg.**set_env_for_package**(*builder*, *pkg_name*, *pkg_roles*, *name*, *value*, *domain=None*)
Set the environment variable name to value in the given package built in the given roles. Useful for customising package behaviour in particular roles in the build description.

17.2.14 muddled.repository

Note: *Repository definition and handling*

A new way of handling repositories

class muddled.repository.**Repository**(*vcs*, *base_url*, *repo_name*, *prefix=None*, *prefix_as_is=False*, *suffix=None*, *inner_path=None*, *revision=None*, *branch=None*, *handler='guess'*, *push=True*, *pull=True*)
Bases: object

The representation of a single repository.

At minimum, a repository is specified by a VCS, a base URL, and a checkout name. For instance:

```
>>> r = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
...                'builds')
```

As you might expect, it remembers what you told it:

```
>>> r.vcs
'git'
>>> r.base_url
```

```
    'ssh://git@project-server/opt/kynesim/projects/042/git/'
    >>> r.repo_name
    'builds'
```

but it also calculates the full URL for accessing the repository:

```
    >>> r.url
    'ssh://git@project-server/opt/kynesim/projects/042/git/builds'
```

(this is calculated when the object is created because it is expected to be queried many times, and a Repository object is intended to be treated as immutable, even though we don't *enforce* that).

> Why do we split the repository into a base URL and a checkout name? Mainly because we assume that we are going to have more than one repository with the same base URL, so we can use the `copy_with_changes` method to construct new instances without having to constantly propagate the base URL explicitly.

Note that it is possible for some project hosts to be treated differently - for instance, we have a built-in rule for google code projects:

```
    >>> g1 = Repository('git', 'https://code.google.com/p/grump', 'default')
    >>> g1.url
    'https://code.google.com/p/grump'
    >>> g2 = Repository('git', 'https://code.google.com/p/grump', 'wiki')
    >>> g2.url
    'https://code.google.com/p/grump.wiki'
```

which is detected automatically, and the appropriate handler used. If we don't want that, we can explicitly say so:

```
    >>> g3 = Repository('git', 'https://code.google.com/p/grump', 'default',
    ...                handler=None)
    >>> g3.url
    'https://code.google.com/p/grump/default'
```

or we can ask for it explicitly by name:

```
    >>> g4 = Repository('git', 'https://code.google.com/p/grump', 'default',
    ...                handler='code.google.com')
    >>> g4.url
    'https://code.google.com/p/grump'
    >>> g4.handler
    'code.google.com'
```

The default handler name is actually 'guess', which tries to decide by looking at the repository URL and the VCS - basically, if the repository starts with "https://code.google.com/p/" and the VCS is 'git', then it will use the 'code.google.com' handler, and otherwise it won't. The 'handler' value reflects which handler was actually used:

```
    >>> g2.handler
    'code.google.com'
    >>> print g3.handler
    None
    >>> g4.handler
    'code.google.com'
```

Sometimes, we need some extra "path" between the repository base path and the checkout name. For instance:

```
    >>> r = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
    ...                'busybox-1.18.5', prefix='core')
```

```
>>> r.base_url
'ssh://git@project-server/opt/kynesim/projects/042/git/'
>>> r.url
'ssh://git@project-server/opt/kynesim/projects/042/git/core/busybox-1.18.5'
```

By default, that prefix is added in as a pseudo-file-path - i.e., with '/' before and after it. If that's the wrong thing to do, then specifying 'use_prefix_as_is=True' will cause the 'prefix' to be used as-is (does anyone actually support this sort of thing?):

```
>>> r = Repository('git', 'http://example.com',
...                'busybox-1.18.5', prefix='?repo=', prefix_as_is=True)
>>> r.base_url
'http://example.com'
>>> r.url
'http://example.com?repo=busybox-1.18.5'
```

Bazaar, in particular, sometimes wants to add trailing text to the checkout name, commonly to indicate a branch (bzr doesn't really support branches as such, but instead sometimes uses conventions on how different repositories are named). So, for instance:

```
>>> r = Repository('bzr', 'ssh://bzr@project-server/opt/kynesim/projects/042/bzr/',
...                'repo42', suffix='/fixit_branch')
>>> r.url
'ssh://bzr@project-server/opt/kynesim/projects/042/bzr/repo42/fixit_branch'
```

Note that we had to specify the '/' in 'suffix', it wasn't assumed.

Git servers sometimes want us to put '.git' on the end of a checkout name. This can be done using the same mechanism:

```
>>> r = Repository('git', 'git@github.com:tibs', 'withdir', suffix='.git')
>>> r.url
'git@github.com:tibs/withdir.git'
```

(although note that github will cope with or without the '.git' at the end). Note that we had to specify the '.' in '.git', it wasn't assumed.

Subversion allows retrieving *part* of a repository, by specifying the internal path leading to the entity to be retrieved. So, for instance:

```
>>> r = Repository('svn', 'ssh://svn@project-server/opt/kynesim/projects/042/svn',
...                'all_our_code', inner_path='core/busybox-1.18.4')
>>> r.url
'ssh://svn@project-server/opt/kynesim/projects/042/svn/all_our_code/core/busybox-1.18.4'
```

It is not intended that <inner_path> and <suffix> be used together, and the result if they are is not guaranteed.

Finally, it is possible to specify a revision and branch. These are both handled as strings, with no defined interpretation (and are not always relevant to a particular VCS - see the discussion of Bazaar above).

Creating a new Repository instance

- 'vcs' is the (short name) for the Version Control System being used to access this repository. For instance, "git" or "svn".
- 'base_url' is the first part of the URL for the repository. This is separated out because it is common for different repositories to share the first part of their URL, not because it necessarily has any greater meaning.
- 'repo_name' is the part that names this particular repository. It is often the same as the name of the checkout using this repository (but that depends on how the reposiory is accessed).

- 'prefix' is a string to put between the 'base_url' and 'repo_name', when constructing the full repository URL.
- if 'prefix_as_is' is true, then 'prefix' is inserted between 'base_url' and 'repo_name' exactly as it is given, otherwise it is delimited by "/" characters. It is ignored if 'prefix' is None.
- 'suffix' is a string to put after the 'repo_name' when constructing the full repository URL.
- 'inner_path' is used to specify a path *inside* the repository, for version control systems that allow this (typically Subversion).
- 'revision' is the revision to use. This should always be a string, regardless of what the VCS expects. It may look like an integer (e.g., "123") or an expression ("-r123" or "date:20120101") depending on the VCS.
- 'branch' is the branch to use.
- 'handler' is either None, or "guess" (the default) or the name of a registered handler for constructing the full repository URL if the normal mechanisms are not adequate.
- 'push' is true if we can push to this repository. The default is True.
- 'pull' is true if we can pull from this repository. The default is True.

If there is no handler in action, the full repository URL is thus either:

- <base_url>/<prefix>/<repo_name>/<inner_path><suffix> or
- <base_url><prefix><repo_name>/<inner_path><suffix> or

depending on the value of 'prefix_as_is', and with values that are None being turned into empty strings. Note that it is not really intended that both <inner_path> and <suffix> be used on the same repository.

copy_with_changed_branch(*branch*, *revision=None*)

Return a Repository that differs only in its branch (and revision).

A simple copy is taken, and then the branch and revision are changed. Typically, the revision is just unset.

Note that we don't check that you don't set the revision to the same value again, although it seems unlikely to be sensible in most version control systems to do this.

For instance:

```
>>> r = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
...                'builds', branch='fred', revision='23')
>>> r
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', revisi
>>> s = r.copy_with_changed_branch('jim')
>>> s
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', branch
>>> s = r.copy_with_changed_branch('jim', revision='99')
>>> s
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', revisi
```

Note that the copy will have the same value of 'from_url_string' as the original.

copy_with_changed_revision(*revision*)

Return a Repository that differs only in its revision.

A simple copy is taken, and then the revision is changed. This is used in version stamping.

For instance:

```
>>> r = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
...                'builds', revision='23')
>>> r
```

```
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', revisi
>>> s = r.copy_with_changed_revision('27')
>>> s
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', revisi
```

Note that the copy will have the same value of 'from_url_string' as the original.

copy_with_changes(*repo_name*, *prefix=None*, *suffix=None*, *inner_path=None*, *revision=None*, *branch=None*, *push=None*, *pull=None*)

Return a new instance based on this one.

A simple copy is taken, and then any amendments are made to it.

'repo_name' must be given.

This is expected to be (typically) useful for working out a repository relative to another (for instance, relative to the default, builds, repository). For instance:

```
>>> r = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
...                'builds')
>>> r
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds')
>>> s = r.copy_with_changes('fred')
>>> s
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'fred')
>>> s = r.copy_with_changes('jim', suffix='bob')
>>> s
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'jim', suffix='b
```

Thus it does *not* default to using the branch or revision from the original object:

```
>>> x = Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/',
...                'builds', revision='123', branch='fred')
>>> x
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'builds', revisi
>>> x.copy_with_changes('jim')
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'jim')
>>> x.copy_with_changes('jim', branch='thrim')
Repository('git', 'ssh://git@project-server/opt/kynesim/projects/042/git/', 'jim', branch='t
```

Looking at a google code example:

```
>>> g = Repository('git', 'https://code.google.com/p/raw-cctv-replay', 'builds')
>>> g.url
'https://code.google.com/p/raw-cctv-replay.builds'
>>> g2 = g.copy_with_changes('stuff')
>>> g2.url
'https://code.google.com/p/raw-cctv-replay.stuff'
```

Note that, by its nature, calling this will result in a Repository instance that has 'from_url_string' unset.

default_path()

Return the default repository path, calculated from all the parts.

It is returned in a manner suitable for passing to the appropriate command line tool for cloning the repository.

static from_url(*vcs*, *repo_url*, *revision=None*, *branch=None*, *push=True*, *pull=True*)

Construct a Repository instance from a URL.

- 'vcs' is the version control system ('git', 'svn', etc.)

- 'repo_url' is the complete URL used to access the repository
- 'revision' and 'branch' are the revision and branch to use, just as for the normal constructor.
- 'push' and 'pull' indicate whether one can push to and/or pull from the repository, again as for the normal constructor.

We make a good guess as to the 'checkout name', assuming it is the last component of the path in the URL. But, of course, no path handler gets called, so the result may not be quite what is expected.

Thus:

```
>>> r = Repository.from_url('git', 'http://example.com/fred/jim.git?branch=a')
>>> r
Repository.from_url('git', 'http://example.com/fred/jim.git?branch=a')
>>> r.url
'http://example.com/fred/jim.git?branch=a'
>>> r.repo_name
'jim.git'
>>> r.suffix
'?branch=a'
```

and even:

```
>>> f = Repository.from_url('file', 'file:///home/tibs/versions')
>>> f
Repository.from_url('file', 'file:///home/tibs/versions')
>>> f.url
'file:///home/tibs/versions'
>>> f.repo_name
'versions'
```

We do, however, set the 'from_url_string' value to the original URL as given:

```
>>> f.from_url_string
'file:///home/tibs/versions'
>>> r.from_url_string
'http://example.com/fred/jim.git?branch=a'
```

which allows one to tell definitively what URL was requested, and also distinguishes this from a Repository instance created in the normal manner (in a normal Repository instance. 'from_url_string' will be None).

Also, the handler will always be None for a Repository created with this method:

```
>>> print f.handler
None
>>> print r.handler
None
```

static `get_path_handler`(*vcs*, *starts_with*)
Retrieve the handler for 'vcs' and 'starts_with'.

Returns None if there isn't one (which is actually the primary reason for providing this function).

`path_handlers` = {('git', 'code.google.com'): <function google_code_handler at 0x7f44a4171c80>}

static `register_path_handler`(*vcs*, *starts_with*, *handler*)
Register a special handler for a particular repository location.

- 'vcs' is the short name of the VCS we're interested in, as returned by split_vcs_url().
- 'starts_with' is what a repository 'base_url' must start with in order for us to use the handler

- 'handler' is a function that takes a Repository instance and returns a path suitable for putting into the Repository instance's 'url' value method.

For instance, the google code handler for git might be registered using:

```
Repository.register_path_handle('git', 'https://code.google.com/p/',
                                google_code_git_handler)
```

The handler is associated with both 'vcs' and 'starts_with'. Calling this function again with the same 'vcs' and 'starts_with', but a different 'handler', will silently override the previous entry.

same_ignoring_revision(*other*)
Requires equality except for the revision.

muddled.repository.**add_upstream_repo**(*builder*, *orig_repo*, *upstream_repo*, *names*)
Add an upstream repo to 'orig_repo'.

- 'orig_repo' is the original Repository that we are adding an upstream for.
- 'upstream_repo' is the upstream Repository. It is an error if that repository is already an upstream of 'orig_repo'.
- 'names' is a either a single string, or a sequence of strings, that can be used to select this (and possibly other) upstream repositories.

Upstream repository names must be formed of A-Z, a-z, 0-9 and underscore or hyphen.

A convenience wrapper around 'builder.db.add_upstream_repo'.

muddled.repository.**get_checkout_repo**(*builder*, *co_label*)
Returns the Repository instance for this checkout label

A convenience wrapper around 'builder.db.get_checkout_repo'.

muddled.repository.**get_upstream_repos**(*builder*, *orig_repo*, *names=None*)
Retrieve the upstream repositories for 'orig_repo'

If 'names' is given, it must be a sequence of strings, in which case only those upstream repositories annotated with any of the names will be returned.

Returns a set of upstream repositories. This will be empty if there are no upstream repositories for 'orig_repo', or none with any of the names in 'names' (if given).

A convenience wrapper around 'builder.db.get_upstream_repos'.

muddled.repository.**google_code_handler**(*repo*)
A repository path handler for google code projects.

Google code project repository URLs are of the form:

- https://code.google.com/p/<project> for the default repository
- https://code.google.com/p/<project>.<repo> for any other repository

This is registered as the "code.google.com" handler for "git".

17.2.15 muddled.rewrite

This module provides for rewrites of .al and pkgconfig files to reflect the realities of existing inside a muddle build tree.

Specifically, it allows you to rewrite the .la and .pkgconfig files from an autoconf'd package (typically created with make install DESTDIR=$(MUDDLE_OBJ_DIR)) so that future packages will pick up libraries in the right places.

It is, of course, up to those packages to not use -rpath-link to force your build tree locations into the target filesystem.

muddled.rewrite.**fix_up_pkgconfig_and_la**(*builder*, *dir*, *subdir=None*, *libPath=None*, *includePath=None*, *execPrefix=None*)

Given a directory, dir, in which there may be .pc and .la files lurking, identify the .pc and .la files and rewrite them.

- subdir, if present, is the subdirectory to search.
- libPath, if present, is the directory in which the target libraries are installed (typically dir/lib)
- includePath, if present, is where the target include files are installed (typically dir/include).
- execPrefix is where the package will be installed.

  This is a bit tricky for us, since we're cross-compiling: we will actually define it, by default, to be dir, since in practice most packages want this (they use it to locate tools, not -rpath-link). But you can set it to whatever you like :-)

For the moment, we work by rewriting.

- In a .la file:
  - libdir - gets prefixed with 'dir'
- In a .pc file:
  - prefix -> dir
  - exec_prefix -> execDir
  - libdir -> libPath (if present)
  - includedir -> includePath (if present)

muddled.rewrite.**parse_line**(*l*)

Parse the given line, returning (key, value) or (None, None) if it wasn't valid

muddled.rewrite.**subst_la**(*builder*, *current*, *dir*, *libPath*, *includePath*, *execPrefix*)

Substitute a .la file.

muddled.rewrite.**subst_pc**(*builder*, *current*, *dir*, *libPath*, *includePath*, *execPrefix*)

Substitute a pkgconfig (.pc) file.

17.2.16 muddled.rrw

Note: *A general collection of toolchain utilities, named after Richard for historical reasons*

rrw's development library of _experimental_ muddle entry points. This file will go away in the next major release of muddle - in the meantime, it provides a useful library of code for reuse.

muddled.rrw.**append_env**(*builder*, *roles*, *bindings*, *domain=None*, *setType=None*)

Set environment variable <var> = <value> for every package in the given roles

Bindings is a series of (<var>, <value>) pairs.

If setType is specified, we set the type of the environment variable to one of the types in env_store.py: the most popular of these is EnvType.SimpleValue which marks the variable as not a path-type variable.

muddled.rrw.**append_to_path**(*builder*, *roles*, *val*)

Append the given value to the PATH for the given roles

muddled.rrw.**apt_get_install**(*builder*, *pkg_list*, *required_by*, *pkg_name='dev_pkgs'*, *role='dev_pkgs'*)

Make sure the host has installed the given packages.

Uses `apt-get install` (or equivalent).

- 'pkg_list' is the list of the names of the packages to check for. The names should be as they are expected by `apt-get`.
- 'required_by' is the list of roles that require the development packages to be installed.

This is essentially a convenience wrapper for `muddled.pkgs.aptget.medium()`, with sensible default values for 'pkg_name' and 'role'.

For instance:

```
apt_get_install(builder, ["bison", "flex", "libtool"], ["text", "graphics"])
```

muddled.rrw.**build_role_on_architecture**(*builder*, *role*, *arch*)

Wraps all the actions in a given role inside an ArchSpecificAction generator.

