Patterns for Writing As-Installed Tests for Debian Packages

Antonio Terceiro, Debian Project

Large software ecosystems, such as GNU/Linux distributions, demand a large amount of effort to make sure all of its components work correctly invidually, and also integrate correctly with each other to form a coherent system. Automated Quality Assurance techniques can prevent issues from reaching end users. This paper presents a pattern language originated in the Debian project 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 mechanisms 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. Even though the patterns are documented here in the context of the Debian project, they can also be generalized to other contexts.

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. HILLSIDE Proc. of Latin American Conf. on Pattern Lang. of Prog. 11 (November 2016), 15 pages. jn 2, 3, Article 1 (November 2016), 15 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 may also be hard to reuse. Test-driven development can help produce modules with lower coupling, as well as modules with better interfaces. 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). They consist in testing the application in an end-to-end manner using a black box approach, to make sure that the needed functionality is provided (Elssamadisy and Whitmore 2006). Acceptance tests try to exercise the application “from the outside”, without knowledge of implementation details. From a technical point of view, acceptance tests can be achieved in a few different ways. Web applications can be tested by driving a remote-controllable web browser. Desktop applications can be tested by injecting

Author email: terceiro@debian.org.

synthetic keyboard and mouse input. Command-line applications can be invoked directly. During these interactions, testing code provides predefined input and makes assertions on the produced outputs.

All of these, and other 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 production environment: end user machines in 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.

Existing automated tests can be reused when performing automated QA in production-like environments. All of the existing unit, integration or acceptance tests can be useful in this context. Besides, 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 the concepts presented can be easily generalized.

The author has been involved in adding and reviewing tests for an expressive number of software packages in the Debian project. This experience led to the perception of recurring similarities among package test suites, and prompted the writing of this paper.

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 a completely free[2] Operating System. Debian's motto is "The Universal Operating System", and 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 – and in other software distrubutions – is organized in terms of packages. Every end user program, system utility or library is provided as a package. For example, in Debian the GNU Debugger is provided by the `gdb` package, and the Evince PDF reader is provided by the `evince` package.

The source code of a package is known as *source package*. Using development tools, a source package can be compiled to produce one or more *binary packages*. The binary packages are the artifacts that can be installed on the user systems. In Debian, and in all of the systems derived from it, the binary packages use the `.deb` format. Another well-known binary package format is the RPM (`.rpm`) format, used by RedHat Linux and its derivatives.

Packages can 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 time, there are 3 distributions of the Debian OS:

—`stable` only receives updates for important bug fixes and security issues. It is the official Debian release, and is the version recommended for usage by end users in private and corporate environments.

—`testing` is the next release in preparation.

[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.

—`unstable` is the main entry point for software updates. It receives direct updates from developers on a rate of hundreds or even thousands per day.

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, users of `unstable` are expected to be able to fix their systems by themselves when problems that can't be fixed by a software update happen. For example, an issue that arises on a package upgrade can be fixed by reverting to a previous version, or by applying a manual workaround. The idea is that issues that happen on upgrades for users of `unstable` are fixed before they reach `testing`. Since `testing` will be the next `stable`, it also prevents those issues from affecting users upgrading systems from one `stable` release to the next.

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 the Debian 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 automatically 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.

[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.

Each Debian source package can include metadata specifying how to execute tests for that package, as defined in an specification known as DEP-8[5]. Those tests can assume they will be executed with the corresponding binary packages installed on a test system (called a "test bed"), and must exercise the functionality in the package from the point of view of an user. Some tests are writen by the original authors of the package, and Debian-specific tests might be written by the corresponding Debian maintainer.

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 the test system, 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 of the packages in its dependency tree reaches the Debian package repository (or otherwise once a month even if no dependency has changed). It provides a useful quality assurance tool during the development cycle of a new version of the Debian operating system.

When the Debian CI platform was first introduced in January of 2014, less than 200 packages were being tested. As of this writing (January 2017), more than 6,500 packages are being tested. This represents an average growth rate of aproximately 175 packages a month, or almost 6 packages a day.

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.

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:

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

—`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 specification for 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.

