Bazaar Testing Guide

Release 2.7.1dev

Bazaar Developers

March 21, 2019

CONTENTS

CHAPTER

ONE

THE IMPORTANCE OF TESTING

Reliability is a critical success factor for any version control system. We want Bazaar to be highly reliable across multiple platforms while evolving over time to meet the needs of its community.

In a nutshell, this is what we expect and encourage:

- New functionality should have test cases. Preferably write the test before writing the code.

  In general, you can test at either the command-line level or the internal API level. See Writing tests below for more detail.

- Try to practice Test-Driven Development: before fixing a bug, write a test case so that it does not regress. Similarly for adding a new feature: write a test case for a small version of the new feature before starting on the code itself. Check the test fails on the old code, then add the feature or fix and check it passes.

By doing these things, the Bazaar team gets increased confidence that changes do what they claim to do, whether provided by the core team or by community members. Equally importantly, we can be surer that changes down the track do not break new features or bug fixes that you are contributing today.

As of September 2009, Bazaar ships with a test suite containing over 23,000 tests and growing. We are proud of it and want to remain so. As community members, we all benefit from it. Would you trust version control on your project to a product *without* a test suite like Bazaar has?

CHAPTER

TWO

RUNNING THE TEST SUITE

As of Bazaar 2.1, you must have the testtools library installed to run the bzr test suite.

To test all of Bazaar, just run:

```
bzr selftest
```

With `--verbose` bzr will print the name of every test as it is run.

This should always pass, whether run from a source tree or an installed copy of Bazaar. Please investigate and/or report any failures.

2.1 Running particular tests

Currently, bzr selftest is used to invoke tests. You can provide a pattern argument to run a subset. For example, to run just the blackbox tests, run:

```
./bzr selftest -v blackbox
```

To skip a particular test (or set of tests), use the –exclude option (shorthand -x) like so:

```
./bzr selftest -v -x blackbox
```

To ensure that all tests are being run and succeeding, you can use the –strict option which will fail if there are any missing features or known failures, like so:

```
./bzr selftest --strict
```

To list tests without running them, use the –list-only option like so:

```
./bzr selftest --list-only
```

This option can be combined with other selftest options (like -x) and filter patterns to understand their effect.

Once you understand how to create a list of tests, you can use the –load-list option to run only a restricted set of tests that you kept in a file, one test id by line. Keep in mind that this will never be sufficient to validate your modifications, you still need to run the full test suite for that, but using it can help in some cases (like running only the failed tests for some time):

```
./bzr selftest -- load-list my_failing_tests
```

This option can also be combined with other selftest options, including patterns. It has some drawbacks though, the list can become out of date pretty quick when doing Test Driven Development.

To address this concern, there is another way to run a restricted set of tests: the –starting-with option will run only the tests whose name starts with the specified string. It will also avoid loading the other tests and as a consequence starts running your tests quicker:

```
./bzr selftest --starting-with bzrlib.blackbox
```

This option can be combined with all the other selftest options including –load-list. The later is rarely used but allows to run a subset of a list of failing tests for example.

2.2 Disabling plugins

To test only the bzr core, ignoring any plugins you may have installed, use:

```
./bzr --no-plugins selftest
```

2.3 Disabling crash reporting

By default Bazaar uses apport to report program crashes. In developing Bazaar it's normal and expected to have it crash from time to time, at least because a test failed if for no other reason.

Therefore you should probably add `debug_flags = no_apport` to your `bazaar.conf` file (in `~/.bazaar/` on Unix), so that failures just print a traceback rather than writing a crash file.

2.4 Test suite debug flags

Similar to the global `-Dfoo` debug options, bzr selftest accepts `-E=foo` debug flags. These flags are:

allow_debug do *not* clear the global debug flags when running a test. This can provide useful logging to help debug test failures when used with e.g. `bzr -Dhpss selftest -E=allow_debug`

Note that this will probably cause some tests to fail, because they don't expect to run with any debug flags on.

