Patterns for Automated As-Installed Tests of Packages in Large Software Ecosystems

Antonio Terceiro, Debian Project

Large software ecosystems, such as GNU/Linux distributions, demand a large amount of effort to make sure all its components work correctly invidually, and also integrate correctly with each other. Automated Quality Assurance techiniques can prevent issues from reaching end users. This paper presents a pattern language for automated software testing in production-like environments. Such environments are closer in similarity to the environment where software will be actually deployed and used, as opposed to the development environment under which developers, and regular Continuous Integration practices, usually test software products. The pattern language covers the handling of issues arising from the difference between development and production-like environments, as well as solutions for writing new, exclusive tests for as-installed functional tests.

Categories and Subject Descriptors: K.6.3 [**Computing Milieux**]: Management of computing and information systems—*Software Developement*

General Terms: Patterns for Automated Testing of Installed Packages

Additional Key Words and Phrases: Debian, Functional Tests, Debian Packages

ACM Reference Format:

Terceiro, A. 2016. Patterns for Automated As-Installed Tests of Packages in Large Software Ecosystems. jn 2, 3, Article 1 (November 2016), 10 pages.

1. MOTIVATION

Unit tests are a very useful tool in development projects, both to serve as insurance against future changes in the code breaking assumptions on interfaces, as well as, and perhaps more importantly, a design tool. Code that is hard to test is also hard to use, so test-driven development can help produce modules with lower coupling. Integration testing can then be used to make sure the nicely-decoupled modules actually work together (Beck 1999).

To help elucidate requirements and track project progress in terms of real business value, projects can also use acceptance tests (also called functional tests), which consist in testing the application in an end-to-end manner using a black box approach, to make sure that the needed functionalities are provided (A. Elssamadisy 2006). Acceptance tests try to exercise the application “from the outside”, without knowledge of implementation details. From a technical point of view this can be achieved by automating the interaction with a web application by driving a remote-controllable web browser for web applications, remote-controlling keyboard and mouse for desktop applications, or calling command-line applications directly; during these interactions, testing code provides predefined input and makes assertions on the produced outputs.

All of these types of testing are very important, but they are usually executed in a development environment, that is, a source code checkout, and with development tools installed. That environment is not representative of the environment where software packages will be installed for their usage in production: end user machines in

Author email: `terceiro@debian.org`.

11th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP’16), November 16-18, 2016, Buenos Aires, Argentina. © 2016 Antonio Terceiro. This work can be used under the terms of the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) See `https://creativecommons.org/licenses/by-sa/4.0/` for the full license text.

the case of end-user applications, or server machines in the case of server-side applications. This production environment is usually only exercised during manual Quality Assurance (QA) activities.

For this type of automated QA in production-like environments, existing automated tests can be reused, be they unit, integration or acceptance tests; specific production deployment tests can also be created.

This paper describes patterns for automated testing of software packages in production-like environments, as noticed in the context of the Debian Continuous Integration project. Throughout the paper, examples will be based on Debian's automated testing infrastructure, but they can be easily generalized.

2. BACKGROUND: AUTOMATED PACKAGE TESTING IN THE DEBIAN PROJECT

The Debian project[1] is an association of individuals who pursue the common goal of providing an completely free[2] Operating System. Debian's motto is "The Universal Operating System", so the Debian OS is suitable for use in a range of scenarios: workstations, servers, embedded devices, and others.

2.1 The Debian development process

Software in Debian if organized in terms of packages. Every end user program, system utility or library is provided as a package. Packages have dependencies between them, and each package provides metadata specifying its dependencies on other packages. For example an application package must depend on all of the libraries used by the application, so that when an user requests the installation of a given application the package management software can also automatically install all the necessary libraries.

At any point in time there are at least 3 versions, also known as distributions, of the Debian OS: `stable`, which only receives updates for important bug fixes and security issues, is the version recommended for usage by end users in private and corporate environments; `unstable`, which is the main entry point for software updates, receives direct updates from developers on a rate of hundreds or even thousands per day; and `testing`, which is the staging area for the next `stable`.