This requires all the actions in that role ("package:{<*role*>}/") to be built on architecture <arch>.

muddled.rrw.**build_with_helper**(*builder*, *helpers*, *pkg_name*, *checkout*, *roles*, *makefileName=None*, *co_dir=None*, *repoRelative=None*, *rev=None*)

Builds a package called 'pkg_name' from a makefile in a helpers checkout called 'helpers', involving the use of the checkout 'checkout', which is a relative checkout with optional second level name co_dir, repo relative name repoRelative, and revision rev.

In other words, declares that 'pkg_name' in the given 'roles' will be built with the Makefile called:

<helpers>/<makefileName>

If 'co_dir' is None, this will be checked out using checkouts.simple.relative(), otherwise it will be checked out using checkouts.twolevel.relative(). The 'co_dir', 'repoRelative' and 'rev' arguments will be used in the obvious ways.

muddled.rrw.**get_domain_param**(*builder*, *domain*, *name*)

A convenience wrapper around builder.get_domain_parameter().

It's slightly shorter to type...

muddled.rrw.**package_requires**(*builder*, *in_pkg*, *pkg_roles*, *reqs*)

Register the information that 'in_pkg', built in all of 'pkg_roles', require (depends on) 'reqs', which is a list of pairs (<package_name>, <role>)

muddled.rrw.**packages_use_role**(*builder*, *pkgs*, *in_role*, *use_role*, *domain=None*)

Specifies that one role uses the results of another role; this is most often used to allow roles built for a target to use roles built for the host.

We would normally want a role_uses_role(), but this tends to lead to undesirable excessive rebuilding of entire roles when the host tools change.

muddled.rrw.**set_domain_param**(*builder*, *domain*, *name*, *value*)

A convenience wrapper around builder.set_domain_parameter().

It's slightly shorter to type...

muddled.rrw.**set_env**(*builder*, *roles*, *bindings*, *domain=None*)

Set environment variable <var> = <value> for every package in the given roles

Bindings is a series of (<var>, <value>) pairs.

muddled.rrw.**set_global_package_env**(*builder, name, value, roles=['*']*)
Set an environment variable 'name = value' for all of the named roles.

(The default sets the environment variable globally, i.e., for all roles.)

muddled.rrw.**set_gnu_tools**(*builder, roles, env_prefix, prefix, cflags=None, ldflags=None, asflags=None, archspec=None, archname=None, archroles=['*'], domain=None, dirname=None, cppflags=None, cxxflags=None*)
This is a utility function which sets up the given roles to use the given compiler prefix (typically the empty string "" for host tools, or something like "arm-linux-none-gnueabi-" for ARM)

Environment variables like:

<env_prefix>GCC

end up with values like:

<prefix>gcc

1. If 'env_prefix' is not None, then we set up the following environment variables:
   - <env_prefix>CC is <prefix>gcc
   - <env_prefix>CXX is <prefix>g++
   - <env_prefix>CPP is <prefix>gpp
   - <env_prefix>LD is <prefix>ld
   - <env_prefix>AR is <prefix>ar
   - <env_prefix>AS is <prefix>as
   - <env_prefix>NM is <prefix>nm
   - <env_prefix>OBJDUMP is <prefix>objdump
   - <env_prefix>OBJCOPY is <prefix>objcopy
   - <env_prefix>RANLIB is <prefix>ranlib
   - <env_prefix>PFX is the <prefix> itself
   - if 'archspec' is not None, <env_prefix>ARCHSPEC is set to it
   - if 'cflags' is not None, <env_prefix>CFLAGS is set to it
   - if 'cppflags' is not None, <env_prefix>CPPFLAGS is set to it
   - if 'cxxflags' is not None, <env_prefix>CXXFLAGS is set to it
   - if 'ldflags' is not None, <env_prefix>LDFLAGS is set to it
   - if 'asflags' is not None, <env_prefix>ASFLAGS is set to it
   - if 'dirname' is not None, <env_prefix>COMPILER_TOOLS_DIR is set to it

   in all of the 'roles' named.

   Note that it is perfectly possible to have 'env_prefix' as the empty string ("") if one wishes to set ${CC}, etc.
2. If 'archname' is not None, we also set <archname>_<XX> to the same set of values, in all of the roles named in 'archroles'. Thus roles which are, for instance, building for the host can access toolchains for other processors in the system.

For instance:

```
    set_gnu_tools(builder, ['tools'], '', HOST_TOOLS_PREFIX,
                  archname='HOST', archroles=['firmware'])

    set_gnu_tools(builder, ['firmware'], '', ARM_TOOLS_PREFIX)
```

After this:

- in role 'tools' ${CC} will refer to the version of gcc in HOST_TOOLS_PREFIX.
- in role 'firmware', ${CC} will refer to the version of gcc in ARM_TOOLS_PREFIX, and ${HOST_CC} will refer to the "host" gcc in HOST_TOOLS_PREFIX.

muddled.rrw.**setup_helpers**(*builder*, *helper_name*)

Set up a helper checkout to be used in subsequent calls to build_with_helper

Basically a wrapper around:

```
    checkouts.simple.relative(builder, helper_name, helper_name)
```

muddled.rrw.**setup_tools**(*builder*, *roles_that_use_tools=['*']*, *tools_roles=['tools']*, *tools_dep='tools'*, *tools_path_env='TOOLS_PATH'*, *tools_install=None*)

Setup the "post-build" environment for particular roles.

This sets up the deployment paths for roles, and also the runtime environment variables. This can typicaly be used to distinguish roles which run in the host environment (using programs and shared libraries from the host) and roles which run in the environment being built (using programs and shared libraries from the muddle deployment directories).

- 'roles_that_use_tools' is a list of the roles that will be *using* the named tools. So, if the tools are GCC and its friends, this would typically be all of the roles that contain things to be built with (that) GCC. These roles will depend on the tools being deployed.
- 'tools_roles' is a list of the roles that *provide* the tools. These do not share libraries with any other roles (so, GCC on the host does not use the same libraries as the roles that will be installed on the target).
- 'tools_dep' is the deployment name for this set-of-tools. It corresponds to a label "deployment:<name>{}/deployed" in the ruleset.
- 'tools_path_env' is the name of an environment variable that will be set to tell each of the roles in 'roles_that_use_tools' about the location of the 'tools_dep' deployment.
- 'tools_install' is currently ignored.

Specifically:

1. Register a tools deployment called 'tools_dep', used by the 'roles_that_use_this', and provided by packages in the 'tools_roles'.
2. In each of the 'roles_that_use_tools', set the environment variable 'tools_path_env' to the deployment path for 'tools_dep'.
3. In each of the 'roles_that_use_tools', amend the following environment variables as follows, where "$role_deploy" is the deployment path for 'tools_dep':
   - LD_LIBRARY_PATH - Prepend $role_deployl/lib
   - PATH - Append $role_deploy/bin
   - PKG_CONFIG_PATH - Prepend $role_deploy/lib/pkgconfig
   - <role>_TOOLS_PATH (where <role> is upper-cased) - Prepend $role_deploy/bin
4. Tell each of the 'tools_roles' that it does not share libraries with any other roles.

17.2.17 muddled.subst

Substitutes a file with query parameters. These can come from environment variables or from an (optional) XML file.

Queries are a bit like XPath:

```
/elem/elem ...
```

An implicit `::text()` is appended so you get all the text in the specified element.

class muddled.subst.**PushbackInputStream**(*str*)
Bases: object

A pushback input stream based on a string. Used in our recursive descent parser

get_line(*line_no*)
Return line 'line_no'. Line numbers start at 1.

next()

peek()

print_what_we_just_read()

push_back(*c*)

report()

class muddled.subst.**TreeNode**(*in_type*, *input_stream*)
Bases: object

A TreeNode contains itself, followed by all its children, so this is essentially a left tree.

ContainerType = 'container'

InstructionType = 'instruction'

StringType = 'string'

append_child(*n*)

append_children(*xml_doc*, *env*, *output_list*)

echo(*xml_doc*, *env*, *output_list*)

eval(*xml_doc*, *env*, *output_list*)
Evaluate this node with respect to xml_doc, env and place your output in output_list - a list of strings.

eval_str(*xml_doc*, *env*)
Evaluate this node and return the result as a string

fnval(*xml_doc*, *env*, *output_list*)

ifeq(*xml_doc*, *env*, *output_list*, *polarity*)

set_fn(*fn_name*, *params*, *rest*)
fn_name is the name of the function params and rest are lists of nodes

set_string(*inStr*)

set_val(*v*)
v is the value which should be evaluated to get the value to evaluate.

val(*xml_doc*, *env*, *output_list*)

muddled.subst.**flatten_literal_node**(*in_node*)
Flatten a literal node into a string. Raise GiveUp if we, um, fail.

muddled.subst.**get_text_in_xml_node**(*node*)
Given an XML node, collect all the text in it.

muddled.subst.**parse_document**(*input_stream*, *node*, *end_chars*, *has_escapes*)
Parse a document into a tree node. Ends with end_char (which may be -1)

Leaves the input stream positioned at end_char.

muddled.subst.**parse_instruction**(*input_stream*, *node*)
An instruction ends at }, and contains:

fn:<name>(<args>, ..) rest}

or

<stuff>}

muddled.subst.**parse_literal**(*input_stream*, *echars*)
Given a set of end chars, parse a literal.

muddled.subst.**parse_param**(*input_stream*, *node*, *echars*)
Parse a parameter: may be quoted (in which case ends at ") else ends at echars

muddled.subst.**query_result**(*keys*, *doc_node*)
Given a list of keys and a document node, return the XML node which matches the query, or None if there isn't one.

muddled.subst.**query_string_value**(*xml_doc*, *env*, *k*)
Given a string-valued query, work out what the result was

muddled.subst.**skip_whitespace**(*in_stream*)
Skip some whitespace

muddled.subst.**split_query**(*query*)
Split a query into a series of keys suitable to be passed to query_result().

muddled.subst.**subst_file**(*in_file*, *out_file*, *xml_doc*, *env*)

muddled.subst.**subst_str**(*in_str*, *xml_doc*, *env*)
Substitute `${...}` in in_str with the appropriate objects - if XML doesn't match, try an environment variable.

Unescape `$${...}` in case someone actually wanted *${...}'* in the output.

Functions can be called with: ${fn:NAME(ARGS) REST}

name can be: ifeq(query,value) - in which case REST is substituted. val(query) - just looks up query.

17.2.18 muddled.utils

Note: *Core utilities used throughout muddle*

Muddle utilities.

class muddled.utils.**Choice**(*choices*)
Bases: object

A choice "sequence".

A choice sequence is:

- a string or dictionary, the only choice. For instance:

```
choice = Choice("libxml-dev2")
assert choice.choose('any string at all') == 'libxml-dev2)
```

- a sequence of the form [(pattern, value), ...]; that is a sequence of one or more '(pattern, value)' pairs, where each 'pattern' is an fnmatch pattern (see below) and each 'value' is a string or dict.

  The patterns are compared to 'what_to_match' in turn, and if one matches, the corresponding 'value' is returned. If none match, a ValueError is raised.

  For instance:

```
choice = Choice([ ('ubuntu-12.*', 'package-v12'),
                  ('ubuntu-1?.*', 'package-v10') ])
try:
    match = choice.choose_to_match_os()
except ValueError:
    print 'No package matched OS %s'%get_os_version_name()
```

- a sequence of the form [(pattern, value), ..., default]; that is a sequence of one or more pairs (as above), with a final "default" value, which must be a string or dict or None.

  The patterns are compared to 'what_to_match' in turn, and if one matches, the corresponding 'value' is returned. If none match, the final default value is returned. None is allowed so the caller can easily tell that no choice was actually made.

  choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), 'package-v09'])

  Definition list ends without a blank line; unexpected unindent.

  # 'match' will always have a "sensible" value match = choice.choose_to_match_os()

  choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), None])

  Definition list ends without a blank line; unexpected unindent.

  match = choice.choose_to_match_os() if match is None:

  Unexpected indentation.

  return # We know there was no given value

- as a result of the previous, we also allow [default], although [None] is of questionable utility.

  choice = Choice(["libxml-dev2"]) assert choice.choose('any string at all') == 'libxml-dev2)

  choice = Choice(["None"]) assert choice.choose('any string at all') is None

  (although that latter is the only way of "forcing" a Choice that will always return None, if you did need such a thing...)

Why not just use a list of pairs (possibly with a default string at the end, essentially just what we pass to Choice)? Well, it turns out that if you want to do something like:

```
pkgs.apt_get(["fromble1",
              Choice([ ('ubuntu-12.*', 'fromble'),
                       ('ubuntu-11.*', 'alex'),
                       None ]),
              "ribbit",
              Choice([ ('ubuntu-12.*', 'package-v12'),
                       ('ubuntu-1?.*', 'package-v10'),
                       'package-v7' ]),
```

```
                    "libxml-dev2",
                ])
```

it is (a) really hard to type it right if it is just nested sequences, and (b) terribly hard to give useful error messages when the user doesn't get it right. There are already enough brackets of various sorts, and if we don't have the "Choice" delimiters, it just gets harder to keep track.

choose(*what_to_match*)
Try to match 'what_to_match', and return the appropriate value.

Raises ValueError if there is no match.

Returns None if (and only if) that was given as a fall-back default value.

choose_to_match_os(*version_name=None*)
A special case of 'decide' to match OS id/version

If 'version_name' is None, then it looks up the system 'id' (the "name" of the OS, e.g., "ubuntu"), and 'version' of the OS (e.g., "12.10") from /etc/os-release, and concatenates them separated by a space (so "ubuntu 12.10").

It returns the result of calling:

choose(version_name)

So, on an Ubuntu system (which also includes a Linux Mint system, since its /etc/os-release identifies it as the underlying Ubuntu system), one might do:

choice = Choice([('ubuntu-12.*', 'package-v12'), ('ubuntu-1?.*', 'package-v10'), 'package-v7'])

Definition list ends without a blank line; unexpected unindent.

choice.choose_to_match_os()

to choose the appropriate version of "package" depending on the OS.

muddled.utils.**Error**
alias of *MuddleBug*

muddled.utils.**Failure**
alias of *GiveUp*

exception muddled.utils.**GiveUp**(*message=None*, *retcode=1*)
Bases: exceptions.Exception

Use this to indicate that something has gone wrong and we are giving up.

This is not an error in muddle itself, however, so there is no need for a traceback.

By default, a return code of 1 is indicated by the 'retcode' value - this can be set by the caller to another value, which __main__.py should then use as its return code if the exception reaches it.

retcode = 1

class muddled.utils.**HashFile**(*name*, *mode='r'*, *ignore_comments=False*, *ignore_blank_lines=False*)
Bases: object

A very simple class for handling files and calculating their SHA1 hash.

We support a subset of the normal file class, but as lines are read or written, we calculate a SHA1 hash for the file.

Optionally, comment lines and/or blank lines can be ignored in calculating the hash, where comment lines are those starting with a '#', or whitespace and a '#', and blank lines are those which contain only whitespace (which includes empty lines).

Open the file, for read or write.

- 'name' is the name (path) of the file to open
- 'mode' is either 'r' (for read) or 'w' (for write). If 'w' is specified, then if the file doesn't exist, it will be created, otherwise it will be truncated.
- if 'ignore_comments' is true, then lines starting with a '#' (or whitespace and a '#') will not be used in calculating the hash.
- if 'ignore_blank_lines' is true, then lines that are empty (zero length), or contain only whitespace, will not be used in calculating the hash.

Note that this "ignore" doesn't mean "don't write to the file", it just means "ignore when calculating the hash".

close()
Close the file.

hash()
Return the SHA1 hash, calculated from the lines so far, as a hex string.

next()

readline()
Read the next line from the file, and add it to the SHA1 hash as well.

Returns '' if there is no next line (i.e., EOF is reached).

write(*text*)
Write the give text to the file, and add it to the SHA1 hash as well.

(Unless we are ignoring comment lines and it is a comment line, or we are ignoring blank lines and it is a blank line, in which case it will be written to the file but not added to the hash.)

As is normal for file writes, the 'n' at the end of a line must be specified.

exception muddled.utils.**MuddleBug**(*message=None*, *retcode=1*)
Bases: *muddled.utils.GiveUp*

Use this to indicate that something has gone wrong with muddle itself.

We thus expect that a traceback will be produced.

class muddled.utils.**MuddleOrderedDict**
Bases: _abcoll.MutableMapping

A simple dictionary-like class that returns keys in order of (first) insertion.

class muddled.utils.**MuddleSortedDict**
Bases: _abcoll.MutableMapping

A simple dictionary-like class that returns keys in sorted order.

exception muddled.utils.**ShellError**(*cmd*, *retcode*, *output=None*)
Bases: *muddled.utils.GiveUp*

exception muddled.utils.**Unsupported**(*message=None*, *retcode=1*)
Bases: *muddled.utils.GiveUp*

Use this to indicate that an action is unsupported.

This is used, for instance, when git reports that it will not pull to a shallow clone, which is not an error, but the user will want to know.

This is deliberately a subclass of GiveUp, because it *is* telling muddle to give up an operation.

class muddled.utils.**VersionNumber**(*major=0*, *minor=0*)
Bases: object

Simple support for two part "semantic version" numbers.

Such version numbers are of the form <major>.<minor>

static from_string(*s*)

next()
Return the next (minor) version number.

static unset()
Return an unset version number.