2.5 Using subunit

Bazaar can optionally produce output in the machine-readable subunit format, so that test output can be post-processed by various tools. To generate a subunit test stream:

```
$ ./bzr selftest --subunit
```

Processing such a stream can be done using a variety of tools including:

- The builtin `subunit2pyunit`, `subunit-filter`, `subunit-ls`, `subunit2junitxml` from the subunit project.
- tribunal, a GUI for showing test results.
- testrepository, a tool for gathering and managing test runs.

2.6 Using testrepository

Bazaar ships with a config file for testrepository. This can be very useful for keeping track of failing tests and doing general workflow support. To run tests using testrepository:

```
$ testr run
```

To run only failing tests:

```
$ testr run --failing
```

To run only some tests, without plugins:

```
$ test run test_selftest -- --no-plugins
```

See the testrepository documentation for more details.

2.7 Babune continuous integration

We have a Hudson continuous-integration system that automatically runs tests across various platforms. In the future we plan to add more combinations including testing plugins. See <http://babune.ladeuil.net:24842/>. (Babune = Bazaar Buildbot Network.)

2.8 Running tests in parallel

Bazaar can use subunit to spawn multiple test processes. There is slightly more chance you will hit ordering or timing-dependent bugs but it's much faster:

```
$ ./bzr selftest --parallel=fork
```

Note that you will need the Subunit library <https://launchpad.net/subunit/> to use this, which is in `python-subunit` on Ubuntu.

2.9 Running tests from a ramdisk

The tests create and delete a lot of temporary files. In some cases you can make the test suite run much faster by running it on a ramdisk. For example:

```
$ sudo mkdir /ram
$ sudo mount -t tmpfs none /ram
$ TMPDIR=/ram ./bzr selftest ...
```

You could also change `/tmp` in `/etc/fstab` to have type `tmpfs`, if you don't mind possibly losing other files in there when the machine restarts. Add this line (if there is none for `/tmp` already):

```
none            /tmp            tmpfs   defaults        0       0
```

With a 6-core machine and `--parallel=fork` using a tmpfs doubles the test execution speed.

CHAPTER

THREE

WRITING TESTS

Normally you should add or update a test for all bug fixes or new features in Bazaar.

3.1 Where should I put a new test?

Bzrlib's tests are organised by the type of test. Most of the tests in bzr's test suite belong to one of these categories:

- Unit tests
- Blackbox (UI) tests
- Per-implementation tests
- Doctests

A quick description of these test types and where they belong in bzrlib's source follows. Not all tests fall neatly into one of these categories; in those cases use your judgement.

3.1.1 Unit tests

Unit tests make up the bulk of our test suite. These are tests that are focused on exercising a single, specific unit of the code as directly as possible. Each unit test is generally fairly short and runs very quickly.

They are found in `bzrlib/tests/test_*.py`. So in general tests should be placed in a file named test_FOO.py where FOO is the logical thing under test.

For example, tests for merge3 in bzrlib belong in bzrlib/tests/test_merge3.py. See bzrlib/tests/test_sampler.py for a template test script.

3.1.2 Blackbox (UI) tests

Tests can be written for the UI or for individual areas of the library. Choose whichever is appropriate: if adding a new command, or a new command option, then you should be writing a UI test. If you are both adding UI functionality and library functionality, you will want to write tests for both the UI and the core behaviours. We call UI tests 'blackbox' tests and they belong in `bzrlib/tests/blackbox/*.py`.

When writing blackbox tests please honour the following conventions:

1. Place the tests for the command 'name' in bzrlib/tests/blackbox/test_name.py. This makes it easy for developers to locate the test script for a faulty command.

2. Use the 'self.run_bzr("name")' utility function to invoke the command rather than running bzr in a subprocess or invoking the cmd_object.run() method directly. This is a lot faster than subprocesses and generates the same logging output as running it in a subprocess (which invoking the method directly does not).
3. Only test the one command in a single test script. Use the bzrlib library when setting up tests and when evaluating the side-effects of the command. We do this so that the library api has continual pressure on it to be as functional as the command line in a simple manner, and to isolate knock-on effects throughout the blackbox test suite when a command changes its name or signature. Ideally only the tests for a given command are affected when a given command is changed.
4. If you have a test which does actually require running bzr in a subprocess you can use `run_bzr_subprocess`. By default the spawned process will not load plugins unless `--allow-plugins` is supplied.