Non-maintenance software updates are introduced by publishing new versions of the corresponding packages first to the `unstable` distribution. Those packages will them migrate automatically from `unstable` to `testing` based on the following criteria:

—they have spent a given number of days in `unstable` without a critical bug[3] being reported against that new version[4].
—all of their dependencies are already available in `testing`, or are available in `unstable` and also satisfy the "N days without critical bugs" criteria.

Since updates to `testing` are guarded by a quarantine period in `unstable`, `testing` presents a good balance between recent software with the greatest new features and stability. This way, `testing` is a reasonably safe system for advanced users.

Users of `unstable`, on the other hand, contribute directly to the development of Debian, receiving all updates first hand, and having the ability to report issues to the attention of the package maintainers in order to prevent critical issues from reaching `testing`, therefore performing a Quality Assurance for the next release. As a consequence,

[1] `https://www.debian.org/`
[2] as in "free software" (also known as "open source"). The conceptual difference between "free software" and "open source" is outside of the scope of this paper.
[3] critical bugs include security-related bugs, bugs that cause the user to lose precious data, or that prevent the package from operating correctly at all.
[4] the exact number of days depends on the nature of the update. The standard is 5 days, but new package versions that contain security updates may require less time (2 days). The package maintainer can also choose a longer delay (10 days) if he/she wants to make sure more users will be able to test those changes and report any issues back if they arise, what might be useful when major changes in the software are introduced.

users of `unstable` are expected to be able to fix their systems by themselves when problems that can't be fixed by a sotfware update happen.

2.2 Automated testing

Several types of issues can be detected with automated testing, which scales better and lets humans focus their work on areas where automation can't (yet) help them. In this context, automated testing can help identify the following types of issues:

—If a library update brings incompatible behavior changes, existing applications that use such a library and that contain assumptions about the previous behavior might start to malfunction.
—In the same manner, changes in the packages forming the base system – which are implicit dependencies of all packages and therefore do not need to be explicitly listed in the dependency metadata – can invalidate assumptions made by code in programs.
—If a new version of program A starts to use functionality from another program B, but the package maintainer forgets to include that new dependency in the package metadata (because he/she already had B installed in their system during local testing), users installing A for the first time or upgrading from a previous version of A will have problems because B will not be installed by the package management software.
—Bugs caused by programming errors that only manifest themselves on special circumstances, such as specific days of the week, or months.

Each Debian package can include metadata specifying how to execute tests for that package, as defined in an specification known as DEP-8[5]. Those tests must assume they will be executed with the package installed on a test system (called a "test bed"), and must exercise the functionality in the package from the point of view of a user.

The main test runner implementation for DEP-8 tests is a tool called `autopkgtest`[6], which supports several types of test beds: Linux containers, virtual machines – both local ones, and on the cloud, and running the tests locally without any virtualization. `autopkgtest` will first install the corresponding packages into a test bed, and then run the tests. If the installation of the packages under test (or of any of their dependencies) fails, the test run is considered failed, which is consistent with the development policy since a installation failure is considered a critical bug. If the installation succeeds, then the actual test code will be executed, and its exit status will be used to determine the status of the test run.

Given that the packages are properly installed before running the tests, the tests can make several assumptions: any programs will be installed on the system-wide location for programs and can be invoked directly; system services such as servers will be automatically started and can be accessed by test scripts; libraries are available in the system-wide locations and can be directly used by test programs.

The Debian Continuous Integration project[7] is a service that executes those tests on the cloud whenever a new version of the package or of any the package in its dependency tree reaches the Debian package repository (or otherwise once a month even if no dependency has changed), providing a useful quality assurance tool during the development cycle of a new version of the Debian operating system.