Unset version numbers compare less than proper ones.

muddled.utils.**arch_name**()
Retrieve the name of the architecture on which we're running. Some builds require packages to be built on a particular (odd) architecture.

muddled.utils.**c_escape**(*v*)
Escape sensitive characters in v.

muddled.utils.**calc_file_hash**(*filename*)
Calculate and return the SHA1 hash for the named file.

muddled.utils.**copy_file**(*from_path*, *to_path*, *object_exactly=False*, *preserve=False*, *force=False*)
Copy a file (either a "proper" file, not a directory, or a symbolic link).

Just like recursively_copy, only not recursive :-)

If the target file already exists, it is overwritten.

Caveat: if the target file is a directory, it will not be overwritten. If the source file is a link, being copied as a link, and the target file is not a link, it will not be overwritten.

If 'object_exactly' is true, then if 'from_path' is a symbolic link, it will be copied as a link, otherwise the referenced file will be copied.

If 'preserve' is true, then the file's mode, ownership and timestamp will be copied, if possible. Note that on Un*x file ownership can only be copied if the process is running as 'root' (or within 'sudo').

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

muddled.utils.**copy_file_metadata**(*from_path*, *to_path*)
Copy file metadata.

If 'to_path' is a link, then it tries to copy whatever it can from 'from_path', treated as a link.

If 'to_path' is not a link, then it copies from 'from_path', or, if 'from_path' is a link, whatever 'from_path' references.

Metadata is: mode bits, atime, mtime, flags and (if the process has an effective UID of 0) the ownership (uid and gid).

muddled.utils.**copy_name_list_with_dirs**(*file_list*, *old_root*, *new_root*, *object_exactly=True*, *preserve=False*)
Given file_list, create file_list[new_root/old_root], creating any directories you need on the way.

file_list is a list of full path names. old_root is the old root directory new_root is where we want them copied

muddled.utils.**copy_without**(*src*, *dst*, *without=None*, *object_exactly=True*, *preserve=False*, *force=False*, *verbose=True*)

Copy files from the 'src' directory to the 'dst' directory, without those in 'without'

If given, 'without' should be a sequence of filenames - for instance, ['.bzr', '.svn'].

If 'object_exactly' is true, then symbolic links will be copied as links, otherwise the referenced file will be copied.

If 'preserve' is true, then the file's mode, ownership and timestamp will be copied, if possible. Note that on Un*x file ownership can only be copied if the process is running as 'root' (or within 'sudo').

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

If 'verbose' is true (the default), print out what we're copying.

Creates directories in the destination, if necessary.

Uses copy_file() to copy each file.

muddled.utils.**current_machine_name**()

Return the identity of the current machine - possibly including the domain name, possibly not

muddled.utils.**current_user**()

Return the identity of the current user, as an email address if possible, but otherwise as a UNIX uid

muddled.utils.**debian_version_is**(*test*, *ref*)

Return 1 if test > ref, -1 if ref > test, 0 if they are equal

muddled.utils.**do_shell_quote**(*str*)

muddled.utils.**domain_subpath**(*domain_name*)

Calculate the sub-path for a given domain name.

For instance:

```
>>> domain_subpath('a')
'domains/a'
>>> domain_subpath('a(b)')
'domains/a/domains/b'
>>> domain_subpath('a(b(c))')
'domains/a/domains/b/domains/c'
>>> domain_subpath('a(b(c)')
Traceback (most recent call last):
...
GiveUp: Domain name "a(b(c)" has mis-matched parentheses
```

muddled.utils.**dynamic_load**(*filename*)

muddled.utils.**ensure_dir**(*dir*, *verbose=True*)

Ensure that dir exists and is a directory, or throw an error.

muddled.utils.**find_by_predicate**(*source_dir*, *accept_fn*, *links_are_symbolic=True*)

Given a source directory and an acceptance function fn(source_base, file_name) -> result

Obtain a list of [result] if result is not None.

muddled.utils.**find_domain**(*root_dir*, *dir*)

Find the domain of 'dir'.

'root_dir' is the root of the (entire) muddle build tree.

This function basically works backwards through the path of 'dir', until it reaches 'root_dir'. As it goes, it assembles the full domain name for the domain enclosing 'dir'.

Returns the domain name, or None if 'dir' is not within a subdomain, and the directory of the root of the domain. That is:

(domain_name, domain_dir) or (None, None)

muddled.utils.**find_label_dir**(*builder*, *label*)
Given a label, find the corresponding directory.

- for checkout labels, the checkout directory
- for package labels, the install directory
- for deployment labels, the deployment directory

This is the heart of "muddle query dir".

muddled.utils.**find_local_relative_root**(*builder*, *label*)
Given a label, find its "local" root directory, relative to toplevel.

Calls find_local_root() and then calculates the location of that relative to the root of the entire muddle build tree.

muddled.utils.**find_local_root**(*builder*, *label*)
Given a label, find its "local" root directory.

For a normal label, this will be the normal muddle root directory (where the top-level .muddle/ directory is).

For a label in a subdomain, it will be the root directory of that subdomain - again, where its .muddle/ directory is.

muddled.utils.**find_root_and_domain**(*dir*)
Find the build tree root containing 'dir', and find the domain of 'dir'.

This function basically works backwards through the path of 'dir', until it finds a directory containing a '.muddle/' directory, that is not within a subdomain. As it goes, it assembles the full domain name for the domain enclosing 'dir'.

Returns a pair (root_dir, current_domain).

If 'dir' is not within a subdomain, then 'current_domain' will be None.

If 'dir' is not within a muddle build tree, then 'root_dir' will also be None.

muddled.utils.**get_cmd_data**(*thing*, *env=None*, *show_command=False*)
Run the command 'thing', and return its output.

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

Note that the output of the command is not shown whilst the command is running.

If the command returns a non-zero exit code, then we raise a ShellError.

(This is basically a muddle-flavoured wrapper around subprocess.check_output)

muddled.utils.**get_domain_name_from**(*dir*)
Given a directory 'dir', extract the domain name.

'dir' should not end with a trailing slash.

It is assumed that 'dir' is of the form "<something>/domains/<domain_name>", and we want to return <domain_name>.

muddled.utils.**get_os_version_name**()
Retrieve a string identifying this version of the operating system

Looks in /etc/os-release, which gives a different result than platform.py, which looks in /etc/lsb-release.

muddled.utils.**get_prefix_pair**(*prefix_one*, *value_one*, *prefix_two*, *value_two*)
Returns a pair (prefix_onevalue_one, prefix_twovalue_two) - used by rrw.py as a utility function

muddled.utils.**indent**(*text*, *indent*)
Return the text indented with the 'indent' string.

(i.e., place 'indent' in front of each line of text).

muddled.utils.**is_release_build**(*dir*)
Check if the given 'dir' is the top level of a release build.

'dir' should be the path to the directory contining the build's `.muddle` directory (the "top" of the build).

The build is assumed to be a release build if there is a file called `.muddle/Release`.

muddled.utils.**is_subdomain**(*dir*)
Check if the given 'dir' is a (sub)domain.

'dir' should be the path to the directory contining the build's `.muddle` directory (the "top" of the build).

The build is assumed to be a (sub)domain if there is a file called `.muddle/am_subdomain`.

muddled.utils.**iso_time**()
Retrieve the current time and date in ISO style `YYYY-MM-DD HH:MM:SS`.

muddled.utils.**join_domain**(*domain_parts*)
Re-join a domain name we split with split_domain.

muddled.utils.**mark_as_domain**(*dir*, *domain_name*)
Mark the build in 'dir' as a (sub)domain

This is done by creating a file `.muddle/am_subdomain`

'dir' should be the path to the directory contining the sub-build's `.muddle` directory (the "top" of the sub-build).

'dir' should thus be of the form "<somewhere>/domains/<domain_name>", but we do not check this.

The given 'domain_name' is written to the file, but this should not be particularly trusted - refer to the containing directory structure for the canonical domain name.

muddled.utils.**maybe_shell_quote**(*str*, *doQuote*)
If doQuote is False, do nothing, else shell-quote `str`.

Annoyingly, shell quoting things correctly must use backslashes, since quotes can (and will) be misinterpreted. Bah.

NB: Despite the name, this is actually "escaping", rather then "quoting". Specifically, any single quote, double quote or backslash characters in the original string will be converted to a backslash followed by the original character, in the final string.

muddled.utils.**normalise_dir**(*dir*)

muddled.utils.**normalise_path**(*dir*)

muddled.utils.**num_cols**()
How many columns on our terminal?

If it can't tell (e.g., because it curses is not available), returns 70.

muddled.utils.**pad_to**(*str*, *val*, *pad_with=' '*)
Pad the given string to the given number of characters with the given string.

muddled.utils.**page_text**(*progname*, *text*)
Try paging 'text' by piping it through 'progname'.

Looks for 'progname' on the PATH, and if os.environ['PATH'] doesn't exist, tries looking for it on os.defpath.

If an executable version of 'progname' can't be found, just prints the text out.

If 'progname' is None (or an empty string, or otherwise false), then just print 'text'.

muddled.utils.**parse_etc_os_release**()
Parse /etc/os-release and return a dictionary

This is *not* a good parser by any means - it is the quickest and simplest thing I could do.

Note that a line like:

```
FRED='Fred's name'
```

will give us:

```
key 'Fred' -> value r"Fred's name"
```

i.e., we do not treat backslashes in a string in any way at all. In fact, we don't do anything with strings other than throw away paired outer "" or ''.

Oh, also we don't check whether the names before the '=' signs are those that are expected, although we do provide a default value for 'ID' if it is not given (as the documentation for /etc/os-release' specified).

(Note that the standard library platform.py (in 2.7) looks at /etc/lsb-release, instead of /etc/os-release, which gives different results.)

muddled.utils.**parse_gid**(*builder*, *text_gid*)

Todo

One day, we should do something more intelligent than just assuming your gid is numeric

muddled.utils.**parse_mode**(*in_mode*)
Parse a UNIX mode specification into a pair (clear_bits, set_bits).

muddled.utils.**parse_uid**(*builder*, *text_uid*)

Todo

One day, we should do something more intelligent than just assuming your uid is numeric

muddled.utils.**print_string_set**(*ss*)
Given a string set, return a string representing it.

muddled.utils.**quote_list**(*lst*)
Given a list, quote each element of it and return them, space separated

muddled.utils.**recursively_copy**(*from_dir*, *to_dir*, *object_exactly=False*, *preserve=True*, *force=False*)
Take everything in from_dir and copy it to to_dir, overwriting anything that might already be there.

Dot files are included in the copying.

If object_exactly is true, then symbolic links will be copied as links, otherwise the referenced file will be copied.

If preserve is true, then the file's mode, ownership and timestamp will be copied, if possible. This is only really useful when copying as a privileged user.

If 'force' is true, then if a target file is not writeable, try removing it and then copying it.

muddled.utils.**recursively_remove**(*a_dir*)
Recursively demove a directory.

muddled.utils.**rel_join**(*vroot*, *path*)
Find what path would be called if it existed inside vroot. Differs from os.path.join() in that if path contains a leading '/', it is not assumed to override vroot.

If vroot is none, we just return path.

muddled.utils.**replace_root_name**(*base*, *replacement*, *filename*)
Given a filename, a base and a replacement, replace base with replacement at the start of filename.

muddled.utils.**run0**(*thing*, *env=None*, *show_command=True*, *show_output=True*)
Run the command 'thing', returning nothing.

(Run and return 0 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used.

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs. Note that this is the default.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

muddled.utils.**run1**(*thing*, *env=None*, *show_command=True*, *show_output=False*)
Run the command 'thing', returning its output.

(Run and return 1 value)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

Otherwise, the command output (stdout and stderr combined) is returned.

muddled.utils.**run2**(*thing*, *env=None*, *show_command=True*, *show_output=False*)
Run the command 'thing', returning the return code and output.

(Run and return 2 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

The output of the command (stdout and stderr) goes to the normal stdout whilst the command is running.

The command return code and output are returned as a tuple:

(retcode, output)

muddled.utils.**run3**(*thing*, *env=None*, *show_command=True*, *show_output=False*)

Run the command 'thing', returning the return code, stdout and stderr.

(Run and return 3 values)

'thing' may be a string (e.g., "ls -l") or a sequence (e.g., ["ls", "-l"]). Internally, a string will be converted into a sequence before it is used. Any non-string items in a 'thing' sequence will be converted to strings using 'str()' (e.g., if a Label instance is given).

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

If 'show_output' is true, then the output of the command (both stdout and stderr) will be printed out as the command runs.

The output of the command is shown whilst the command is running; its stdout goes to the normal stdout, and its stderr to stderr.

The command return code, stdout and stderr are returned as a tuple:

(retcode, stdout, stderr)

muddled.utils.**shell**(*thing*, *env=None*, *show_command=True*)

Run the command 'thing' in the shell.

If 'thing' is a string (e.g., "ls -l"), then it will be used as it is given.

If 'thing' is a sequence (e.g., ["ls", "-l"]), then each component will be escaped with pipes.quote(), and the result concatenated (with spaces between) to give the command line to run.

If 'env' is given, then it is the environment to use when running 'thing', otherwise 'os.environ' is used.

If 'show_command' is true, then "> <thing>" will be printed out before running the command.

The output of the command will always be printed out as it runs.

If the command returns a non-zero return code, then a ShellError will be raised, containing the returncode, the command string and any output that occurred.

Unlike the various 'runX' functions, this calls subprocess.Popen with 'shell=True'. This makes things like "cd" available in 'thing', and use of shell specific things like value expansion. It also, morei mportantly for muddle, allows commands like "git clone" to do their progress report "rolling" output. However, the warnings in the Python subprocess documentation should be heeded about not using unsafe command lines.

NB: If you *do* want to do "cd xxx; yyy", you're probably better doing:

```
    with Directory("xxx"):
        shell("yyy")
```

muddled.utils.**sort_domains**(*domains*)

Given a sequence of domain names, return them sorted by depth.

So, given some random domain names (and we forgot to forbid strange names starting with '+' or '-'):

```
>>> a = ['a', '+1', '-2', 'a(b(c2))', 'a(b(c1))', '+1(+2(+4(+4)))',
...     'b(b)', 'b', 'b(a)', 'a(a)', '+1(+2)', '+1(+2(+4))', '+1(+3)']
```

sorting "alphabetically" gives the wrong result:

```
>>> sorted(a)
['+1', '+1(+2(+4(+4)))', '+1(+2(+4))', '+1(+2)', '+1(+3)', '-2', 'a', 'a(a)', 'a(b(c1))', 'a(b(c
```

so we needed this function:

```
>>> sort_domains(a)
['+1', '+1(+2)', '+1(+2(+4))', '+1(+2(+4(+4)))', '+1(+3)', '-2', 'a', 'a(a)', 'a(b(c1))', 'a(b(c
```

If we're given a domain name that is None, we'll replace it with ''.

muddled.utils.**split_debian_version**(*v*)

Takes a debian-style version string - <major>.<minor>.<subminor>-<issue><additional> - and turns it into a dictionary with those keys.

muddled.utils.**split_domain**(*domain_name*)

Given a domain name, return a tuple of the hierarchy of sub-domains.

For instance:

```
>>> split_domain('a')
['a']
>>> split_domain('a(b)')
['a', 'b']
>>> split_domain('a(b(c))')
['a', 'b', 'c']
>>> split_domain('a(b(c)')
Traceback (most recent call last):
...
GiveUp: Domain name "a(b(c)" has mis-matched parentheses
```

We don't actually allow "sibling" sub-domains, so we try to complain helpfully:

```
>>> split_domain('a(b(c)(d))')
Traceback (most recent call last):
...
GiveUp: Domain name "a(b(c)(d))" has 'sibling' sub-domains
```

If we're given '' or None, we return [''], "normalising" the domain name.

```
>>> split_domain('')
['']
>>> split_domain(None)
['']
```

muddled.utils.**split_path_left**(*in_path*)

Given a path `a/b/c ...`, return a pair `(a, b/c..)` - ie. like `os.path.split()`, but leftward.

What we actually do here is to split the path until we have nothing left, then take the head and rest of the resulting list.

For instance:

```
>>> split_path_left('a/b/c')
('a', 'b/c')
>>> split_path_left('a/b')
('a', 'b')
```

For a single element, behave in sympathy (but, of course, reversed) to `os.path.split`:

```
>>> import os
>>> os.path.split('a')
('', 'a')
>>> split_path_left('a')
('a', '')
```

The empty string isn't really a sensible input, but we cope:

```
>>> split_path_left('')
('', '')
```

And we take some care with delimiters (hopefully the right sort of care):

```
>>> split_path_left('/a///b/c')
('', 'a/b/c')
>>> split_path_left('//a/b/c')
('', 'a/b/c')
>>> split_path_left('///a/b/c')
('', 'a/b/c')
```

`muddled.utils.`**`split_vcs_url`**(*url*)
Split a URL into a vcs and a repository URL. If there's no VCS specifier, return (None, None).

`muddled.utils.`**`string_cmp`**(*a*, *b*)
Return -1 if a < b, 0 if a == b, +1 if a > b.

`muddled.utils.`**`text_in_node`**(*in_xml_node*)
Return all the text in this node.

`muddled.utils.`**`total_ordering`**(*cls*)
Class decorator that fills-in missing ordering methods

`muddled.utils.`**`truncate`**(*text*, *columns=None*, *less=0*)
Truncate the given text to fit the terminal.

More specifically:

1. Split on newlines
2. If the first line is too long, cut it and add '...' to the end.
3. Return the first line

If 'columns' is 0, then don't do the truncation of the first line.