3.1.3 Per-implementation tests

Per-implementation tests are tests that are defined once and then run against multiple implementations of an interface. For example, `per_transport.py` defines tests that all Transport implementations (local filesystem, HTTP, and so on) must pass. They are found in `bzrlib/tests/per_*/*.py`, and `bzrlib/tests/per_*.py`.

These are really a sub-category of unit tests, but an important one.

Along the same lines are tests for extension modules. We generally have both a pure-python and a compiled implementation for each module. As such, we want to run the same tests against both implementations. These can generally be found in `bzrlib/tests/*__*.py` since extension modules are usually prefixed with an underscore. Since there are only two implementations, we have a helper function `bzrlib.tests.permute_for_extension`, which can simplify the `load_tests` implementation.

3.1.4 Doctests

We make selective use of doctests. In general they should provide *examples* within the API documentation which can incidentally be tested. We don't try to test every important case using doctests — regular Python tests are generally a better solution. That is, we just use doctests to make our documentation testable, rather than as a way to make tests. Be aware that doctests are not as well isolated as the unit tests, if you need more isolation, you're likely want to write unit tests anyway if only to get a better control of the test environment.

Most of these are in `bzrlib/doc/api`. More additions are welcome.

There is an *assertDoctestExampleMatches* method in *bzrlib.tests.TestCase* that allows you to match against doctest-style string templates (including `...` to skip sections) from regular Python tests.

3.2 Shell-like tests

`bzrlib/tests/script.py` allows users to write tests in a syntax very close to a shell session, using a restricted and limited set of commands that should be enough to mimic most of the behaviours.

A script is a set of commands, each command is composed of:

- one mandatory command line,
- one optional set of input lines to feed the command,
- one optional set of output expected lines,
- one optional set of error expected lines.

Input, output and error lines can be specified in any order.

Except for the expected output, all lines start with a special string (based on their origin when used under a Unix shell):

- '$ ' for the command,
- '<' for input,
- nothing for output,
- '2>' for errors,

Comments can be added anywhere, they start with '#' and end with the line.

The execution stops as soon as an expected output or an expected error is not matched.

If output occurs and no output is expected, the execution stops and the test fails. If unexpected output occurs on the standard error, then execution stops and the test fails.

If an error occurs and no expected error is specified, the execution stops.

An error is defined by a returned status different from zero, not by the presence of text on the error stream.

The matching is done on a full string comparison basis unless '...' is used, in which case expected output/errors can be less precise.

Examples:

The following will succeeds only if 'bzr add' outputs 'adding file':

```
$ bzr add file
>adding file
```

If you want the command to succeed for any output, just use:

```
$ bzr add file
...
2>...
```

or use the `--quiet` option:

```
$ bzr add -q file
```

The following will stop with an error:

```
$ bzr not-a-command
```

If you want it to succeed, use:

```
$ bzr not-a-command
2> bzr: ERROR: unknown command "not-a-command"
```

You can use ellipsis (...) to replace any piece of text you don't want to be matched exactly:

```
$ bzr branch not-a-branch
2>bzr: ERROR: Not a branch...not-a-branch/".
```

This can be used to ignore entire lines too:

```
$ cat
<first line
<second line
<third line
# And here we explain that surprising fourth line
<fourth line
```

```
<last line
>first line
>...
>last line
```

You can check the content of a file with cat:

```
$ cat <file
>expected content
```

You can also check the existence of a file with cat, the following will fail if the file doesn't exist:

```
$ cat file
```

You can run files containing shell-like scripts with:

```
$ bzr test-script <script>
```

where `<script>` is the path to the file containing the shell-like script.