2.3 A brief introduction to the DEP-8 specification

The DEP-8 specification defines how Debian packages can specify tests to be executed by an automated infrastructure. In this section we provide a brief description of that specification, just enough for the examples to make sense.

[5]http://dep.debian.net/deps/dep8/
[6]http://packages.debian.org/autopkgtest
[7]https://ci.debian.net/

A Debian source package can specify its tests by including a file called `debian/tests/control`, i.e. a file named `control` inside the directory `debian/tests`. This file should contain one or more RFC822-style paragraphs. An example:

```
Tests: test1, test2

Tests: test3
Depends: @, shunit2

Test-Command: wget http://localhost/package/
Depends: @, wget
```

It is important to note the semantics of the following fields:

—`Tests` contains a comma-separated list of test program names, assumed to be in the `debian/tests` directory. During the test run, each program is assumed to be a separate test, and is executed. If it exits with a status code of 0 (the status code of a successful command in the UNIX tradition), the corresponding test is assumed to have passed. Any other status code means a failure.
—`Test-Command` contains a direct command that is executed, and its status code is handled in the same way as `Tests`: if the command finishes with status 0 the test passed, otherwise it failed.
—`Depends` indicates which packages need to be installed in the testbed *before* the corresponding test is executed. The `@` symbol is handled in a special way, and means “all the binary packages produced from this source package”, and is the default when the `Depends:` field is ommited (for example in the first paragraph of the example above).

There are other supported fields, but they are details that can be ignored for the purposes of this paper. Interested readers may refer to the actual specificationfor more information.

3. PATTERNS

This pattern language helps package maintainers with providing automated tests for a package. Figure 1 displays all of the patterns, in a suggested order of application. The first and obvious choice should be to Reuse Existing Tests that already come with the package. From there, the solid black arrows point what to try next. Dashed gray arrows indicate a related pattern that is not necessarily the immediate next step, but that should probably be taken into account.

3.1 Reuse Existing Tests

Problem. A package does not have automated as-installed tests yet, but the original authors of the package provide tests intended to be executed against the source tree.

Solution. Implement as-installed tests as a simple program that calls the existing tests provided by the original authors of the package. These tests can be unit tests or acceptance tests.

Reusing *Unit tests* is very useful as they make sure that the modules in the code still work as compilers/interpreters, libraries and other system components get upgraded. Unit tests work even if the package has no user interaction interface, which is for example the case for libraries.

Reusing *Acceptance tests* is even more useful as they make sure that the user interaction interfaces continue to work as the infrastructure under the package changes.

However, since we want to Test The Installed Package, we also want to make sure those tests run against the *installed* files of the package, and not against the source tree. For interpreted languages, the test framework must be instructed to not load library code from the source directory, and assume that the libraries are already installed system-wide.