If 'columns' is None, then try to work out the current terminal width (using "curses"), and otherwise use 80.

If 'less' is specified, then the actual width used will be the calculated or given width, minus 'less' (so if columns=80 and less=2, then the maximum line length would be 78). Clearly this is ignored if 'columns' is 0.

`muddled.utils.`**`unescape_backslashes`**(*str*)
Replace every string 'X' with X, as if you were a shell

`muddled.utils.`**`unix_time`**()
Return the current UNIX time since the epoch.

muddled.utils.**unquote_list**(*lst*)
Given a list of objects, potentially enclosed in quotation marks or other shell weirdness, return a list of the actual objects.

muddled.utils.**well_formed_dot_muddle_dir**(*dir*)
Return True if this seems to be a well-formed .muddle directory

We're not trying to be absolutely rigorous, but do want to detect (for instance) an erroneous file with that name, or an empty directory

muddled.utils.**wrap**(*text*, *width=None*, ***kwargs*)
A convenience wrapper around textwrap.wrap()

(basically because muddled users will have imported utils already).

muddled.utils.**xml_elem_with_child**(*doc*, *elem_name*, *child_text*)
Return an element 'elem_name' containing the text child_text in doc.

17.2.19 muddled.version_control

Note: *The top-level VCS infrastructure, using VCS specific plugins to do the actual work.*

Routines which deal with version control.

class muddled.version_control.**VersionControlHandler**(*vcs*)
Bases: object

Handle all version control operations for a checkout.

The VersionControlSystem class knows how to do individual VCS operations, but is not required to know anything muddle-specific beyond how a Repository object works (or, at least, that is the aim).

This class acts as a translator between muddle actions on a checkout label and the underlying VCS actions.

Each underlying VCS (git, bzr, etc.) is used via an instance of this class.

- 'vcs' is the class corresponding to the particular version control system - e.g., Git.

branch_exists(*builder*, *co_label*, *branch*, *verbose=False*, *show_pushd=False*)
Returns True if a branch of that name exists.

This allowed to be conservative - e.g., in git the existence of a remote branch with the given name can be counted as True.

Will be called in the actual checkout's directory.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

branch_to_follow(*builder*, *co_label*)
Determine what branch is *actually* wanted.

Returns a branch name or None. None means that the description of this checkout in the build description is adequate - i.e., we do not need to override it with the branch of the build description.

BEWARE: this duplicates some of the code in sync().

checkout(*builder*, *co_label*, *verbose=True*)
Check this checkout out of version control.

The actual operation we perform is commonly called "clone" in actual version control systems. We retain the name "checkout" because it instantiates a muddle checkout.

commit(*builder*, *co_label*, *verbose=True*)
Commit any changes in the local working copy to the local repository.

In a centralised VCS, like subverson, this does not do anything, as there is no *local* repository.

create_branch(*builder*, *co_label*, *branch*, *verbose=False*, *show_pushd=False*)
Create a (new) branch of the given name.

Will be called in the actual checkout's directory.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

get_current_branch(*builder*, *co_label*, *verbose=False*, *show_pushd=False*)
Return the name of the current branch.

Will be called in the actual checkout's directory.

Return the name of the current branch (e.g., "master" or "Fred"), or None if there is no current branch.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

Raises a GiveUp exception if the VCS does not support this operation, or if something goes wrong.

get_file_content(*url*, *verbose=True*)
Retrieve a file's content via a VCS.

get_vcs_special_files()
Return the names of the 'special' files/directories used by this VCS.

For instance, if 'url' starts with "git+" then we might return [".git", ".gitignore", ".gitmodules"]

Returns an empty list if there is no such concept.

goto_branch(*builder*, *co_label*, *branch*, *verbose=False*, *show_pushd=False*)
Make the named branch the current branch.

Will be called in the actual checkout's directory.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

goto_revision(*builder*, *co_label*, *revision*, *branch=None*, *verbose=False*, *show_pushd=False*)
Go to the specified revision.

If branch is given, this may alter the behaviour.

Will be called in the actual checkout's directory.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

long_name

merge(*builder*, *co_label*, *verbose=True*)
Retrieve changes from the remote repository, and apply them to the local working copy, performing a merge operation if necessary.

Returns True if it changes its checkout (changes the files visible to the user), False otherwise.

must_pull_before_commit(*builder*, *co_label*)
Do we need to pull before we can commit?

This may depend on the options chosen for this checkout.

pull(*builder*, *co_label*, *upstream=None*, *repo=None*, *verbose=True*)
Retrieve changes from the remote repository, and apply them to the local working copy, but not if a merge operation would be required, in which case an exception shall be raised.

If 'upstream' and 'repo' are given, then they specify the upstream repository we should pull from, instead of using the 'normal' repository from the build description.

Returns True if it changes its checkout (changes the files visible to the user), False otherwise.

push (*builder*, *co_label*, *upstream=None*, *repo=None*, *verbose=True*)
Push changes in the local repository to the remote repository.

If 'upstream' and 'repo' are given, then they specify the upstream repository we should push to, instead of using the 'normal' repository from the build description.

Note that in a centralised VCS, like subversion, this is typically called "commit", since there is no local repository.

This operaton does not do a 'commit'.

reparent (*builder*, *co_label*, *force=False*, *verbose=True*)
Re-associate the local repository with its original remote repository,

(This is not relevant for all VCS systems, and will only be overridden for those where it does make sense - notably Bazaar)

This re-associates the local repository with the remote repository named in the muddle build description.

If 'force' is true, it does this regardless. If 'force' is false, then it only does it if the checkout is actually not so associated.

revision_to_checkout (*builder*, *co_label*, *force=False*, *before=None*, *verbose=False*, *show_pushd=True*)
Determine a revision id for this checkout, usable to check it out again.

The revision id we want is that we could use to check out an identical checkout.

If the local working set/repository/whatever appears to have been altered from the remove repository, or otherwise does not yield a satisfactory revision id (this is something only the subclass can tell), then the method should raise GiveUp, with as clear an explanation of the problem as possible.

If 'force' is true, then if the revision cannot be determined, return the orginal revision that was specified when the checkout was checked out.

> (Individual version control classes may opt to ignore the 'force' argument, either because it is not useful in their context, or because they can tell that the checkout is seriously astray/broken.)

If 'before' is given, it should be a string describing a date/time, and the revision id chosen will be the last revision at or before that date/time.

Note: This depends upon what the VCS concerned actually supports. This feature is experimental.

NB: if 'before' is specified, 'force' is ignored.

If 'show_pushd' is false, then we won't report as we "pushd" into the checkout directory.

NB: If the VCS class does not override this method, then the default implementation will raise a GiveUp unless 'force' is true, in which case it will return the string '0'.

short_name

status (*builder*, *co_label*, *verbose=False*, *quick=False*)
Report on the status of the checkout, in a VCS-appropriate manner

If there is nothing to be done for this repository, returns None.

Otherwise, returns a string comprising a report on the status of the repository, in a VCS appropriate manner.

If 'verbose', then report each checkout label as it is checked.

The reliability and accuracy of this varies by VCS, but the idea is that a checkout is 'safe' if:

- there are no files in the local checkout that are not also in the (local) repository, unless explicitly marked to be ignored
- there are no files that need committing to the local repository

In general, if a checkout is 'safe' then it should be OK to 'merge' the remote repository into it.

sync (*builder*, *co_label*, *verbose=False*, *sync=True*)

Attempt to go to the branch indicated by the build description.

- 'co_label' is the label for which to sync
- if 'verbose' is True, then report extra information about what is being done and why
- if 'sync' is False, don't actually do the sync - this is expected to be used with "verbose=True" to report on what we would do and why.

Do the first applicable of the following

- If this is the top-level build description, then:
  - if it has "builder.follow_build_desc_branch = True", then nothing needs to be done, as we're already there.
  - if it does not have "builder.follow_build_desc_branch = True", but a branch was specified for it (i.e., via "muddle init -branch"), then go to that branch.
  - if it does not have "builder.follow_build_desc_branch = True", and no branch was specified (at "muddle init"), then go to "master".
- If the build description specifies a revision for this checkout, go to that revision.
- If the build description specifies a branch for this checkout, and the checkout VCS supports going to a specific branch, go to that branch
- If the build description specifies that this checkout should not follow the build description (both Subversion and Bazaar support the "no_follow" option), then go to "master".
- If the build description specifies that this checkout is shallow, then give up.
- If the checkout's VCS does not support lightweight branching, then give up (the following choices require this).
- If the build description has "builder.follow_build_desc_branch = True", then go to the same branch as the build description.
- Otherwise, go to "master".

BEWARE: this duplicates some of the code in branch_to_follow().

class muddled.version_control.**VersionControlSystem**

Bases: object

Provide version control operations for a particular VCS.

This is a super-class, acting as a template for the actual classes.

The intent is that implementors of this interface do not need to know much about muddle, although they will have to have some understanding of the contents of a Repository object.

add_files (*files=None*, *verbose=True*)

If files are given, add them, but do not commit.

Will be called in the actual checkout's directory.

allowed_options = set(['no_follow', 'shallow_checkout'])

allows_relative_in_repo()
Does this VCS allow relative locations within the repository to be checked out?

Subversion does. Distributed revision control systems tend not to.

branch_exists(*branch*)
Returns True if a branch of that name exists.

This allowed to be conservative - e.g., in git the existence of a remote branch with the given name can be counted as True.

Will be called in the actual checkout's directory.

Raises Unsupported if the VCS does not support this operation.

checkout(*repo*, *co_leaf*, *options*, *verbose=True*)
Checkout (clone) a given checkout.

Will be called in the parent directory of the checkout.

Expected to create a directory called <co_leaf> therein.

Any exception raised will be "wrapped" by the calling handler.

commit(*repo*, *options*, *verbose=True*)
Will be called in the actual checkout's directory.

Any exception raised will be "wrapped" by the calling handler.

create_branch(*branch*, *verbose=False*)
Create a (new) branch of the given name.

Will be called in the actual checkout's directory.

Raises Unsupported if the VCS does not support this operation.

Raises GiveUp if a branch of that name already exists.

get_current_branch()
Return the name of the current branch.

Will be called in the actual checkout's directory.

Returns the name of the branch, or None if there is no current branch. Note that the master branch is thus returned as "master", *not* as None.

Raises Unsupported if the VCS does not support this operation.

get_file_content(*url*, *verbose=True*)
Retrieve a file's content via a VCS.

get_vcs_special_files()
Return the names of the 'special' files/directories used by this VCS.

For instance, if 'url' starts with "git+" then we might return [".git", ".gitignore", ".gitmodules"]

Returns an empty list if there is no such concept.

goto_branch(*branch*, *verbose=False*)
Make the named branch the current branch.

Will be called in the actual checkout's directory.

Raises Unsupported if the VCS does not support this operation.

Raises GiveUp if there is no existing branch of that name.

goto_revision(*revision*, *branch=None*, *repo=None*, *verbose=False*)
Make the specified revision current.

Note that this may leave the working data (the actual checkout directory) in an odd state, in which it is not sensible to commit, depending on the VCS and the revision.

Will be called in the actual checkout's directory.

If the VCS supports it, may also take a branch name.

Raises Unsupported if the VCS does not support this operation.

Raises GiveUp if there is no such revision.

init_directory(*files=None*, *verbose=True*)
If the directory does not appear to have had '<vcs> init' run in it, then do so first.

Will be called in the actual checkout's directory.

merge(*other_repo*, *options*, *verbose=True*)
Will be called in the actual checkout's directory.

Any exception raised will be "wrapped" by the calling handler.

Returns True if it changes its checkout (changes the files visible to the user), False otherwise.

must_pull_before_commit(*options*)
Do we need to 'pull' before we 'commit'?

In a centralised VCS like subverson, this is highly recommended.

In a distributed VCS like bazaar or git, it is unnecessary.

We shall default to the distributed answer, and individual VCS support can override if necessary.

pull(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

Any exception raised will be "wrapped" by the calling handler.

Returns True if it changes its checkout (changes the files visible to the user), False otherwise.

push(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

Any exception raised will be "wrapped" by the calling handler.

reparent(*co_leaf*, *remote_repo*, *options*, *force=False*, *verbose=True*)
Will be called in the actual checkout's directory.

revision_to_checkout(*repo*, *co_leaf*, *options*, *force=False*, *before=None*, *verbose=True*)
Will be called in the actual checkout's directory.

status(*repo*, *options*, *quick=False*)
Will be called in the actual checkout's directory.

Return status text or None if there is no interesting status.

supports_branching()
Does this VCS support "lightweight" branching like git?

Git clearly does. Subversion does not (branches are a different path inside the repository - if we ever wanted/needed to, we might accomodate that, but at the moment we don't), and bazaar does not (branches *are* clones/checkouts), although I believe there are proposals to handle lightweight branching as well (but that is "as well", so particular user might or might not want to use that facility).

Thus the default is False.

muddled.version_control.**checkout_from_repo**(*builder*, *co_label*, *repo*, *co_dir=None*, *co_leaf=None*)

Declare that the checkout for 'co_label' comes from Repository 'repo'

We will take the repository described in 'repo' and check it out into:

- src/<co_label>.name or
- src/<co_dir>/<co_label>.name or
- src/<co_dir>/<co_leaf> or
- src/<co_leaf>

depending on whether <co_dir> and/or <co_leaf> are given. We will assign the label <co_label> to this directory/repository combination.

muddled.version_control.**get_vcs_docs**(*vcs*)

Given a VCS short name, return the docs for how muddle handles it

muddled.version_control.**get_vcs_instance**(*vcs*)

Given a VCS short name, return a VCS instance.

muddled.version_control.**get_vcs_instance_from_string**(*repo_str*)

Given a <vcs>+<url> string, return a VCS instance and <url>.

muddled.version_control.**list_registered**(*indent=''*)

Return a list of registered version control systems.

muddled.version_control.**register_vcs**(*scheme*, *vcs_instance*, *docs=None*, *options=None*)

Register a VCS instance with a VCS scheme prefix.

Also, preferably, register the VCS documentation on how muddle handles it, and maybe also any allowed options.

muddled.version_control.**vcs_get_directory**(*url*, *directory=None*)

Retrieve (clone) the directory identified by the URL, via its VCS.

If 'directory' is given, then clones to the named directory.

Looks at the first few characters of the URL to determine the VCS to use - so, e.g., "bzr" for "bzr+ssh://whatever".

Raises KeyError if the scheme is not one for which we have a registered handler.

muddled.version_control.**vcs_get_file_data**(*url*)

Return the content of the file identified by the URL, via its VCS.

Looks at the first few characters of the URL to determine the VCS to use - so, e.g., "bzr" for "bzr+ssh://whatever".

Returns a string (the content of the file).

Raises KeyError if the scheme is not one we have a registered file getter for.

muddled.version_control.**vcs_handler_for**(*builder*, *co_label*)

Create a VCS handler for the given checkout label.

We look up the repository for this label, and which VCS to use is determined by interpreting the initial part of the repository URI's protocol.

We then create a handler that will call the appropriate VCS-specific mechanisms for any VCS operations on this checkout.

- co_label - The label for this checkout. This includes the name and domain (if any) for the checkout

muddled.version_control.**vcs_init_directory**(*scheme*, *files=None*)
Initialised the current directory for this VCS, and add the given list of files.

'scheme' is "git", "bzr", etc. - as taken from the first few characters of the muddle repository URL - so, e.g., "bzr" for "bzr+ssh://whatever".

Raises KeyError if the scheme is not one for which we have a registered handler.

muddled.version_control.**vcs_pull_directory**(*url*)
Pull the current directory from the repository indicated by the URL

Looks at the first few characters of the URL to determine the VCS to use - so, e.g., "bzr" for "bzr+ssh://whatever".

Raises KeyError if the scheme is not one for which we have a registered handler.

muddled.version_control.**vcs_push_directory**(*url*)
Push the current directory to the repository indicated by the URL

Looks at the first few characters of the URL to determine the VCS to use - so, e.g., "bzr" for "bzr+ssh://whatever".

Raises KeyError if the scheme is not one for which we have a registered handler.

muddled.version_control.**vcs_special_files**(*url*)
Return the names of the 'special' files/directories used by this VCS.

For instance, if 'url' starts with "git+" then we might return [".git", ".gitignore", ".gitmodules"]

17.2.20 muddled.version_stamp

Note: *Stamp files - a record of the content of a build tree.*

VersionStamp and stamp file support.

Stamp files

Stamp files are INI files, implemented using the Python ConfigParser module (specifically, RawConfigParser).

INI files are composed of one or more sections, each of the form:

```
[header]
key = value
key = value
```

Indentation is allowed, but will be ignored.

Comment lines may be introduced with either '#' or ';', and in-line comments may be added by following '[header]' or 'key = value' with whitespace and a ';' (but not a '#').

Muddle treats '#' comment lines specially in stamp files, but is not aware of ';' comments, and we recommend not using them in muddle stamp files.

When a stamp file is written or read, a SHA1 hash is calculated from the lines of text being written/read. Since mid-July 2012, comment lines starting with '#', and blank lines (whitespace only) are not included in that SHA1 hash. Since a stamp file uniquely describes the current state of a muddle build tree, the hash calculated from its content can be regarded as a version id for the build tree.

It can sometimes be useful to edit a stamp file. If you do so, it is advisable to add a comment or comments indicating what has been done and why.