The actual use of ScriptRunner within a TestCase looks something like this:

```
from bzrlib.tests import script

def test_unshelve_keep(self):
    # some setup here
    script.run_script(self, '''
        $ bzr add -q file
        $ bzr shelve -q --all -m Foo
        $ bzr shelve --list
        1: Foo
        $ bzr unshelve -q --keep
        $ bzr shelve --list
        1: Foo
        $ cat file
        contents of file
        ''')
```

You can also test commands that read user interaction:

```
def test_confirm_action(self):
    """You can write tests that demonstrate user confirmation"""
    commands.builtin_command_registry.register(cmd_test_confirm)
    self.addCleanup(commands.builtin_command_registry.remove, 'test-confirm')
    self.run_script("""
        $ bzr test-confirm
        2>Really do it? [y/n]:
        <yes
        yes
        """)
```

To avoid having to specify "-q" for all commands whose output is irrelevant, the run_script() method may be passed the keyword argument `null_output_matches_anything=True`. For example:

```
def test_ignoring_null_output(self):
    self.run_script("""
        $ bzr init
        $ bzr ci -m 'first revision' --unchanged
        $ bzr log --line
        1: ...
        """, null_output_matches_anything=True)
```

3.3 Import tariff tests

bzrlib.tests.test_import_tariff has some tests that measure how many Python modules are loaded to run some representative commands.

We want to avoid loading code unnecessarily, for reasons including:

- Python modules are interpreted when they're loaded, either to define classes or modules or perhaps to initialize some structures.
- With a cold cache we may incur blocking real disk IO for each module.
- Some modules depend on many others.
- Some optional modules such as *testtools* are meant to be soft dependencies and only needed for particular cases. If they're loaded in other cases then bzr may break for people who don't have those modules.

test_import_tariff allows us to check that removal of imports doesn't regress.

This is done by running the command in a subprocess with `PYTHON_VERBOSE=1`. Starting a whole Python interpreter is pretty slow, so we don't want exhaustive testing here, but just enough to guard against distinct fixed problems.

Assertions about precisely what is loaded tend to be brittle so we instead make assertions that particular things aren't loaded.

Unless selftest is run with `--no-plugins`, modules will be loaded in the usual way and checks made on what they cause to be loaded. This is probably worth checking into, because many bzr users have at least some plugins installed (and they're included in binary installers).

In theory, plugins might have a good reason to load almost anything: someone might write a plugin that opens a network connection or pops up a gui window every time you run 'bzr status'. However, it's more likely that the code to do these things is just being loaded accidentally. We might eventually need to have a way to make exceptions for particular plugins.

Some things to check:

- non-GUI commands shouldn't load GUI libraries
- operations on bzr native formats sholudn't load foreign branch libraries
- network code shouldn't be loaded for purely local operations
- particularly expensive Python built-in modules shouldn't be loaded unless there is a good reason

3.4 Testing locking behaviour

In order to test the locking behaviour of commands, it is possible to install a hook that is called when a write lock is: acquired, released or broken. (Read locks also exist, they cannot be discovered in this way.)

A hook can be installed by calling bzrlib.lock.Lock.hooks.install_named_hook. The three valid hooks are: *lock_acquired*, *lock_released* and *lock_broken*.

Example:

```
locks_acquired = []
locks_released = []

lock.Lock.hooks.install_named_hook('lock_acquired',
    locks_acquired.append, None)
lock.Lock.hooks.install_named_hook('lock_released',
    locks_released.append, None)
```

locks_acquired will now receive a LockResult instance for all locks acquired since the time the hook is installed.

The last part of the *lock_url* allows you to identify the type of object that is locked.

- BzrDir: */branch-lock*
- Working tree: */checkout/lock*
- Branch: */branch/lock*
- Repository: */repository/lock*

To test if a lock is a write lock on a working tree, one can do the following:

```
self.assertEndsWith(locks_acquired[0].lock_url, "/checkout/lock")
```

See bzrlib/tests/commands/test_revert.py for an example of how to use this for testing locks.