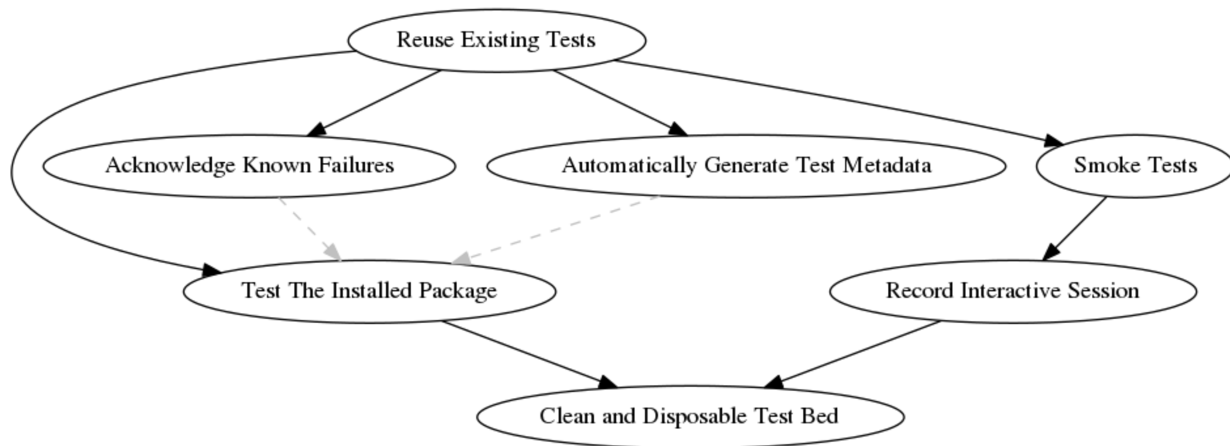

Fig. 1. Relationships between patterns

For compiled languages, compiled test programs must be linked against libraries installled system-wide; test scripts that use programs provided by the package should assume those programs are already installed system-wide, and not modify the system path to find those programs in the source tree. As a consequence of these requirements, it is not necessary to do a full compilation in the source tree. If the tests themselves need to be compiled first, ideally it should be possible to do that without compiling the entire project.

Examples. A large part of **Ruby and Perl packages in Debian** are provided by their original authors with automated tests, and the corresponding Debian package has the infrastructure to automatically run them both during package builds, and when testing the installed package.

In practical terms, a very large part of the Debian packages that have as-installed test suites are in fact reusing the tests provided with the package with some small customization to make them test the installed files instead of the source tree as explained above.

Related or Interacting Patterns. Existing test suites will often have to be modified to make sure they Test The Installed Package and not the source tree. If a given package type, say, libraries for a given programming language, has test suites that can be invoked in the same way, it is usually useful to Automatically Generate Test Metadata.

3.2 Test The Installed Package

Problem. Running tests against a source code tree does not effectively reproduces the conditions of production environments.

Solution. Run tests against the software package as it is installed on end user systems, by installing them from a repository of builds. The test programs themselves may be contained in the source code, but that is as far as the source code tree should be used during the tests.

Programs can be called directly since they should be installed at the system standard location for programs (e.g. `$PATH` on Unix-like systems).

Libraries can be imported into test programs without looking them up in the local source directory.

In most cases no build is necessary, since the tests will be exercising the code that is already installed in the test system. As an exception, if the tests themselves are written in a compiled language, at least they need to be built before running the tests.

Examples. **Conditionally setting $PATH.** To make a test suite work both during a package build (when most probably the package is *not* already installed), and against an installed package, path-related environment variables can be conditionally manipulated based on whether the tests are running under `autopkgtest` or not:

```
if [ -z "$ADTTMP" ]; then
  # if *not* running under adt-run, use programs from the source tree
  export PATH="$SOURCE_ROOT/bin:$PATH"
fi
```

The example above uses the fact that as per the DEP-8 specification, the `$ADTTMP` environment variable must be defined and contain the location of a temporary directory that the test code can use.

Making code from the source tree unavailable. Ruby packages have their library code under `lib/` in the source tree, and most packages rely on the testing framework setting the Ruby load path correctly so that when the test suite requests loading a given library file, Ruby will automatically select the version from the source tree. To avoid having that code loaded by the test suite when testing the installed package, the test runner program for Ruby packages, `gem2deb-test-runner`, will move that directory away so that Ruby will not find required libraries in the source tree, and will look them up in the system-wide locations, which is the desired behavior since we want to test the installed package.

Related or Interacting Patterns. When apoplying Reuse Existing Tests, the maintainer has to keep an eye out for test code that contains assumptions about being executed against a source tree, and is not actually testing the artifacts that are installed systemd-wide. For example, test code will often try to load libraries using relative paths, that is, they are explicitly loading code from the source tree. Good test code, on the other hand, will just load libraries as if they were installed system-wide, and trust that the testing framework will set up the environment correctly so that when running tests against the source tree, the libraries from the source tree will be made available to the test code.

3.3 Clean and Disposable Test Bed

Problem. Tests must reproduce the environment a user gets when the package being tested is installed on a clean system.

Solution. Utilize virtualization or container technology to provide fresh and clean systems for the tests to run on.