All muddle commands for handling stamp files are subcommands of either "muddle stamp" or "muddle unstamp".

Version 1 Stamp Files

We retain limited support for version 1 stamp files, mainly to allow reading existing (legacy) files. There *is* provision for writing version 1 stamp files, but it is not guaranteed to be accurate.

Version 1 stamp files were replaced because they could not unambiguously store all of the information needed by the Repository class we now use to remember where a checkout "came from".

Support for version 1 stamp files will be removed at some future time.

Version 2 Stamp Files

Version 2 stamp files are the current form of stamp file.

This section describes the current content of a stamp file, as currently written by muddle. For the rest of this section, "stamp files" should be taken to mean "version 2 stamp files".

All stamp files start with some standard comment lines:

```
# Muddle stamp file
# Written at 2012-07-12 13:13:13
#            2012-07-12 12:13:13 UTC
```

The date and time stamps are in local time and UTC respectively.

This is followed by a general section:

```
[STAMP]
version = 2
```

which at the moment just identifies the version of the stamp file.

Next comes a section identifying the build description - this repeats information taken from the RootRepository, Description and VersionsRepository files in the `.muddle/` directory of the build. For instance:

```
[ROOT]
repository = git+ssh://git@palmera.c//opt/kynesim/projects/001
description = builds/01.py
versions_repo = git+file:///home/tibs/temp/r/versions
```

and, if "muddle init -branch" was originally used, also the DescriptionBranch file:

```
[ROOT]
repository = git+ssh://git@palmera.c//opt/kynesim/projects/001
description = builds/01.py
description_branch = test-v0.1
versions_repo = git+file:///home/tibs/temp/r/versions
```

The keys ('repository', etc.) are always presented in the same order - this is a general principle within stamp file sections, so that stamp files themselves can be comparable with simple tools such as `diff`.

Next, if the build tree has subdomains, will come sections describing those domains. For instance:

```
[DOMAIN subdomain1]
description = builds/01.py
name = subdomain1
repository = git+file:///home/tibs/sw/muddle/tests/transient/repo/subdomain1

[DOMAIN subdomain1(subdomain3)]
description = builds/01.py
name = subdomain1(subdomain3)
repository = git+file:///home/tibs/sw/muddle/tests/transient/repo/subdomain3
```

Domain sections occur in "C" sort order of the domain names. Note that there will not be any domain sections if there are no subdomains (i.e., if there is no `domains/` directory in the build tree).

Note: The [DOMAIN] sections do not record a 'description_branch' for each subdomain. We do not currently support specifying a particular branch for the subdomain build description in the same manner as is done at the top level with "muddle init -branch".

Next come sections for each checkout. For instance:

```
[CHECKOUT (subdomain1)builds]
co_label = checkout:(subdomain1)builds/checked_out
co_leaf = builds
repo_vcs = git
repo_from_url_string = None
repo_base_url = file:///home/tibs/sw/muddle/tests/transient/repo/subdomain1
repo_name = builds
repo_prefix_as_is = False
repo_revision = 7d8377a18efcd8b3d7b788de8cee26aa6d770005
```

and:

```
[CHECKOUT builds]
co_label = checkout:builds/checked_out
co_leaf = builds
repo_vcs = git
repo_from_url_string = None
repo_base_url = ssh://git@palmera.c//opt/kynesim/projects/001
repo_name = builds
repo_prefix_as_is = False
repo_revision = dfb06f29c81828bbdfc48ce53d7bc4203b68a459

[CHECKOUT busybox-1.17.1]
co_label = checkout:busybox-1.17.1/checked_out
co_dir = linux
co_leaf = busybox-1.17.1
repo_vcs = git
repo_from_url_string = None
repo_base_url = ssh://git@palmera.c//opt/kynesim/projects/001
repo_name = busybox-1.17.1
repo_prefix = linux
repo_prefix_as_is = False
repo_revision = 965b4809b5ca93df0a4973e043a5a9af0ecf50e3
repo_branch = issue99-fix
```

The values presented give the checkout label, its location in the build tree ('co_dir' and 'co_leaf'), and its Repository instance (the 'repo_xxx' values. See "muddle doc version_control.checkout_from_repo" for some information on how the 'co_xxx' values are used, and "muddle doc repository.Repository" for more information on the Repository class).

Note: The 'repo_revision' is the current revision of the checkout, and 'repo_branch' is its current branch if that is not "master", and if the VCS for the checkout supports setting branches.

Again, these are presented in "C" sort order of the checkout name, including the domain component - this means that subdomain checkouts will occur before toplevel checkouts. Also, while not all checkout sections will contain the same values, the order in which they are presented will always be the same.

Finally, if "muddle stamp" reported any problems in creating the stamp file, these will be saved in a problems section. For instance:

```
[PROBLEMS]
problem1 = builds: 'svnversion' reports checkout has revision '459M'
```

(a subversion revision ending in 'M' means that the checkout was modified and not yet committed). Problems are presented as 'problem1', 'problem2', etc. In general, the checkout name should be the first part of the message occurring as the problem value.

Release stamp files

Release stamp files are an extension of normal stamp files that also specify a release. As such, they have an extra section (after the `[STAMP]` section) of the form:

```
[RELEASE]
name = project99
version = 1.2.3
archive = tar
compression = gzip
```

This indicates that the rest of the stamp file describes a release called (or for) "project99", and that it is version 1.2.3. The release will be archived using tar, and the tar file will be compressed using gzip.

Both the `name` and `version` specified must start with an ASCII alphanumeric, and may only contain ASCII alphanumerics, and the characters '.', '-' or '_'. For instance:

```
version = v1-2.11
version = xvii
version = 0.9.3alpha1
```

The `archive` value must currently be `tar` - other values may perhaps be allowed in the future.

The `compression` value must be either `gzip` or `bzip2`.

Note that any release stamp file can be read as a "normal" stamp file - the extra release-specific information will just be ignored.

class `muddled.version_stamp.`**`CheckoutTupleV1`**(*name*, *repo*, *rev*, *rel*, *dir*, *domain*, *co_leaf*, *branch*)

Bases: `tuple`

`branch`
Alias for field number 7

`co_leaf`
Alias for field number 6

`dir`
Alias for field number 4

domain
Alias for field number 5

name
Alias for field number 0

rel
Alias for field number 3

repo
Alias for field number 1

rev
Alias for field number 2

class muddled.version_stamp.**ReleaseSpec**(*name=None*, *version=None*, *archive=None*, *compression=None*, *hash=None*)

Bases: object

The basic specification of a Release.

The following correspond to the [RELEASE] section in a release stamp file:

- 'name'
- 'version'
- 'archive'
- 'compression'

'name' and 'version' may be set to None, or to a valid name/version.

When a 'name' or 'version' of None is written out to a release file, it is written out as '<REPLACE THIS>', a carefully invalid value, indicating that the user needs to edit the file.

Otherwise, 'name' and 'version' values must start with ASCII alphanumeric and continue with ASCII alphanumeric, hyphen, underscore or dot.

'archive' and 'compression' may be set to None or a value from the '**allowed_**' lists. If they are set to None, they in fact get set to the first '**allowed_**' value, so this is a simple way of selecting the default.

We also allow the SHA1 hash of a stamp file to be remembered:

- 'hash'

There is no special handling for this, and it defaults to None.

allowed_archive_values = ['tar']

allowed_compression_values = ['gzip', 'bzip2']

archive

compression

static from_file(*filename*)
Read a simple representation of ourself from a file.

name

name_re = <_sre.SRE_Pattern object>

version

version_re = <_sre.SRE_Pattern object>

write_to_file(*filename*)
Write a simple representation of ourself out to a file.

class muddled.version_stamp.**ReleaseStamp**
Bases: *muddled.version_stamp.VersionStamp*

A VersionStamp with extra stuff to describe a release.

static from_builder(*builder*, *quiet=False*)
Construct a ReleaseStamp from a muddle build description.

'builder' is the muddle Builder for our build description.

If 'quiet' is True, then we will not print information about what we are doing, and we will not print out problems as they are found.

Returns a tuple of:

- the new ReleaseStamp instance
- a (possibly empty) list of problem summaries. If this is empty, then the stamp was calculated fully. Note that this is the same list as held within the ReleaseStamp instance itself.

static from_file(*filename*)
Construct a ReleaseStamp by reading in a stamp file.

Returns a new ReleaseStamp instance.

Note that the SHA1 computed and reported for that ReleaseStamp does not include blank lines or comment lines that start with a '#' (or whitespace and a '#').

write_to_file_object(*fd*, *version=2*)
Write our data out to a file-like object (one with a 'write' method).

class muddled.version_stamp.**VersionStamp**
Bases: object

A representation of the revision state of a build tree's checkouts.

Our internal data is:

- 'repository' is a string giving the default repository (as stored in `.muddle/RootRepository`)
- `description` is a string naming the build description (as stored in `.muddle/Description`)
- `description_branch` is None or a string naming the branch of the build description (as stored in `.muddle/DescriptionBranch`, and originally specified by "muddle init -branch")
- 'versions_repo' is a string giving the versions repository (as stored in `.muddle/VersionsRepository`)
- 'domains' is a dictionary mapping domain names to tuples of the form (domain_repo, domain_desc), where:
  - domain_repo is the default repository for the domain
  - domain_desc is the build description for the domain
- 'checkouts' is a dictionary mapping checkout labels to tuples of the form (co_dir, co_leaf, repo), where:
    - co_dir is the sub-path between src/ and the co_leaf
    - co_leaf is the name of the directory within src/ that actually contains the checkout
    - repo is a Repository instance, where to find the checkout remotely

These are the appropriate arguments for the checkout_from_repo() function in version_control.py.

The checkout will be in src/<co_dir>/<co_leaf> (if <co_dir> is set), or src/<co_leaf> (if it is not).

- 'options' is a dictionary mapping checkout labels to dictionaries of the form {option_name : option_value}. There will only be entries for those checkouts which do have options.
- 'problems' is a list of problems in determining the stamp information. This will be of zero length if the stamp if accurate, but will otherwise contain a string for each checkout whose revision could not be accurately determined.

Note that when problems descriptions are written to a stamp file, they are truncated.

MAX_PROBLEM_LEN = 100

compare_checkouts(*other*, *fd=<open file '<stdout>', mode 'w'>*)

Compare the checkouts in this VersionStamp with those in another.

'other' is another VersionStamp.

'fd' is where any messages should be written, defaulting to stdout. If this is None, then no messages will be writen.

Note that this only compares the checkouts (including their options) - it does not compare any of the other fields in a VersionStamp.

Returns a tuple of (deleted, new, changed, problems) sequences, where these are:

- a sequence of tuples of the form:

  (co_label, co_dir, co_leaf, repo)

  for checkouts that are in this VersionStamp but not in the 'other' - i.e., "deleted" checkouts

- a sequence of tuples of the form:

  (co_label, co_dir, co_leaf, repo)

  for checkouts that are in the 'other' VersionStamp but not in this - i.e., "new" checkouts

- a sequence of tuples for any checkouts with differing revisions, of the form:

  (co_label, revision1, revision2)

  where 'this_repo' and 'other_repo' are relevant Repository instances.

- a sequence of tuples of the form:

  (co_label, problem_string)

  for checkouts that are present in both VersionStamps, but differ in something other than revision.

static **from_builder**(*builder*, *force=False*, *just_use_head=False*, *before=None*, *quiet=False*)

Construct a VersionStamp from a muddle build description.

'builder' is the muddle Builder for our build description.

If 'force' is true, then attempt to "force" a revision id, even if it is not necessarily correct. For instance, if a local working directory contains uncommitted changes, then ignore this and use the revision id of the committed data. If it is actually impossible to determine a sensible revision id, then use the revision specified by the build description (which defaults to HEAD). For really serious problems, this may refuse to guess a revision id.

(Typical use of this is expected to be when a trying to calculate a stamp reports problems in particular checkouts, but inspection shows that these are artefacts that may be ignored, such as an executable built in the source directory.)

If 'just_use_head' is true, then HEAD will be used for all checkouts. In this case, the repository specified in the build description is used, and the revision id and status of each checkout is not checked.

If 'before' is given, it should be a string describing a date/time, and the revision id chosen for each checkout will be the last revision at or before that date/time.

Note: This depends upon what the VCS concerned actually supports. This feature is experimental.

If 'quiet' is True, then we will not print information about what we are doing, and we will not print out problems as they are found.

Returns a tuple of:

- the new VersionStamp instance
- a (possibly empty) list of problem summaries. If this is empty, then the stamp was calculated fully. Note that this is the same list as held within the VersionStamp instance itself.

static `from_file` (*filename*)
Construct a VersionStamp by reading in a stamp file.

Returns a new VersionStamp instance.

Note that the SHA1 computed and reported for that VersionStamp does not include blank lines or comment lines that start with a '#' (or whitespace and a '#').

`print_problems` (*output=None*, *truncate=None*, *indent=''*)
Print out any problems.

If 'output' is not specified, then it will be STDOUT, otherwise it should be a file-like object (supporting 'write').

If 'truncate' is None (or zero, non-true, etc.) then the problems will be truncated to the same length as when writing them to a stamp file.

'indent' should be a string to print in front of every line.

If there are no problems, this method does not print anything out.

`write_to_file` (*filename*, *version=2*)
Write our data out to a file.

By default, writes out a version 2 stamp file, as opposed to the older version 1 format.

Returns the SHA1 hash for the file.

`write_to_file_object` (*fd*, *version=2*)
Write our data out to a file-like object (one with a 'write' method).

By default, writes out a version 2 stamp file, as opposed to the older version 1 format.

Returns the SHA1 hash for the file.

Note that the SHA1 does not include blank lines or comment lines that start with a '#' (or whitespace and a '#').

`muddled.version_stamp.`**`get_and_remove_option`** (*config*, *section*, *name*)
Get an option from a section, as a string, and also remove it.

`muddled.version_stamp.`**`make_RawConfigParser`** (*ordered=False*, *sorted=False*)
Make a RawConfigParrser.

Always tell it we want to preserve the case of keys.

If 'ordered', then use a MuddleOrderedDict inside it, so that things remember their insertion order, and that is used on output.

If 'sorted', use a MuddleSortedDict, so that things are sorted, and thus output in sorted order.

If neither, don't specify a dict, and random stuff might happen

muddled.version_stamp.**maybe_get_option**(*config*, *section*, *name*, *remove=False*)
Get an option from a section, as a string, or as None.

If the option is present, return it, otherwise return None.

If remove is true, and the option was present, also remove it.

muddled.version_stamp.**maybe_set_option**(*config*, *section*, *name*, *value*)
Set an option in a section, if its value is not None.

17.2.21 muddled.xmlconfig

A utility module which allows you to read XML files and then query them as XPath-like paths.

Call readXml() to read an XML file and return an xmlConfig object which you can then pass queries like:

/elem1/elem2/elem3

This is essentially a (restricted) XPath query with an implicit ::text() appended.

This file is hereby placed in the public domain - Richard Watts, <rrw@kynesim.co.uk> 2009-10-23.

(as this is about the third time I have had to write it and I am getting quite bored ..)

class muddled.xmlconfig.**Config**(*in_file*)
Bases: object

Represents a configuration file

Parse an XML config file into a local representation

exists(*key*)
Given a key, decide if its value exists in the configuration file.

query(*keys*)
Perform a query on this list of keys and return the node which matches (and which you can then call text() on)

query_bool(*key*)
Given an XPath-like expression, return a boolean value based on a text value of 'true' (True) or anything else (False)

If the node doesn't exist, throw.

query_hashlist(*key*, *subkeys*)
Given an XPath-like expression and a list of subkeys, take the list denoted by key and return a list of hashes pointing the subkeys at their values.

query_int(*key*)
Given an XPath-like expression a/b/c.., return the text in the final node interpreted as an integer.

If the node doesn't exist, throw.

query_list(*key*)
Given an XPath-like expression, return a list containing the text from all values key0..keyN that actually exist

query_string(*key*)
Given an XPath-like expression /a/b/c , return the text in the final node. If the node doesn't exist, throw.

split_key(*instring*)
Take a series of components and split them by '/'.

text(*node*)
Collect all the text in an XML node

exception muddled.xmlconfig.**ConfigError**
Bases: exceptions.Exception

17.3 Lower-level modules

17.3.1 muddled.checkouts

muddled.checkouts.simple

Simple entry points so that descriptions can assert the existence of checkouts easily

muddled.checkouts.simple.**absolute**(*builder*, *co_name*, *repo_url*, *rev=None*, *branch=None*)
Check out a repository from an absolute URL.

<repo_url> must be of the form <vcs>+<url>, where <vcs> is one of the support version control systems (e.g., 'git', 'svn').

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

The repository <repo_url>/<co_name> will be checked out into src/<co_name>.

muddled.checkouts.simple.**relative**(*builder*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)
A simple, VCS-controlled, checkout.

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out into src/<co_name>, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

For example:

```
<base_url>/<co_name>  -->  src/<co_name>
```

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> will be checked out instead:

```
<base_url>/<repo_relative>  -->  src/<co_name>
```

muddled.checkouts.twolevel

Two-level checkouts. Makes it slightly easier to separate checkouts out into roles. I've deliberately not implemented arbitrary-level checkouts for fear of complicating the checkout tree.

muddled.checkouts.twolevel.**absolute**(*builder*, *co_dir*, *co_name*, *repo_url*, *rev=None*, *branch=None*)

Check out a twolevel repository from an absolute URL.