3.5 Skipping tests

In our enhancements to unittest we allow for some addition results beyond just success or failure.

If a test can't be run, it can say that it's skipped by raising a special exception. This is typically used in parameterized tests — for example if a transport doesn't support setting permissions, we'll skip the tests that relating to that.

```
try:
    return self.branch_format.initialize(repo.bzrdir)
except errors.UninitializableFormat:
    raise tests.TestSkipped('Uninitializable branch format')
```

Raising TestSkipped is a good idea when you want to make it clear that the test was not run, rather than just returning which makes it look as if it was run and passed.

Several different cases are distinguished:

TestSkipped Generic skip; the only type that was present up to bzr 0.18.

TestNotApplicable The test doesn't apply to the parameters with which it was run. This is typically used when the test is being applied to all implementations of an interface, but some aspects of the interface are optional and not present in particular concrete implementations. (Some tests that should raise this currently either silently return or raise TestSkipped.) Another option is to use more precise parameterization to avoid generating the test at all.

UnavailableFeature The test can't be run because a dependency (typically a Python library) is not available in the test environment. These are in general things that the person running the test could fix by installing the library. It's OK if some of these occur when an end user runs the tests or if we're specifically testing in a limited environment, but a full test should never see them.

See Test feature dependencies below.

KnownFailure The test exists but is known to fail, for example this might be appropriate to raise if you've committed a test for a bug but not the fix for it, or if something works on Unix but not on Windows.

Raising this allows you to distinguish these failures from the ones that are not expected to fail. If the test would fail because of something we don't expect or intend to fix, KnownFailure is not appropriate, and TestNotApplicable might be better.

KnownFailure should be used with care as we don't want a proliferation of quietly broken tests.

We plan to support three modes for running the test suite to control the interpretation of these results. Strict mode is for use in situations like merges to the mainline and releases where we want to make sure that everything that can be tested has been tested. Lax mode is for use by developers who want to temporarily tolerate some known failures. The default behaviour is obtained by `bzr selftest` with no options, and also (if possible) by running under another unittest harness.

result	strict	default	lax
TestSkipped	pass	pass	pass
TestNotApplicable	pass	pass	pass
UnavailableFeature	fail	pass	pass
KnownFailure	fail	pass	pass

3.6 Test feature dependencies

3.6.1 Writing tests that require a feature

Rather than manually checking the environment in each test, a test class can declare its dependence on some test features. The feature objects are checked only once for each run of the whole test suite.

(For historical reasons, as of May 2007 many cases that should depend on features currently raise TestSkipped.)

For example:

```
class TestStrace(TestCaseWithTransport):

    _test_needs_features = [StraceFeature]
```

This means all tests in this class need the feature. If the feature is not available the test will be skipped using UnavailableFeature.

Individual tests can also require a feature using the `requireFeature` method:

```
self.requireFeature(StraceFeature)
```

The old naming style for features is CamelCase, but because they're actually instances not classses they're now given instance-style names like `apport`.

Features already defined in `bzrlib.tests` and `bzrlib.tests.features` include:

- apport
- paramiko
- SymlinkFeature
- HardlinkFeature
- OsFifoFeature
- UnicodeFilenameFeature
- FTPServerFeature
- CaseInsensitiveFilesystemFeature.

- chown_feature: The test can rely on OS being POSIX and python supporting os.chown.
- posix_permissions_feature: The test can use POSIX-style user/group/other permission bits.