Package dependencies must be correctly specified so that when they are installed on a clean system, everything they need to work properly is also installed.

Any packages that are need to run the tests, but not to make normal use of the package, must be specified as *test dependencies*. This includes testing libraries and tools, debugging tools, etc.

During development of the tests themselves, it's useful to run them on a local system where all the dependencies are already installed, for a quick feedback loop. Running the tests against a clean system, however, is still a necessity before release.

Examples. **Supported virtualization backends.** `autopkgtest` itself already supports different "virtualization" backends, such as LXC (Linux containers), QEMU/KVM, and even spawning virtual machines on the cloud and connecting to them via SSH. It also supports a *"null"* virtualization backend, which will run the tests against the local system, assuming all dependencies are already installed, what is useful during test suite development.

Virtualization backends currently in use. At the time of writing, Debian[8] runs on LXC on the `amd64` and `arm64` hardware architectures, while Ubuntu[9] runs QEMU/KVM for the `amd64`, `i386` and `ppc64el` hardware architectures, and LXC on the `armhf` and `s390x` architectures.

[8] https://ci.debian.net/

[9] https://autopkgtest.ubuntu.com/

Related or Interacting Patterns. A Clean and Disposable Test Bed helps to Test The Installed Package since the package being tested will be installed in a clean system, making sure that its installation process is working correctly and also that its dependencies are correctly specified.

3.4 Acknowledge Known Failures

Problem. The majority of the tests pass succesfully, but not all of them. The failures may be due to the tests assuming they are testing the code in the source tree (instead of the installed code), or due to known platform incompatibilities.

Solution. Make the failure of the tests that are known to fail non-fatal, so that only a failure in a test that was known to pass previously causes an overal tests failure status.

The passing tests act as a regression test suite. It is expected that the known-failing tests fail, but if any of the other tests fail, that is probably a sign that there is something wrong.

The list of tests to ignore can be kept in a explicit blacklist file, which can then be used as a "TODO-list" for investigating why exactly they fail, and how to fix them or the corresponding code. From time to time, maintainers need to review the blacklist file to check if there aren't tests that have been fixed in the meantime and now "just work".

Known failures can mean problems with the tests themselves, and that is most probably the case if they pass when run against the source tree. But they can also be a sign of real issues.

Examples. The **Ruby interpreter packages in Debian** have a file called `known-failures.txt` which contains a list of tests known to fail when executed against a installed Ruby interpreter (as opposed to when they are executed against a source tree where a Ruby interpreter has just been built). The test runner code executes each test file, and any failing test file that is listed in `known-failures.txt` is added to the list of known (and expected) failures instead of the list of failures. At the end of a successfull test run, something like this is reported:

```
Finished
--------
   Tests executed: 744
             PASS: 714
             FAIL: 0
EXPECTED FAILURES: 30
```

Related or Interacting Patterns. Sometimes you Reuse Existing Tests but some of them will not work correcly when you want to Test The Installed Package, in which case you will often need to treat known failures as non-fatal.

3.5 Automatically Generate Test Metadata

Problem. Similar packages tend to have very similar test suite specifications.

Solution. Extract the hard-coded, duplicated test definitions from packages, and let them be automatically generated by a specialized tool at the beginning of the test execution process.

This way, several similar packages can have their tests executed in a similar way, without having to manually specify the tests for each one. And when developers come up with new ways of testing the group of packages, that can be introduced by modifying only the centralized tool, instead of modifying all packages.

Examples. **Ruby packages in Debian** have a standard way of declaring how their test suite has to be executed during build time. During the package build, `gem2deb-test-runner` is executed, detects how the test suite is supposed to be executed, and executes it. For it to support also running the same test suite against the installed package, it just has to receive a new `--autopkgtest` command line option to implement the Test The Installed

Package pattern. This makes it possible to not include an explicit – and duplicated – test declaration on every package; instead, the testing infrastructure assumes an implicit declaration that looks like this:

```
Test-Command: gem2deb-test-runner --autopkgtest --check-dependencies 2>&1
Depends: @, @builddeps@
```

Not having the test declaration hardcoded in every package provides also provides the benefit that if there is a need to make a change to that test declaration, it can be made centrally without having to manually modify hundreds of packages.