<repo_url> must be of the form <vcs>+<url>, where <vcs> is one of the support version control systems (e.g., 'git', 'svn').

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

The repository <repo_url>/<co_name> will be checked out into src/<co_dir>/<co_name>.

muddled.checkouts.twolevel.**relative**(*builder*, *co_dir*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A two-level version of checkout.simple.relative().

It attempts to check out <co_dir>/<co_name> (but see below).

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> ...

In the normal case, the location in the repository and in the checkout is assumed the same (i.e., <co_dir>/<co_name>). So, for instance, with co_dir="A" and co_name="B", the repository would have:

```
<base_url>/A/B
```

which we would check out into:

```
src/A/B
```

Occasionally, though, the repository is organised differently, so for instance, one might want to checkout:

```
<base_url>/B
```

into:

```
src/A/B
```

In this latter case, one can use the 'repo_relative' argument, to say where the checkout is relative to the repository's "base". So, in the example above, we still have co_dir="A" and co_name="B", but we also want to say repo_relative=B.

muddled.checkouts.twolevel.**twolevel**(*builder*, *co_dir*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A two-level version of checkout.simple.relative().

It attempts to check out <co_dir>/<co_name> (but see below).

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

If <repo_relative> is None then the repository <base_url>/<co_name> will be checked out, where <base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

If <repo_relative> is not None, then the repository <base_url>/<repo_relative> ...

In the normal case, the location in the repository and in the checkout is assumed the same (i.e., <co_dir>/<co_name>). So, for instance, with co_dir="A" and co_name="B", the repository would have:

```
<base_url>/A/B
```

which we would check out into:

```
src/A/B
```

Occasionally, though, the repository is organised differently, so for instance, one might want to checkout:

```
<base_url>/B
```

into:

```
src/A/B
```

In this latter case, one can use the 'repo_relative' argument, to say where the checkout is relative to the repository's "base". So, in the example above, we still have co_dir="A" and co_name="B", but we also want to say repo_relative=B.

muddled.checkouts.multilevel

Multi-level checkouts. Required for embedding things like android, which have a lot of deep internal checkouts.

muddled.checkouts.multilevel.**absolute**(*builder*, *co_dir*, *co_name*, *repo_url*, *rev=None*, *branch=None*)

Check out a multilevel repository from an absolute URL.

<repo_url> must be of the form <vcs>+<url>, where <vcs> is one of the support version control systems (e.g., 'git', 'svn').

<rev> may be a revision (specified as a string). "HEAD" (or its equivalent) is assumed by default.

<branch> may be a branch. "master" (or its equivalent) is assumed by default.

The repository <repo_url> will be checked out into src/<co_dir>. The checkout will be identified by the label checkout:<co_name>/checked_out.

muddled.checkouts.multilevel.**relative**(*builder*, *co_dir*, *co_name*, *repo_relative=None*, *rev=None*, *branch=None*)

A multilevel checkout, with checkout name unrelated to checkout directory.

Sometimes it is necessary to cope with checkouts that either:

1. are more than two directories below src/, or
2. have a checkout name that is not the same as the "leaf" directory in their path

Both of these can happen when trying to represent an Android build, for instance.

Thus:

```
multilevel.relative(builder, co_dir='this/is/here', co_name='checkout1')
```

will look for the repository <base_url>/this/is/here and check it out into src/this/is/here, but give it label checkout:checkout1/checked_out.

(<base_url> is the base URL as specified in .muddle/RootRepository (i.e., the base URL of the build description, as used in "muddle init").

For the moment, <repo_relative> is ignored.

17.3.2 muddled.deployments

muddled.deployments.collect

Collect deployment.

This deployment is used to collect elements from:

- checkout directories
- package 'obj' directories
- package role 'install' directories
- other deployments

into deployment directories, usually to be processed by some external tool.

class muddled.deployments.collect.**AssemblyDescriptor**(*from_label*, *from_rel*, *to_name*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *usingRSync=False*, *obeyInstructions=True*)

Bases: object

Construct an assembly descriptor.

We copy from the directory from_rel in from_label (package, deployment, checkout) to the name to_name under the deployment.

Give a package of '*' to copy from the install directory for a given role.

If recursive is True, we'll copy recursively.

•failOnAbsentSource - If True, we'll fail if the source doesn't exist.

•copyExactly - If True, keeps links. If false, copies the file they point to.

get_source_dir(*builder*)

class muddled.deployments.collect.**CollectApplyChmod**

Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.collect.**CollectApplyChown**

Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.collect.**CollectDeploymentBuilder**

Bases: *muddled.depend.Action*

Builds the specified collect deployment.

add_assembly(*assembly_descriptor*)

apply_instructions(*builder*, *label*, *prepare*, *deploy_path*)

build_label(*builder*, *label*)
Actually do the copies ..

deploy(*builder*, *label*, *target_base*)

sort_out_and_run_instructions(*builder*, *label*)

class muddled.deployments.collect.**InstructionImplementor**
Bases: object

apply(*builder*, *instruction*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instruction*, *role*, *path*)
Prepares for rsync. This means fixing up the destination file (e.g. removing it if it may have changed uid by a previous deploy) so we will be able to rsync it.

muddled.deployments.collect.**copy_from_checkout**(*builder*, *name*, *checkout*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

muddled.deployments.collect.**copy_from_deployment**(*builder*, *name*, *dep_name*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

usingRSync - set to True to copy with rsync - substantially faster than cp

muddled.deployments.collect.**copy_from_package_obj**(*builder*, *name*, *pkg_name*, *pkg_role*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*)

- **If 'usingRSync' is true, copy with rsync - substantially faster than** cp, if you have rsync. Not very functional if you don't :-)

muddled.deployments.collect.**copy_from_role_install**(*builder*, *name*, *role*, *rel*, *dest*, *recursive=True*, *failOnAbsentSource=False*, *copyExactly=True*, *domain=None*, *usingRSync=False*, *obeyInstructions=True*)

Add a requirement to copy from the given role's install to the named deployment.

'name' is the name of the collecting deployment, as created by:

```
deploy(builder, name)
```

which is remembered as a rule whose target is `deployment:<name>/deployed`, where <name> is the 'name' given.

'role' is the role to copy from. Copying will be based from 'rel' within the role's `install`, to 'dest' within the deployment.

The label `package:(<domain>)*{<role>}/postinstalled` will be added as a dependency of the collecting deployment rule.

An AssemblyDescriptor will be created to copy from 'rel' in the install directory of the label `package:*{<role>}/postinstalled`, to 'dest' within the deployment directory of 'name', and added

to the rule's actions.

So, for instance:

```
copy_from_role_install(builder,'fred','data','public','data/public',
                       True, False, True)
```

might copy (recursively) from:

```
install/data/public
```

to:

```
deploy/fred/data/public
```

'rel' may be the empty string ('') to copy all files in the install directory.

- If 'recursive' is true, then copying is recursive, otherwise it is not.
- If 'failOnAbsentSource' is true, then copying will fail if the source does not exist.
- If 'copyExactly' is true, then symbolic links will be copied as such, otherwise the linked file will be copied.
- **If 'usingRSync' is true, copy with rsync - substantially faster than** cp, if you have rsync. Not very functional if you don't :-)
- If 'obeyInstructions' is False, don't obey any applicable instructions.

muddled.deployments.collect.**deploy**(*builder*, *name*)
Create a collection deployment builder.

This adds a new rule linking the label `deployment:<name>/deployed` to the collection deployment builder.

You can then add assembly descriptors using the other utility functions in this module.

Dependencies get registered when you add an assembly descriptor.

muddled.deployments.cpio

cpio deployment.

Most commonly used to create Linux ramdisks, this deployment creates a CPIO archive from the relevant install directory and applies the relevant instructions.

Because python has no native CPIO support, we need to do this by creating a tar archive and then invoking cpio in copy-through mode to convert the archive to cpio. Ugh.

class muddled.deployments.cpio.**CIApplyChmod**
Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CIApplyChown**
Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CIApplyMknod**
Bases: *muddled.deployments.cpio.CpioInstructionImplementor*

apply(*builder*, *instr*, *role*, *target_base*, *hierarchy*)

class muddled.deployments.cpio.**CpioDeploymentBuilder**(*target_file*, *target_base*, *compressionMethod=None*, *pruneFunc=None*)

Bases: *muddled.depend.Action*

Builds the specified CPIO deployment.

- 'target_file' is the CPIO file to construct.
- 'target_base' is an array of pairs mapping labels to target locations, or (label, src) -> location
- 'compressionMethod' is the compression method to use, if any - gzip -> gzip, bzip2 -> bzip2.
- if 'pruneFunc' is not None, it is a function to be called like pruneFunc(Hierarchy) to prune the hierarchy prior to packing. Usually something like deb.deb_prune, it's intended to remove spurious stuff like manpages from initrds and the like.

attach_env(*builder*)

Attaches an environment containing:

MUDDLE_TARGET_LOCATION - the location in the target filesystem where this deployment will end up.

to every package label in this role.

build_label(*builder*, *label*)

Actually cpio everything up, following instructions appropriately.

class muddled.deployments.cpio.**CpioInstructionImplementor**

Bases: object

apply(*builder*, *instruction*, *role*, *path*)

class muddled.deployments.cpio.**CpioWrapper**(*builder*, *action*, *label*)

Bases: object

copy_from_role(*from_role*, *from_fragment*, *to_fragment*, *with_base=None*)

Copy the relative path from_fragment in from_role to to_fragment in the CPIO package or deployment given by 'action'

Use 'with_base' to change the base offset we apply when executing instructions; this is useful when using repeated copy_from_role() invocations to copy a subset of one role to a package/deployment.

done()

Call this once you've added all the roles you want; it attaches the deployment environment to them and generally finishes up

muddled.deployments.cpio.**create**(*builder*, *target_file*, *name*, *compressionMethod=None*, *pruneFunc=None*)

Create a CPIO deployment and return it.

- 'builder' is the muddle builder that is driving us
- 'target_file' is the name of the CPIO file we want to create. Note that this may include a sub-path (for instance, "fred/file.cpio" or even "/fred/file.cpio").
- 'name' is either:
  1. The name of the deployment that will contain this CPIO file (in the builder's default domain), or
  2. A deployment or package label, ditto
- 'comporessionMethod' is the compression method to use:
  - None means no compression

– 'gzip' means gzip

– 'bzip2' means bzip2

- if 'pruneFunc' is not None, it is a function to be called like pruneFunc(Hierarchy) to prune the hierarchy prior to packing. Usually something like deb.deb_prune, it's intended to remove spurious stuff like manpages from initrds and the like.

Normal usage is thus something like:

```
fw = cpio.create(builder, 'firmware.cpio', deployment)
fw.copy_from_role(role1, '', '/')
fw.copy_from_role(role2, 'bin', '/bin')
fw.done()
```

or:

```
fw = cpio.create(builder, 'firmware.cpio', package('firmware', role))
fw.copy_from_role(role, '', '/')
fw.done()
```

muddled.deployments.filedep

File deployment. This deployment just copies files into a role subdirectory in the /deployed directory, applying appropriate instructions.

class muddled.deployments.filedep.**FIApplyChmod**
Bases: *muddled.deployments.collect.CollectApplyChmod*

needs_privilege(*builder*, *instr*, *role*, *path*)

class muddled.deployments.filedep.**FIApplyMknod**
Bases: *muddled.deployments.collect.InstructionImplementor*

apply(*builder*, *instr*, *role*, *path*)

needs_privilege(*builder*, *instr*, *role*, *path*)

prepare(*builder*, *instr*, *role*, *path*)

class muddled.deployments.filedep.**FileDeploymentBuilder**(*roles*, *target_dir*)
Bases: *muddled.depend.Action*

Builds the specified file deployment

role is actually a list of (role, domain) pairs.

apply_instructions(*builder*, *label*)

attach_env(*builder*)
Attaches an environment containing:

MUDDLE_TARGET_LOCATION - the location in the target filesystem where this deployment will end up.

to every package label in this role.

build_label(*builder*, *label*)
Performs the actual build.

We actually do need to copy all files from install/ (where unprivileged processes can modify them) to deploy/ (where they can't).

Then we apply instructions to deploy.

deploy(*builder*, *label*)

muddled.deployments.filedep.**deploy**(*builder*, *target_dir*, *name*, *roles*)
Register a file deployment.

This is a convenience wrapper around deploy_with_domains().

'roles' is a sequence of role names. The deployment will take the roles specified, and build them into a deployment at deploy/[name].

More specifically, a rule will be created for label:

"deployment:<name>/deployed"

which depends on "package:*{<role}/postinstalled" (in the builder's default domain) for each <role> in 'roles'.

In other words, the deployment called 'name' will depend on the given roles (in the default domain) having been "finished" (postinstalled).

An "instructions applied" label "deployment:<name>/instructionsapplied" will also be created.

The deployment should eventually be located at 'target_dir'.

muddled.deployments.filedep.**deploy_with_domains**(*builder*, *target_dir*, *name*, *role_domains*)
Register a file deployment.

'role_domains' is a sequence of (role, domain) pairs. The deployment will take the roles and domains specified, and build them into a deployment at deploy/[name].

More specifically, a rule will be created for label:

"deployment:<name>/deployed"

which depends on "package:(<domain>)*{<role}/postinstalled" for each (<role>, <domain>) pair in 'role_domains'.

In other words, the deployment called 'name' will depend on the given roles in the appropriate domains having been "finished" (postinstalled).

An "instructions applied" label "deployment:<name>/instructionsapplied" will also be created.

The deployment should eventually be located at 'target_dir'.

muddled.deployments.tools

Tools deployment. This deployment merely adds the appropriate environment variables to use the tools in the given role install directories to everything in another list of deployments.

Instructions are ignored - there's no reason to follow them (yet) and it's simpler not to.

XXX Do we support this anymore?

class muddled.deployments.tools.**ToolsDeploymentBuilder**(*dependent_roles*)
Bases: *muddled.depend.Action*

Copy the dependent roles into the tools deployment.

build_label(*builder*, *label*)

deploy(*builder*, *label*)

muddled.deployments.tools.**attach_env**(*builder*, *role*, *env*, *name*)
Attach suitable environment variables for the given input role to the given environment store.

We set:

- LD_LIBRARY_PATH - Prepend $role_installl/lib
- PATH - Append $role_install/bin
- PKG_CONFIG_PATH - Prepend $role_install/lib/pkgconfig
- $role_TOOLS_PATH - Prepend $role_install/bin

The PATH/TOOLS_PATH stuff is so you can still locate tools which were in the path even if they've been overridden with your built tools.

muddled.deployments.tools.**deploy**(*builder*, *name*, *rolesThatUseThis=[]*, *rolesNeededForThis=[]*)

Register a tools deployment.

This is used to:

1. Set the environment for each role in 'rolesThatUseThis' so that PATH, LD_LIBRARY_PATH and PKG_CONFIG_PATH include the 'name' deployment
2. Make deployment:<name>/deployed depend upon the 'rolesNeededForThis'
3. Register cleanup for this deployment

The intent is that we have a "tools" deployment, which provides useful host tools (for instance, something to mangle a file in a particular manner). Those roles which need to use such tools in their builds (normally in a Makefile.muddle) then need to have the environment set appropriately to allow them to find the tools (and ideally, not system provided tools which mighth have the same name).

17.3.3 muddled.pkgs

muddled.pkgs.aptget

An apt-get package. When you try to build it, this package pulls in a pre-canned set of packages via apt-get.

class muddled.pkgs.aptget.**AptGetBuilder**(*name*, *role*, *pkgs_to_install*, *os_version=None*)

Bases: *muddled.pkg.PackageBuilder*

Make sure that particular OS packages have been installed.

The "build" action for AptGetBuilder uses the Debian tool apt-get to ensure that each package is installed.

Our arguments are:

- 'name' - the name of this builder
- 'role' - the role to which it belongs
- 'pkgs_to_install' - a sequence specifying which packages are to be installed.

Each item in the sequence 'pkgs_to_install' can be:

- the name of an OS package to install - for instance, 'libxml2-dev'

  (this is backwards compatible with how this class worked in the past)

- a Choice allowing a particular package to be selected according to the operating system.

  See "muddle doc Choice" for details on the Choice class.

  Note that a choice resulting in None (i.e., where the default value is None, and the default is selected) will not do anything.

  If 'os_version' is given, then it will be used as the version name, otherwise the result of calling utils.get_os_version_name() will be used.

We also allow a single string, or a single Choice, treated as if they were wrapped in a list.

already_installed(*pkg*)
Decide if the quoted debian package is already installed.

We use dpkg-query:

```
$ dpkg-query -W -f=\${Status}\n libreadline-dev
install ok installed
```

That third word means what it says (installed). Contrast with a package that is either not recognised or has not been downloaded at all:

```
$ dpkg-query -W -f=\${Status}\n a0d
dpkg-query: no packages found matching a0d
```

So we do some fairly simple processing of the output...

build_label(*builder*, *label*)
This time, build is the only one we care about.

muddled.pkgs.aptget.**depends_on_aptget**(*builder*, *name*, *role*, *pkg*, *pkg_role*)
Make a package dependant on a particular apt-builder.

- pkg - The package we want to add a dependency to. '*' is a good thing to add here ..

muddled.pkgs.aptget.**medium**(*builder*, *name*, *role*, *apt_pkgs*, *roles*, *os_version=None*)
Construct an apt-get package and make every package in the named roles depend on it.