3.6.2 Defining a new feature that tests can require

New features for use with `_test_needs_features` or `requireFeature` are defined by subclassing `bzrlib.tests.Feature` and overriding the `_probe` and `feature_name` methods. For example:

```
class _SymlinkFeature(Feature):

    def _probe(self):
        return osutils.has_symlinks()

    def feature_name(self):
        return 'symlinks'

SymlinkFeature = _SymlinkFeature()
```

A helper for handling running tests based on whether a python module is available. This can handle 3rd-party dependencies (is `paramiko` available?) as well as stdlib (`termios`) or extension modules (`bzrlib._groupcompress_pyx`). You create a new feature instance with:

```
# in bzrlib/tests/features.py
apport = tests.ModuleAvailableFeature('apport')


# then in bzrlib/tests/test_apport.py
class TestApportReporting(TestCaseInTempDir):

    _test_needs_features = [features.apport]
```

3.7 Testing translations

Translations are disabled by default in tests. If you want to test that code is translated you can use the `ZzzTranslations` class from `test_i18n`:

```
self.overrideAttr(i18n, '_translations', ZzzTranslations())
```

And check the output strings look like `u"zz\xe5{{output}}"`.

To test the gettext setup and usage you override i18n.installed back to self.i18nInstalled and _translations to None, see test_i18n.TestInstall.

3.8 Testing deprecated code

When code is deprecated, it is still supported for some length of time, usually until the next major version. The `applyDeprecated` helper wraps calls to deprecated code to verify that it is correctly issuing the deprecation warning, and also prevents the warnings from being printed during test runs.

Typically patches that apply the `@deprecated_function` decorator should update the accompanying tests to use the `applyDeprecated` wrapper.

`applyDeprecated` is defined in `bzrlib.tests.TestCase`. See the API docs for more details.

3.9 Testing exceptions and errors

It's important to test handling of errors and exceptions. Because this code is often not hit in ad-hoc testing it can often have hidden bugs – it's particularly common to get NameError because the exception code references a variable that has since been renamed.

In general we want to test errors at two levels:

1. A test in `test_errors.py` checking that when the exception object is constructed with known parameters it produces an expected string form. This guards against mistakes in writing the format string, or in the `str` representations of its parameters. There should be one for each exception class.
2. Tests that when an api is called in a particular situation, it raises an error of the expected class. You should typically use `assertRaises`, which in the Bazaar test suite returns the exception object to allow you to examine its parameters.

In some cases blackbox tests will also want to check error reporting. But it can be difficult to provoke every error through the commandline interface, so those tests are only done as needed — eg in response to a particular bug or if the error is reported in an unusual way(?) Blackbox tests should mostly be testing how the command-line interface works, so should only test errors if there is something particular to the cli in how they're displayed or handled.

3.10 Testing warnings

The Python `warnings` module is used to indicate a non-fatal code problem. Code that's expected to raise a warning can be tested through callCatchWarnings.

The test suite can be run with `-Werror` to check no unexpected errors occur.

However, warnings should be used with discretion. It's not an appropriate way to give messages to the user, because the warning is normally shown only once per source line that causes the problem. You should also think about whether the warning is serious enought that it should be visible to users who may not be able to fix it.

3.11 Interface implementation testing and test scenarios

There are several cases in Bazaar of multiple implementations of a common conceptual interface. ("Conceptual" because it's not necessary for all the implementations to share a base class, though they often do.) Examples include transports and the working tree, branch and repository classes.

In these cases we want to make sure that every implementation correctly fulfils the interface requirements. For example, every Transport should support the `has()` and `get()` and `clone()` methods. We have a sub-suite of tests in `test_transport_implementations`. (Most per-implementation tests are in submodules of `bzrlib.tests`, but not the transport tests at the moment.)

These tests are repeated for each registered Transport, by generating a new TestCase instance for the cross product of test methods and transport implementations. As each test runs, it has `transport_class` and `transport_server` set to the class it should test. Most tests don't access these directly, but rather use `self.get_transport` which returns a transport of the appropriate type.

The goal is to run per-implementation only the tests that relate to that particular interface. Sometimes we discover a bug elsewhere that happens with only one particular transport. Once it's isolated, we can consider whether a test should be added for that particular implementation, or for all implementations of the interface.

See also Per-implementation tests (above).

3.12 Test scenarios and variations

Some utilities are provided for generating variations of tests. This can be used for per-implementation tests, or other cases where the same test code needs to run several times on different scenarios.