This is supported by the `autodep8`[10] tool, which **automatically generates test suite declarations** for known package types. At the time of writing, besides Ruby, `autodep8` also supports Perl, Python, DKMS, R, and NodeJS packages. `autopkgtest` automatically calls `autodep8` whenever it is told to run tests for a package that does not contain an explicit test suite declaration.

Related or Interacting Patterns. Automatically generating test metadata helps you Reuse Existing Tests in multiple packages that have a similar structure.

3.6 Smoke Tests

Problem. Package does not provide its own test suite, and yet its maintainer wants to make sure it works at least at a basic level.

Solution. Write test programs that exercise the basic functionality of the package and check the results against known good values. These test are called smoke tests because even though they do not exercise the entire feature set, their failure might indicate a serious issue ("where there is smoke, the is probably fire"); if very basic functionality does not work, that might also indicate more serious issues.

Even the simplest of the test cases are better than having no tests at all. For example, a failure when doing a very simple program invocation like `myprogram --version` might mean:

—A silent ABI changed in a shared library used by `myprogram`, and the dynamic linker fails to resolve all the symbols in the binary. This is a serious issue in an operating system with shared libraries.
—An issue with any of the needed libraries loaded by `myprogram` at startup, even if `myprogram` is written in an interpreted language.
—A compiler issue that causes `myprogram` to contain an invalid instruction for a given processor model.
—A failure in the packaging, that makes `myprogram` be installed to the wrong location.

Of course, having test that exercise more than that is extremely useful in the long-term maintainability of the package, in special against unexpected changes in its environment (dependencies, underlying operating system semantics, etc).

Examples. `chef` is a system provisioning tool, which supports automating system configuration, such as installing packages, defining users and their credentials, managing configuration files, among several other tasks. This is the main part of a smoke test contained in the `chef` Debian package, written as a shell script:

```
run chef-solo -c debian/tests/config.rb -j debian/tests/node.json

test_install_package() {
  assertTrue 'dpkg-query --show vim'
}

. shunit2
```

[10]https://packages.debian.org/autodep8

The `chef-solo` call invokes the chef local client with a well known configuration, provided together with the test itself, which should cause the `vim` package to be installed (the details of how exactly that happens are out of scope here). Then, in `test_install_package()`, we test that the installation actually happened, using the `shunit2` testing framework for shell scripts. For that to work properly, it means that chef itself is correctly installed and configured, and that the `chef-solo` had everything it needed in order to work properly in place. Extending this test program to test other aspects of the `chef` functionality would be a matter of extending the example configuration, and adding the corresponding tests for the desired effect.

Related or Interacting Patterns. Smoke tests are an alternative when you cannot Reuse Existing Tests. A good way to produce smoke tests is to Record Interactive Session.

3.7 Record Interactive Session

Problem. You want to provide tests for a package that does not have tests yet.

Solution. Install the package on a clean testbed, and based on the package documentation, issue commands and verify whether their output matches the expected, documented, behavior. Then copy-and-paste the commands and their output, and turn that into test cases.

Additionally, the exit status code of a program is an important part of its expected behavior, but it is not always represented in a console output. Make sure to take notes on the relevant exit codes and their expected values.

Turning the output of the interactive session contents into executable test cases may be harder or easier, depending on the test tool is use.