Note that apt_pkgs can be an OS package name or a choices sequence - see the documentation for AptGetBuilder.

muddled.pkgs.aptget.**simple**(*builder*, *name*, *role*, *apt_pkgs*, *os_version=None*)
Construct an apt-get package in the given role with the given apt_pkgs.

Note that apt_pkgs can be an OS package name or a Choice - see the documentation for AptGetBuilder for more details.

For instance (note: not a real example - the dependencies don't make sense!):

```
from muddled.utils import Choice
from muddled.pkgs import aptget
aptget.simple(builder, "host_packages", "host_environment",
       [
       "gcc-multilib",
       "g++-multilib",
       "lib32ncurses5-dev",
       "lib32z1-dev",
       "bison",
       "flex",
       "gperf",
       "libx11-dev",
       # On Ubuntu 11 or 12, choose icedtea-7, otherwise icedtea-6
       Choice([ ("ubuntu 1[12].*", "icedtea-7-jre"),
                ("ubuntu *", "icedtea-6-jre") ]),
       # On Ubuntu 10 or later, use libgtiff5
       # On Ubuntu 3 through 9, use libgtiff4
       # Otherwise, just don't try to use libgtiff
       Choice([ ("ubuntu 1?", "libgtiff5"),
                ("ubuntu [3456789]", "libgtiff4"),
               None ])
       ])
```

muddled.pkgs.deb

Some code which sneakily steals binaries from Debian/Ubuntu.

Quite a lot of code for embedded systems can be grabbed pretty much directly from the relevant Ubuntu binary packages - this won't work with complex packages like exim4 without some external frobulation, since they have relatively complex postinstall steps, but it works supported architecture it's a quick route to externally maintained binaries which actually work and it avoids having to build absolutely everything in your linux yourself.

This package allows you to 'build' a package from a source file in a checkout which is a .deb. We run dpkg with enough force options to install it in the relevant install directory.

You still need to provide any relevant instruction files (we'll register <filename>.instructions.xml for you automatically if it exists).

We basically ignore the package database (there is one, but it's always empty and stored in the object directory).

class muddled.pkgs.deb.**DebAction**(*name*, *role*, *co*, *pkg_name*, *pkg_file*, *instr_name=None*, *postInstallMakefile=None*)

Bases: *muddled.pkg.PackageBuilder*

Use dpkg to extract debian archives from the given checkout into the install directory.

- co - is the checkout name in which the package resides.
- pkg_name - is the name of the package (dpkg needs it)
- pkg_file - is the name of the file the package is in, relative to the checkout directory.
- instr_name - is the name of the instruction file, if any.
- postInstallMakefile - if not None:

```
make -f postInstallMakefile <pkg-name>
```

will be run at post-install time to make links, etc.

build_label(*builder*, *label*)

Build the relevant label.

ensure_dirs(*builder*, *label*)

class muddled.pkgs.deb.**DebDevAction**(*name*, *role*, *co*, *pkg_name*, *pkg_file*, *instr_name=None*, *postInstallMakefile=None*, *nonDevCoName=None*, *nonDevPkgFile=None*)

Bases: *muddled.pkg.PackageBuilder*

Use dpkg to extract debian archives into obj/include and obj/lib directories so we can use them to build other packages.

As for a DebAction, really.

build_label(*builder*, *label*)

Actually install the dev package.

ensure_dirs(*builder*, *label*)

muddled.pkgs.deb.**deb_prune**(*h*)

Given a cpiofile hierarchy, prune it so that only the useful stuff is left.

We do this by lopping off directories, which is easy enough in cpiofile heirarchies.

muddled.pkgs.deb.**dev**(*builder*, *coName*, *name*, *roles*, *depends_on=[]*, *pkgFile=None*, *debName=None*, *nonDevCoName=None*, *nonDevDebName=None*, *instrFile=None*, *postInstallMakefile=None*)

A wrapper for 'deb.simple', with the "idDev" flag set True.

- nonDevCoName is the checkout in which the non-dev version of the package resides.
- nonDevDebName is the non-dev version of the package; this is sometimes needed because of the odd way in which debian packages the '.so' link in the dev package and the sofiles themselves into the non-dev.

muddled.pkgs.deb.**extract_into_obj**(*inv*, *co_name*, *label*, *pkg_file*)

muddled.pkgs.deb.**rewrite_links**(*inv*, *label*)

muddled.pkgs.deb.**simple**(*builder*, *coName*, *name*, *roles*, *depends_on=[]*, *pkgFile=None*, *debName=None*, *instrFile=None*, *postInstallMakefile=None*, *isDev=False*, *nonDevCoName=None*, *nonDevPkgFile=None*)

Build a package called 'name' from co_name / pkg_file with an instruction file called instr_file.

'name' is the name of the muddle package and of the debian package. if you want them different, set deb_name to something other than None.

Set isDev to True for a dev package, False for an ordinary binary package. Dev packages are installed into the object directory where MUDDLE_INC_DIRS etc. expects to look for them. Actual packages are installed into the installation directory where they will be transported to the target system.

muddled.pkgs.depmode_merge

Merge depmod databases.

Linux's depmod tool is absurdly stupid. It:

- Will not merge module databases from more than one location
- Will not deduce kernel versions from its naming scheme
- Will not accept absolute directory names in its configuration file
- Will not document its database format so I can use it directly

This module contains a depmod_merge() package. You create one with create() and then add every package which produces kernel modules to it with add_deps()/add_roles()/add() - whichever is most convenient.

By default, we expect your packages to leave their modules in <pkg_install_dir>/lib/modules/KERNEL_VERSION/... - you can change this by specifying 'subdir' to the add_XXX() routines and we will then expect <pkg_install_dir>/<subdir>/KERNEL_VERSION/...

(the KERNEL_VERSION is sadly a requirement of depmod. Go, um, depmod)

Your modules must have a '.ko' extension.

This package then depends on all those packages and when built will create a temporary database in its object directory containing all the .kos from all of the packages it's been told to look at.

It then scans <objdir>/lib/modules/d+.d+.* for all the kernel versions and depmod's them all.

It then copies module.* (i.e. all the module dependency files produced) back into <install_dir>/<subdir>/KERNEL_VERSION/ .

The dependency mechanism means that so long as you have your roles set up correctly, even if your sub-packages attempt to depmod on their own (as the kernel does), this package will always run later and overwrite the bad module dependencies with new, good ones.

If you are deploying multiple roles which each compute their dependencies separately, you will need to use the cpio ordering feature to make sure the right module database gets copied into your final cpio archive - we can't do this for you (yet) because there are no facilities yet for post-deployment operations and we can't create the dependency database on the way because the files don't all exist in the same place that we'd need them to to run depmod.

Grr. Aargh. *beat head against wall*. Etc.

class muddled.pkgs.depmod_merge.**MergeDepModBuilder**(*name*, *role*, *custom_depmod=None*)
Bases: *muddled.pkg.PackageBuilder*

Use depmod_merge to merge several depmod databases into a single result. We do this by writing a depmod.conf in our object directory and then invoking depmod.

custom_depmod tells us we want to use a custom depmod.

Constructor for the depmod package

self.components is a list of (label, subdir) pairs which we combine in our object directory to form a unified module database on which we can run depmod.

add_label(*label*, *subdir*)

build_label(*builder*, *label*)

muddled.pkgs.depmod_merge.**add**(*builder*, *merger*, *pkg*, *role*, *subdir='/lib/modules'*)

muddled.pkgs.depmod_merge.**add_deps**(*builder*, *merger*, *deps*, *subdir='/lib/modules'*)
Add a set of packages and roles to this merger as packages which create linux kernel modules. deps is a list of (pkg,role)

muddled.pkgs.depmod_merge.**add_roles**(*builder*, *merger*, *pkg*, *roles*, *subdir='/lib/modules'*)

muddled.pkgs.depmod_merge.**create**(*builder*, *name*, *role*, *pkgs_and_roles*, *custom_depmod=None*, *subdir='/lib/modules'*)
Create a depmod_merge . It will depend on each of the mentioned packages.

pkgs is a list of (pkg, role) pairs.

We return the depmod_merge we've created.

muddled.pkgs.depmod_merge.**predicate_is_kernel_module**(*name*)

muddled.pkgs.depmod_merge.**predicate_is_module_db**(*name*)
Decide if a full path name is list modules.*

muddled.pkgs.initscripts

Write an initialisation script into $(MUDDLE_TARGET_LOCATION)/bin/$(something)

This is really just using utils.subst_file() with the current environment, on a resource stored in resources/.

We also write a setvars script with a suitable set of variables for running code in the context of the deployment, and any variables you've set in the environment store retrieved with get_env_store().

class muddled.pkgs.initscripts.**InitScriptBuilder**(*name*, *role*, *script_name*, *deployments*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)
Bases: *muddled.pkg.PackageBuilder*

Build an init script.

build_label(*builder*, *label*)
Install is the only one we care about ..

muddled.pkgs.initscripts.**get_effective_env**(*builder*, *name*, *role*, *domain=None*)
Retrieve the effective runtime environment for this initscripts package. Note that setting variables here will have no effect.

muddled.pkgs.initscripts.**get_env**(*builder*, *name*, *role*, *domain=None*)
Retrieve an environment to which you can make changes which will be reflected in the generated init scripts. The

actual environment used will have extra values inserted from wildcarded environments - see get_effective_env() above.

muddled.pkgs.initscripts.**medium**(*builder*, *name*, *roles*, *script_name*, *deployments=[]*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)
Build an init script for the given roles.

muddled.pkgs.initscripts.**setup_default_env**(*builder*, *env_store*)
Set up the default environment for this initscript.

muddled.pkgs.initscripts.**simple**(*builder*, *name*, *role*, *script_name*, *deployments=[]*, *writeSetvarsSh=True*, *writeSetvarsPy=False*)
Build an init script for the given role.

muddled.pkgs.linux_kernel

autodoc: failed to import module u'muddled.pkgs.linux_kernel'; the following exception was raised: Traceback (most recent call last): File "/home/docs/checkouts/readthedocs.org/user_builds/muddle/envs/latest/local/lib/python2.7/site-packages/sphinx/ext/autodoc.py", line 385, in import_object __import__(self.modname) ImportError: No module named linux_kernel

muddled.pkgs.make

Some standard package implementations to cope with packages that use Make

class muddled.pkgs.make.**ExpandingMakeBuilder**(*name*, *role*, *co_name*, *archive_file*, *archive_dir*, *makefile='Makefile.muddle'*)
Bases: *muddled.pkgs.make.MakeBuilder*

A MakeBuilder that first expands an archive file.

A MakeBuilder that first expands an archive file.

For package 'name' in role 'role', look in checkout 'co_name' for archive 'archive_file'. Unpack that into $MUDDLE_OBJ, as 'archive_dir', with 'obj/' linked to it, and use 'makefile' to build it.

build_label(*builder*, *label*)
Build our label.

Cleverly, Richard didn't define anything for MakeBuilder to do at the PreConfigure step, which means we can safely do whatever we need to do in this subclass...

unpack_archive(*builder*, *label*)

class muddled.pkgs.make.**MakeBuilder**(*name*, *role*, *co*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *rewriteAutoconf=False*, *usesAutoconf=False*, *execRelPath=None*)
Bases: *muddled.pkg.PackageBuilder*

Use make to build your package from the given checkout.

We assume that the makefile is smart enough to build in the object directory, since any other strategy (e.g. convolutions involving cp) will lead to dependency-based disaster.

Constructor for the make package.

build_label(*builder*, *label*)
Build the relevant label. We'll assume that the checkout actually exists.

ensure_dirs(*builder*, *label*)
Make sure all the relevant directories exist.

muddled.pkgs.make.**attach_env**(*builder*, *name*, *role*, *checkout*, *domain=None*)
Write the environment which attaches MUDDLE_SRC to makefiles.

We retrieve the environment for `package:<name>{<role>}/*`, and set MUDDLE_SRC therein to the checkout path for 'checkout:<checkout>'.

muddled.pkgs.make.**deduce_makefile_name**(*makefile_name*, *per_role*, *role*)
Deduce our actual muddle Makefile name.

'makefile_name' is the base name. If it is None, then we use DEFAULT_MAKEFILE_NAME.

If 'per_role' is true, and 'role' is not None, then we add the extension '.<role>' to the end of the makefile name.

Abstracted here so that it can be used outside this module as well.

muddled.pkgs.make.**expanding_package**(*builder*, *name*, *archive_dir*, *role*, *co_name*, *co_dir*, *makefile='Makefile.muddle'*, *deps=None*, *archive_file=None*, *archive_ext='.tar.bz2'*)
Specify how to expand and build an archive file.

As normal, 'name' is the package name, 'role' is the role to build it in, 'co_name' is the name of the checkout, and 'co_dir' is the directory in which that lives.

We expect to unpack an archive

<co_dir>/<co_name>/<archive_dir><archive_ext>

into $(MUDDLE_OBJ)/<archive_dir>. (NB: if the archive file does not expand into a directory of the obvious name, you can specify the archive file name separately, using 'archive_file').

So, for instance, all of our X11 "stuff" lives in checkout "X11R7.5" which is put into directory "x11" – i.e., "src/X11/X11R7.5".

That lets us keep stuff together in the repository, without leading to a great many packages that are of no direct interest to anyone else.

Within that we then have various muddle makefiles, and a set of .tar.bz archive files.

1. The archive file expands into a directory called 'archive_dir'
2. It is assumed that the archive file is named 'archive_dir' + 'archive_ext'. If this is not so, then specify 'archive_file' (and/or 'archive_ext') appropriately.

This function is used to say: take the named archive file, use package name 'name', unpack the archive file into $(MUDDLE_OBJ_OBJ), and build it using the named muddle makefile.

This allows various things to be build with the same makefile, which is useful for (for instance) X11 proto[type] archives.

Note that in $(MUDDLE_OBJ), 'obj' (i.e., $(MUDDLE_OBJ_OBJ)) will be a soft link to the expanded archive directory.

muddled.pkgs.make.**medium**(*builder*, *name*, *roles*, *checkout*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)
Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

muddled.pkgs.make.**multilevel**(*builder*, *name*, *roles*, *co_dir=None*, *co_name=None*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *repo_relative=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

muddled.pkgs.make.**simple**(*builder*, *name*, *role*, *checkout*, *rev=None*, *branch=None*, *simpleCheckout=False*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, called name with role role from the sources in checkout checkout.

- simpleCheckout - If True, register the checkout too.
- config - If True, we have make config. If false, we don't.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.
- **usesAutoconf - If True, this package is given access to .la and .pc** files from things it depends on.
- rewriteAutoconf - If True, we will rewrite .la and .pc files in the output directory so that packages which use autoconf continue to depend correctly. Intended for use with the MUDDLE_PKGCONFIG_DIRS environment variable.
- **execRelPath - Where, relative to the object directory, do we find** binaries for this package?

muddled.pkgs.make.**single**(*builder*, *name*, *role*, *deps=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

A simple make package with a single checkout named after the package and a single role.

muddled.pkgs.make.**twolevel**(*builder*, *name*, *roles*, *co_dir=None*, *co_name=None*, *rev=None*, *branch=None*, *deps=None*, *dep_tag='preconfig'*, *simpleCheckout=True*, *config=True*, *perRoleMakefiles=False*, *makefileName='Makefile.muddle'*, *repo_relative=None*, *usesAutoconf=False*, *rewriteAutoconf=False*, *execRelPath=None*)

Build a package controlled by make, in the given roles with the given dependencies in each role.

- simpleCheckout - If True, register the checkout as simple checkout too.
- dep_tag - The tag to depend on being installed before you'll build.
- perRoleMakefiles - If True, we run 'make -f Makefile.<rolename>' instead of just make.

muddled.pkgs.version

Write a version.xml file containing version information for the current build

class muddled.pkgs.version.**VersionBuilder**(*name*, *role*, *filename*, *swname=None*, *version=None*, *build=None*, *withDate=True*, *withUser=True*, *withMachine=True*)

Bases: *muddled.pkg.PackageBuilder*

Write a version number file

Constructor for the version package type

build_label(*builder*, *label*)
Build the version.xml file.

dir_name(*builder*)

erase_version_file(*builder*)
Erase the version file.

file_name(*builder*)

write_elem(*f*, *elem*, *val*)

write_version_file(*builder*)
Write the version file

muddled.pkgs.version.**simple**(*builder*, *name*, *roles*, *filename='/version.xml'*, *swname=None*, *version=None*, *build=None*, *withDate=True*, *withUser=True*, *withMachine=True*)

17.3.4 muddled.resources

Not Python files:

- `c_env.c` - boilerplate for accessing environments from C.
- `initscript.sh` - a generic `init` script

17.3.5 muddled.vcs

muddled.vcs.bazaar

Muddle support for Bazaar.

Note that Bazaar does not support "branches" in the muddle sense. Bazaar itself uses the "bzr branch" command to make a clone of a repository. It does not (or did not at time of writing) support lightweight branching in the manner of git - i.e., separate branches stored within the same clone. Thus the "branch" argument of a Repository class is not supported for Bazaar.

Available Bazaar specific options are:

- no_follow: In a build description that has set:

```
builder.follow_build_desc_branch = True
```

then a Bazaar repository can either:

1. Specify a particular revision
2. Choose a different repository location (presumably a bazaar "branch"), and set the no_follow option to True
3. Continue using the original repository without setting a revision (almost certainly not sensible, but still), and set the no_follow option to True.