The general approach is to define a class that provides test methods, which depend on attributes of the test object being pre-set with the values to which the test should be applied. The test suite should then also provide a list of scenarios in which to run the tests.

A single *scenario* is defined by a *(name, parameter_dict)* tuple. The short string name is combined with the name of the test method to form the test instance name. The parameter dict is merged into the instance's attributes.

For example:

```
load_tests = load_tests_apply_scenarios

class TestCheckout(TestCase):

    scenarios = multiply_scenarios(
        VaryByRepositoryFormat(),
        VaryByTreeFormat(),
        )
```

The *load_tests* declaration or definition should be near the top of the file so its effect can be seen.

3.13 Test support

We have a rich collection of tools to support writing tests. Please use them in preference to ad-hoc solutions as they provide portability and performance benefits.

3.13.1 TestCase and its subclasses

The `bzrlib.tests` module defines many TestCase classes to help you write your tests.

TestCase A base TestCase that extends the Python standard library's TestCase in several ways. TestCase is build on `testtools.TestCase`, which gives it support for more assertion methods (e.g. `assertContainsRe`), `addCleanup`, and other features (see its API docs for details). It also has a `setUp` that makes sure that global state like registered hooks and loggers won't interfere with your test. All tests should use this base class (whether directly or via a subclass). Note that we are trying not to add more assertions at this point, and instead to build up a library of `bzrlib.tests.matchers`.

TestCaseWithMemoryTransport Extends TestCase and adds methods like `get_transport`, `make_branch` and `make_branch_builder`. The files created are stored in a MemoryTransport that is discarded at the end of the test. This class is good for tests that need to make branches or use transports, but that don't require storing things on disk. All tests that create bzrdirs should use this base class (either directly or via a subclass) as it ensures that the test won't accidentally operate on real branches in your filesystem.

TestCaseInTempDir Extends TestCaseWithMemoryTransport. For tests that really do need files to be stored on disk, e.g. because a subprocess uses a file, or for testing functionality that accesses the filesystem directly rather than via the Transport layer (such as dirstate).

TestCaseWithTransport Extends TestCaseInTempDir. Provides `get_url` and `get_readonly_url` facilities. Subclasses can control the transports used by setting `vfs_transport_factory`, `transport_server` and/or `transport_readonly_server`.

See the API docs for more details.

3.13.2 BranchBuilder

When writing a test for a feature, it is often necessary to set up a branch with a certain history. The `BranchBuilder` interface allows the creation of test branches in a quick and easy manner. Here's a sample session:

```
builder = self.make_branch_builder('relpath')
builder.build_commit()
builder.build_commit()
builder.build_commit()
branch = builder.get_branch()
```

`make_branch_builder` is a method of `TestCaseWithMemoryTransport`.

Note that many current tests create test branches by inheriting from `TestCaseWithTransport` and using the `make_branch_and_tree` helper to give them a `WorkingTree` that they can commit to. However, using the newer `make_branch_builder` helper is preferred, because it can build the changes in memory, rather than on disk. Tests that are explictly testing how we work with disk objects should, of course, use a real `WorkingTree`.

Please see bzrlib.branchbuilder for more details.

If you're going to examine the commit timestamps e.g. in a test for log output, you should set the timestamp on the tree, rather than using fuzzy matches in the test.

3.13.3 TreeBuilder

The `TreeBuilder` interface allows the construction of arbitrary trees with a declarative interface. A sample session might look like:

```
tree = self.make_branch_and_tree('path')
builder = TreeBuilder()
builder.start_tree(tree)
builder.build(['foo', "bar/", "bar/file"])
tree.commit('commit the tree')
builder.finish_tree()
```

Usually a test will create a tree using `make_branch_and_memory_tree` (a method of `TestCaseWithMemoryTransport`) or `make_branch_and_tree` (a method of `TestCaseWithTransport`).

Please see bzrlib.treebuilder for more details.