Examples. `clitest` is a tool that will read files formatted like an interactive console session, and replay them as a test case. For example, at the time of writing, the file `examples/cut.txt` inside the clitest source code contains the following:

```
$ echo "one:two:three:four:five:six" | cut -d : -f 1
one
$ echo "one:two:three:four:five:six" | cut -d : -f 4
four
$ echo "one:two:three:four:five:six" | cut -d : -f 1,4
one:four
$ echo "one:two:three:four:five:six" | cut -d : -f 4,1
one:four
$ echo "one:two:three:four:five:six" | cut -d : -f 1-4
one:two:three:four
$ echo "one:two:three:four:five:six" | cut -d : -f 4-
four:five:six
```

That file could well be – and probably was – just copy-and-pasted from a console session where the prompt is a $ sign. It can be directly fed into `clitest`, which will replay the commands, and check their output against the output previously recorded:

```
$ clitest examples/cut.txt
#1  echo "one:two:three:four:five:six" | cut -d : -f 1
#2  echo "one:two:three:four:five:six" | cut -d : -f 4
#3  echo "one:two:three:four:five:six" | cut -d : -f 1,4
#4  echo "one:two:three:four:five:six" | cut -d : -f 4,1
#5  echo "one:two:three:four:five:six" | cut -d : -f 1-4
#6  echo "one:two:three:four:five:six" | cut -d : -f 4-
```

```
OK: 6 of 6 tests passed
```

`clitest` makes it very easy to record interactive sessions as test cases. It also has options for checking exit statuses, approximate output checks with pattern matching, useful checks for programs that produce non-determinisc output, etc.

Related or Interacting Patterns. Smoke Tests can be written by applying this pattern. You will usually want to record an interactive session in a Clean and Disposable Test Bed, to make sure that you notice missing test dependencies as you go, and take notes about them.

4. KNOWN USES OF AUTOMATED AS-INSTALLED TESTS

The first heavy user of Automated As-Installed Tests is the Debian project, which is the context in which this paper was originated, and where the patterns were observed.

Ubuntu[11], another operating system which is based on Debian and shares a large part of Debian packages, also uses DEP-8 tests and `autopkgtest` in their own infrastructure[12], going one step beyond: in Ubuntu, updated packages are only promoted to the version under test that is normally available for end-users when they do not cause regressions in their test suite status, or in the test suite status of the packages that depend on them.

The OpenStack[13] project also makes use of as-installed tests. Tempest[14] is a OpenStack subproject that consists of a integration test suite that is designed to be run against an existing cloud platform setup, be it a test OpenStack installation in a single server or a full-blown cloud platform deployed onto dozens of servers in a datacenter. Tempest is also designed to only use the exposed APIs of an OpenStack deployment and to not interact directly with implementation details such as databases or virtualization hypervisors. The OpenStack project uses Tempest as a quality assurance during development by automatically deploying the latest version of all its components to a test infrastructure and then running Tempest against that. New OpenStack releases are also required to pass the Tempest test suite, as are e.g. the OpenStack packages provided in Debian.

5. CONCLUSION

This paper presented a set of patterns for handling test automation in large software collections, such as GNU/Linux distributions, based on the author's experience in the Debian project. The presented patterns help when planning and designing functional tests targetted at production-like environments, with the aim of reproducing as faithfully as possible the environment where software products will be deployed and executed.

The patterns cover the handling of issues arising from the difference between development and production-like environments, as well as solutions for writing new, exclusive tests for as-installed functional tests.

References

A. Elssamadisy, J. Whitmore. 2006. "Functional Testing: A Pattern to Follow and the Smells to Avoid." In *PLoP Pattern Languages of Programs*.

Beck, Kent. 1999. *Extreme Programming Explained: Embrace Change*. 1st ed. Cambridge, MA, USA: Addison-Wesley Professional.

[11]https://www.ubuntu.com/
[12]https://autopkgtest.ubuntu.com/
[13]https://www.openstack.org/
[14]https://github.com/openstack/tempest

11th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP'16), November 16-18, 2016, Buenos Aires, Argentina.
© 2016 Antonio Terceiro. This work can be used under the terms of the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) See https://creativecommons.org/licenses/by-sa/4.0/ for the full license text.