If none of these are done, then muddle will fail with a complaint like:

```
The build description wants checkouts to follow branch '<branch-name>',
but checkout <co-name> uses VCS Bazaar for which we do not support branching.
The build description should specify a revision for checkout <co-name>.
```

class muddled.vcs.bazaar.**Bazaar**
Bases: *muddled.version_control.VersionControlSystem*

Provide version control operations for Bazaar

add_files(*files=None*, *verbose=True*)
If files are given, add them, but do not commit.

Will be called in the actual checkout's directory.

allows_relative_in_repo()

checkout(*repo*, *co_leaf*, *options*, *verbose=True*)
Checkout (clone) a given checkout.

Will be called in the parent directory of the checkout.

Expected to create a directory called <co_leaf> therein.

commit(*repo*, *options*, *verbose=True*)
Will be called in the actual checkout's directory.

get_file_content(*url*, *verbose=True*)
Retrieve a file's content via BZR.

get_vcs_special_files()

goto_revision(*revision*, *branch=None*, *repo=None*, *verbose=False*)
Make the specified revision current.

Note that this may leave the working data (the actual checkout directory) in an odd state, in which it is not sensible to commit, depending on the VCS and the revision.

Will be called in the actual checkout's directory.

Raises GiveUp if there is no such revision, or if a branch is given.

init_directory(*verbose=True*)
If the directory does not appear to have had '<vcs> init' run in it, then do so first.

Will be called in the actual checkout's directory.

merge(*other_repo*, *options*, *verbose=True*)
Merge 'other_repo' into the local repository and working tree,

'bzr merge' will not (by default) merge if there are uncommitted changes in the destination (i.e., local) tree. This is what we want.

Will be called in the actual checkout's directory.

pull(*repo*, *options*, *upstream=None*, *verbose=True*)
Pull changes, but don't do a merge.

Will be called in the actual checkout's directory.

push(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

reparent(*co_leaf*, *remote_repo*, *options*, *force=False*, *verbose=True*)
Re-associate the local repository with its original remote repository,

- 'co_leaf' should be the name of this checkout directory, for use in messages reporting what we are doing. Note that we are called already *in* that directory, though.
- 'remote_repo' is the repository we would like to associate it with.

`bzr info` is your friend for finding out if the checkout is already associated with a remote repository. The "parent branch" is used for pulling and merging (and is what we set). If present, the "push branch" is used for pushing.

If force is true, we set "parent branch", and delete "push branch" (so it will default to the "parent branch").

If force is false, we only set "parent branch", and then only if it is not set.

The actual information is held in <checkout-dir>/.bzr/branch/branch.conf, which is a .INI file.

revision_to_checkout (*repo*, *co_leaf*, *options*, *force=False*, *before=None*, *verbose=True*)
Determine a revision id for this checkout, usable to check it out again.

- 'co_leaf' should be the name of this checkout directory, for use in messages reporting what we are doing. Note that we are called already *in* that directory, though.

If 'force' is true, then if we can't get one from bzr, and it seems "reasonable" to do so, use the original revision from the muddle depend file (if it is not HEAD).

If 'before' is given, it should be a string describing a date/time, and the revision id chosen will be the last revision at or before that date/time.

Note: This depends upon what the VCS concerned actually supports. This feature is experimental. XXX NOT YET IMPLEMENTED XXX

'bzr revno' always returns a simple integer (or so I believe)

'bzr version-info' returns several lines, including:

```
revision-id: <something>
revno: <xxx>
```

where <xxx> is the same number as 'bzr revno', and <something> will be different depending on whether we're "the same" as the far repository.

If the –check-clean flag is used, then there will also be a line of the form:

```
clean: True
```

indicating whether the source tree contains uncommitted changes (although not whether it is matching the far repository).

So ideally we would (1) grumble if not clean, and (2) grumble if our revision id was different than after the last push/pull/checkout

Well, 'bzr missing' should show unmerged/unpulled revisions between two branches, so if it ends "Branches are up to date" then that may be useful. Or no output with '-q' if they're OK. (needs to ignore stderr output, since I get that for mismatch in Bazaar network protocols)

status (*repo*, *options=None*, *branch=None*, *verbose=False*, *quick=False*)
Will be called in the actual checkout's directory.

Return status text or None if there is no interesting status.

muddled.vcs.file

Muddle support for naive file copying.

class muddled.vcs.file.**File**
Bases: *muddled.version_control.VersionControlSystem*

Provide version control operations for simple file copying

add_files (*files=None*, *verbose=True*)
If files are given, add them, but do not commit.

Will be called in the actual checkout's directory.

allows_relative_in_repo ()

checkout (*repo*, *co_leaf*, *options*, *verbose=True*)
Clone a given checkout.

Will be called in the parent directory of the checkout.

Expected to create a directory called <co_leaf> therein.

commit (*repo*, *options*, *verbose=True*, *quick=False*)
Will be called in the actual checkout's directory.

get_file_content (*url*, *verbose=True*)
Retrieve a file's content via Subversion.

init_directory (*verbose=True*)
If the directory does not appear to have had '<vcs> init' run in it, then do so first.

Will be called in the actual checkout's directory.

merge (*other_repo*, *options*, *verbose=True*)
Merge 'other_repo' into the local repository and working tree,

Just copies everything again. This is an imperfect sort of "merge".

pull (*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

Just copies everything again.

push (*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

We refuse to copy anything back to the original directory.

reparent (*co_dir*, *remote_repo*, *options*, *force=False*, *verbose=True*)

revision_to_checkout (*repo*, *co_leaf*, *options*, *force=False*, *before=None*, *verbose=False*)
Determine a revision id for this checkout, usable to check it out again.

status (*repo*, *options*, *branch=None*, *quick=None*)
Status is not supported for 'file'.

Just report 'nothing important has happened'.

muddled.vcs.git

Muddle suppport for Git.

TODO: The following needs rewriting after work for issue 225

- muddle checkout

  Clones the appropriate checkout with `git clone`.

  If the build description (by whatever means) requires a branch, then `git clone -b <branch>` is used. If no branch is specified, we default to `git clone -b master`.

If a shallow checkout is selected, then the `--depth 1` switch is added to the clone command. Note that there are some restrictions on what can be done with a shallow checkout - git says:

```
A shallow repository has a number of limitations (*you cannot clone or
fetch from it, nor push from nor into it*), but is adequate if you are
only interested in the recent history of a large project with a long
history, and would want to send in fixes as patches.
```

(the emphasis is mine).

If a revision is requested, then `git checkout` is used to check it out.

If a branch *and* a revision are requested, then muddle checks to see if cloning the branch gave the correct revision, and only does the `git checkout` if it did not. This avoids unnecessary detached HEADs,

Note: the checking of the revision id is very simple, and assumes that the revision is specified as a full SHA1 string (it is compared with the output of `git rev-parse HEAD`).

- muddle pull, muddle pull-upstream

  This does a `git fetch` followed by a fast-forwards `git merge`.

  If the build description specifies a particular revision, and the checkout is already at that revision, then an error will be reported, saying that.

  If there are any local changes uncommitted, or untracked files, then appropriate error messages will also be reported.

  If the build description specified a branch for this repository (by whatever means), then we will first go to that branch. If no branch was specified, we first go to "master".

  The command checks that the remote is configured as such, then does `git fetch`. If a revision was specified, it then checks out that revision, otherwise it does `git merge --ff-only`, which will merge in the fetch if it doesn't require human interaction.

- muddle push, muddle push-upstream

  If the checkout is marked as "shallow', or is on a detached HEAD, then an appropriate error message will be given.

  The command checks that the remote is configured as such, then does `git push` of the current branch. If the branch does not exist at the far end, it will be created.

- muddle merge

  This is identical to "muddle pull", except that instead of doing a fast-forward merge, it does a simple `git merge`, allowing human interaction if necessary.

- muddle commit

  Simply runs `git commit -a`.

- muddle status

  This first does `git status --porcelain`. If that does not return anything, it then determines the SHA1 for the local HEAD, and the SHA1 for the equivalent HEAD in the remote repository. If these are different (so presumably the remote repository is ahead of the local one), then it reports as much,

  Note that a normal `git status` does not talk to the remote repository, and is thus fast. If this command does talk over the network, it can be rather slower.

- muddle reparent

  Re-associates the local repository's "origin" with the remote repository indicated by the build description. It tries to detect if this is necessary first.

Available git specific options are:

- shallow_checkout: If True, then only clone to a depth of 1 (i.e., pass the git switch "–depth 1"). If False, then no effect. The default is False.

  This is typically of use when cloning the Linux kernel (or some other large tree with a great deal of history), when one is not expecting to modify the checkout in any way in the future (i.e., neither to push it nor to pull it again).

  If 'shallow_checkout' is specified, then "muddle push" will refuse to do anything.

class muddled.vcs.git.**Git**
Bases: *muddled.version_control.VersionControlSystem*

Provide version control operations for Git

add_files(*files=None*, *verbose=True*)
If files are given, add them, but do not commit.

Will be called in the actual checkout's directory.

allows_relative_in_repo()
TODO: Check that this is correct!

branch_exists(*branch*)
Is there a branch of this name?

Will be called in the actual checkout's directory.

checkout(*repo*, *co_leaf*, *options*, *verbose=True*)
Clone a given checkout.

Will be called in the parent directory of the checkout.

Expected to create a directory called <co_leaf> therein.

commit(*repo*, *options*, *verbose=True*)
Will be called in the actual checkout's directory.

Does 'git commit -a' - i.e., this implicitly does 'git add' for you. This is a contentious choice, and needs review.

create_branch(*branch*, *verbose=False*)
Create a branch of the given name.

Will be called in the actual checkout's directory.

Also sets up the equivalent remote.

It is an error if the branch already exists, in which case a GiveUp exception will be raised.

get_current_branch()
Return the name of the current branch.

Will be called in the actual checkout's directory.

Returns None if we are not on a branch (e.g., a detached HEAD)

get_vcs_special_files()

goto_branch(*branch*, *verbose=False*)
Make the named branch the current branch.

Will be called in the actual checkout's directory.

It is an error if the branch does not exist, in which case a GiveUp exception will be raised.

goto_revision(*revision*, *branch=None*, *repo=None*, *verbose=False*)
Make the specified revision current.

Note that this may leave the working data (the actual checkout directory) in an odd state, in which it is not sensible to commit, depending on the VCS and the revision.

Will be called in the actual checkout's directory.

If a branch name is given, we will go to that branch first, and see if we already got to the correct revision. Note that the check for this assumes that 'revision' is a full SHA1, so is a bit simplistic. If we don't appear to be at the required revision, we'll then go there as normal.

Raises GiveUp if there is no such revision, or no such branch.

init_directory(*verbose=True*)
If the directory does not appear to have had '<vcs> init' run in it, then do so first.

Will be called in the actual checkout's directory.

merge(*repo*, *options*, *verbose=True*)
Merge changes from 'repo' into the local repository and working tree.

Will be called in the actual checkout's directory.

Broadly, does a 'git fetch' followed by a merge. Be aware that this last may require user interaction.

If a fast-forward merge is possible, then this identical to doing a "muddle pull".

If the build description specifies a particular revision, then if it was already at that revision, nothing needs doing. Otherwise, the 'git fetch' is done and then the specified revision is checked out using 'git checkout'.

pull(*repo*, *options*, *upstream=None*, *verbose=True*)
Pull changes from 'repo' into the local repository and working tree.

Will be called in the actual checkout's directory.

Broadly, does a 'git fetch' followed by a fast-forward merge - so it will only merge if it is obvious how to do it.

If the build description specifies a particular revision, then if it was already at that revision, nothing needs doing. Otherwise, the 'git fetch' is done and then the specified revision is checked out using 'git checkout'.

push(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

XXX Should we grumble if the 'effective' branch is not the same as XXX the branch that is currently checked out?

reparent(*co_dir*, *remote_repo*, *options*, *force=False*, *verbose=True*)
Re-associate the local repository with its original remote repository,

revision_to_checkout(*repo*, *co_leaf*, *options*, *force=False*, *before=None*, *verbose=True*)
Determine a revision id for this checkout, usable to check it out again.

1.Document

2.Review the code to see if we now support versions of git that would allow us to do this more sensibly

If 'before' is given, it should be a string describing a date/time, and the revision id chosen will be the last revision at or before that date/time.

NB: if 'before' is specified, 'force' is ignored.

XXX TODO: Needs reviewing given we're using a later version of git now

status(*repo*, *options*, *quick=False*)
Will be called in the actual checkout's directory.

Return status text or None if there is no interesting status.

supports_branching()

muddled.vcs.git.**expand_revision**(*revision*)
Given something that names a revision, return its full SHA1.

Raises GiveUp if the revision appears non-existent or ambiguous

muddled.vcs.git.**git_supports_ff_only**()
Does my git support –ff-only?

muddled.vcs.svn

Muddle support for Subversion

Note that Subversion does not support "branches" in the muddle sense. Subversion branches are handled by a different mechanism, and in a muddle sense are more like inner paths of a Repository. At the moment muddle does not provide any special support for Subversion branches.

Available Subversion specific options are:

- no_follow: In a build description that has set:

```
builder.follow_build_desc_branch = True
```

then a Subversion repository can either:

1. Specify a particular revision
2. Choose a different repository location (presumably a subversion "branch"), and set the no_follow option to True
3. Continue using the original repository without setting a revision (almost certainly not sensible, but still), and set the no_follow option to True.

If none of these are done, then muddle will fail with a complaint like:

```
The build description wants checkouts to follow branch '<branch-name>',
but checkout <co-name> uses VCS Subversion for which we do not support branching.
The build description should specify a revision for checkout <co-name>.
```

class muddled.vcs.svn.**Subversion**
Bases: *muddled.version_control.VersionControlSystem*

Provide version control operations for Subversion

add_files(*files=None*, *verbose=True*)
If files are given, add them, but do not commit.

Will be called in the actual checkout's directory.

allows_relative_in_repo()

checkout(*repo*, *co_leaf*, *options*, *verbose=True*)
Clone a given checkout.

Will be called in the parent directory of the checkout.

Expected to create a directory called <co_leaf> therein.

commit(*repo*, *options*, *verbose=True*)
Will be called in the actual checkout's directory.

This command does nothing, because Subversion does not have a local repository. Use 'muddle push' instead.

get_file_content(*url*, *verbose=True*)
Retrieve a file's content via Subversion.

get_vcs_special_files()

goto_revision(*revision*, *branch=None*, *repo=None*, *verbose=False*)
Make the specified revision current.

Note that this may leave the working data (the actual checkout directory) in an odd state, in which it is not sensible to commit, depending on the VCS and the revision.

Will be called in the actual checkout's directory.

Raises GiveUp if there is no such revision, or if branch is given.

init_directory(*verbose=True*)
If the directory does not appear to have had '<vcs> init' run in it, then do so first.

Will be called in the actual checkout's directory.

merge(*other_repo*, *options*, *verbose=True*)
Merge 'other_repo' into the local repository and working tree,

This runs Subversion's "update" - its "merge" command does something different.

Will be called in the actual checkout's directory.

must_pull_before_commit(*options*)
Subversion recommends doing 'commit' before "pull" (i.e., pull/update)

pull(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

This runs Subversion's "update", but only if no merging will be needed. That is, first it runs "svn status", and if any lines contain a "C" (for Conflict) in columns 0, 1 or 6, then it will not perform the update.

("svn help status" would call those columns 1, 2 and 7)

push(*repo*, *options*, *upstream=None*, *verbose=True*)
Will be called in the actual checkout's directory.

This actually does a "svn commit", i.e., committing to the remote repository (which is the only one subversion has).

reparent(*co_dir*, *remote_repo*, *options*, *force=False*, *verbose=True*)
Re-associate the local repository with its original remote repository,

Our subversion support does not provide this.

revision_to_checkout(*repo*, *co_leaf*, *options*, *force=False*, *before=None*, *verbose=False*)
Determine a revision id for this checkout, usable to check it out again.

Uses 'svnversion', which I believe is installed as standard with 'svn'

For the moment, at lease, the 'force' argument is ignored (so the working copy must be be equivalent to the repository).

status(*repo*, *options*, *quick=False*)
Will be called in the actual checkout's directory.

Return status text or None if there is no interesting status.

CHAPTER 18

The muddle documentation and sphinx and ReadTheDocs

18.1 Pre-built documentation

For your comfort and convenience, a pre-built version of the muddle documentation is available at:

http://muddle.readthedocs.org/

This is hosted by Read the Docs, who are wonderful people for providing such a facility. The documentation should get rebuilt on each push to the repository, which means that it should always be up-to-date.

18.2 Building the documentation

The muddle documentation is built using Sphinx.

Note: It needs (at least) version 0.6 of Sphinx, which is later than the version installed via apt-get on Ubuntu 8.10. The best way to upgrade is with easy_install, as described on the Sphinx website.

You may also also need graphviz (which provides `dot`).

As said above, the easiest way to get the documentation is via Read the Docs, but if you want to build a copy yourself, then all you need to do is install Sphinx, and use the Makefile:

```
$ cd docs
$ make html
```

18.3 The Python bindings

Read the muddle-package.txt file to see how individual classes and functions within the muddled package are documented. Obviously, if you add, remove or rename such, you may need to alter this file – please do so appropriately.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

m

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X