3.13.4 PreviewTree

PreviewTrees are based on TreeTransforms. This means they can represent virtually any state that a WorkingTree can have, including unversioned files. They can be used to test the output of anything that produces TreeTransforms, such as merge algorithms and revert. They can also be used to test anything that takes arbitrary Trees as its input.

```
# Get an empty tree to base the transform on.
b = self.make_branch('.')
empty_tree = b.repository.revision_tree(_mod_revision.NULL_REVISION)
tt = TransformPreview(empty_tree)
self.addCleanup(tt.finalize)
# Empty trees don't have a root, so add it first.
root = tt.new_directory('', ROOT_PARENT, 'tree-root')
# Set the contents of a file.
tt.new_file('new-file', root, 'contents', 'file-id')
preview = tt.get_preview_tree()
```

```
# Test the contents.
self.assertEqual('contents', preview.get_file_text('file-id'))
```

PreviewTrees can stack, with each tree falling back to the previous:

```
tt2 = TransformPreview(preview)
self.addCleanup(tt2.finalize)
tt2.new_file('new-file2', tt2.root, 'contents2', 'file-id2')
preview2 = tt2.get_preview_tree()
self.assertEqual('contents', preview2.get_file_text('file-id'))
self.assertEqual('contents2', preview2.get_file_text('file-id2'))
```

3.13.5 Temporarily changing state

If your test needs to temporarily mutate some global state, and you need it restored at the end, you can say for example:

```
self.overrideAttr(osutils, '_cached_user_encoding', 'latin-1')
```

This should be used with discretion; sometimes it's better to make the underlying code more testable so that you don't need to rely on monkey patching.

3.13.6 Observing calls to a function

Sometimes it's useful to observe how a function is called, typically when calling it has side effects but the side effects are not easy to observe from a test case. For instance the function may be expensive and we want to assert it is not called too many times, or it has effects on the machine that are safe to run during a test but not easy to measure. In these cases, you can use *recordCalls* which will monkey-patch in a wrapper that records when the function is called.

3.13.7 Temporarily changing environment variables

If yout test needs to temporarily change some environment variable value (which generally means you want it restored at the end), you can use:

```
self.overrideEnv('BZR_ENV_VAR', 'new_value')
```

If you want to remove a variable from the environment, you should use the special `None` value:

```
self.overrideEnv('PATH', None)
```

If you add a new feature which depends on a new environment variable, make sure it behaves properly when this variable is not defined (if applicable) and if you need to enforce a specific default value, check the `TestCase._cleanEnvironment` in `bzrlib.tests.__init__.py` which defines a proper set of values for all tests.

3.13.8 Cleaning up

Our base `TestCase` class provides an `addCleanup` method, which should be used instead of `tearDown`. All the cleanups are run when the test finishes, regardless of whether it passes or fails. If one cleanup fails, later cleanups are still run.

(The same facility is available outside of tests through `bzrlib.cleanup`.)

CHAPTER

FOUR

MANUAL TESTING

Generally we prefer automated testing but sometimes a manual test is the right thing, especially for performance tests that want to measure elapsed time rather than effort.

4.1 Simulating slow networks

To get realistically slow network performance for manually measuring performance, we can simulate 500ms latency (thus 1000ms round trips):

```
$ sudo tc qdisc add dev lo root netem delay 500ms
```

Normal system behaviour is restored with

```
$ sudo tc qdisc del dev lo root
```

A more precise version that only filters traffic to port 4155 is:

```
tc qdisc add dev lo root handle 1: prio
tc qdisc add dev lo parent 1:3 handle 30: netem delay 500ms
tc filter add dev lo protocol ip parent 1:0 prio 3 u32 match ip dport 4155 0xffff flowid 1:3
tc filter add dev lo protocol ip parent 1:0 prio 3 u32 match ip sport 4155 0xffff flowid 1:3
```

and to remove this:

```
tc filter del dev lo protocol ip parent 1: pref 3 u32
tc qdisc del dev lo root handle 1:
```

You can use similar code to add additional delay to a real network interface, perhaps only when talking to a particular server or pointing at a VM. For more information see <http://lartc.org/>.