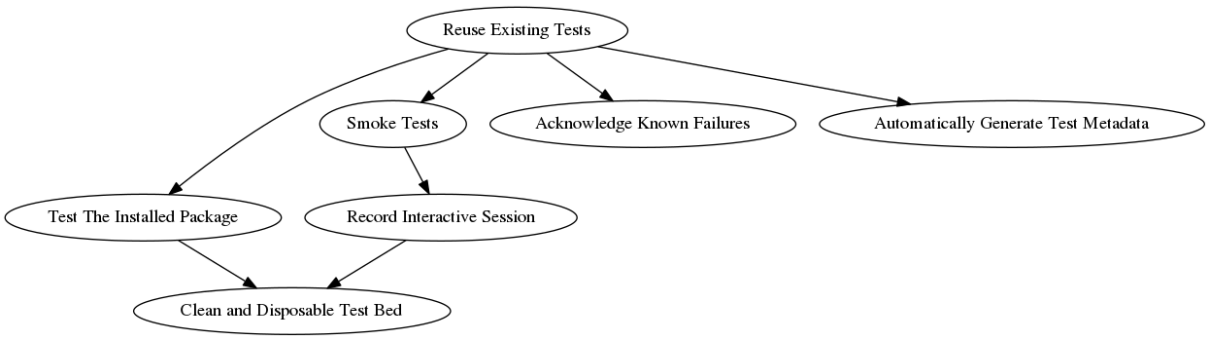

Fig. 1. Relationships between patterns

3.1 Reuse Existing Tests

The original authors of the package provide tests. Those tests are intended to be executed against the source tree, but still they are an useful assessment of whether the package works.

However, the package does not have as-installed tests.

The package maintainers might not have the time or the skills necessary to write the tests themselves. On the other hand, the original authors of the software already spent time and effort writing tests that work for them. Those tests might already be used for continuous integration on different platforms.

Therefore:

Implement as-installed tests as a simple wrapper program that calls the existing tests provided by the original authors of the package.

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.

A large part of *Ruby and Perl packages in Debian* are provided by their original authors with automated tests. The corresponding Debian packages will 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.

Let's examine the tests for the LXC package. LXC itself provides test programs, which in Debian are shipped in a `lxc-tests` package (i.e. they are not provided in the main `lxc` package. The main test script for LXC (`debian/tests/exercise` in the `lxc` source package) reads like this:

```
1  #!/bin/sh
2  # ...
3  set -eu
4  # ...
5  for testbin in /usr/bin/lxc-test-*; do
6      STRING="lxc-tests: $testbin"
7      [ ! -x "$testbin" ] && continue
8      # ...
9      OUT=$(mktemp)
10     $testbin >$OUT 2>&1 && pass "$STRING" || fail "$STRING" "$testbin" "$OUT"
11     rm $OUT
12 done
13
14 [ "$TEST_FAIL" != "0" ] && exit 1
15
16 exit 0
```

The test script will call each of the test programs (line 5), and fail if any of them fails (line 14). In the above excerpt, the `fail` function is omitted, but one of its features is keeping a count of failed tests in the `$TEST_FAIL` global variable.

3.2 Test The Installed Package

A package is being tested by reusing existing tests provided by its original authors, or tests prepared by its maintainers. The goals of the DEP-8 specification is to test software in its installed form. The advantage of doing that is to more closely reproduce an environment that is similar to what end user will have on their systems.

Tests that exercise the source code tree do not effectively reproduce the conditions of production environments. And by using code from the source tree, such tests might pass even if the corresponding package is not installed at all!

Some test suites will explicitly use absolute locations to invoke programs or load library code from the source tree. This can usually be detected, for example, by the usage of keyworks that expand to the location of current file, such as `__FILE__` in Ruby, or `__file__` in Python.

Some test suites will rely on the testing framework in use to properly setup the runtime environment. They won't explitly reference code from the source tree. When running during a package build, the testing framework will make sure that programs and library from the source tree is available.

It might not be always clear which is which. Sometimes package maintainers will have to carefully stufy the test code.

Therefore:

Eliminate usage of programs and library code from the source tree in favor of their counterparts that have been installed system-wide.

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 the case of packages written in compiled languages, no build is necessary, since the tests will be exercising the code that is already installed in the test system. If the tests themselves are not distributed with the package, they might need to be compiled. It should be possible to build only the tests without building all the rest of the code.

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, you **conditionally manipulate path-related environment variables**, based on whether the tests are running under `autopkgtest` or not:

```
if [ -z "$ADTTMP" ]; then
  # if *not* running under autopkgtest, use programs from the source tree
  export PATH="$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 contains the location of a temporary directory that the test code can use.

When test code explicitly loads library code from the source tree, they need to be changed to **remove explicit references to the source tree**. An example in Ruby:

```
require File.expand_path(__FILE__, '../../lib/library')
```

Assuming the testing framework sets up the environment correctly, this can be replaced with something like the following:

```
require 'library'
```

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. When the test suite requests loading a given library file, Ruby will automatically select the version from the source tree. `gem2deb-test-runner`, the utility that runs tests in Ruby

packages, does a trick to avoid loading code from the source tree when testing installed packages. It will move the `lib/` directory away, **making code from the source tree unavailable**. The Ruby interpreter will not find required libraries in the source tree, and fallback to looking them up in the system-wide locations.

`pkg-perl-autopkgtest`, the test runner helper for Perl packages, does something similar. It will **copy the test code to a temporary directory**, and run the tests from there. The Perl interpreter will have no reference to the original source tree, and we can be sure it won't be able to load code from it.

3.3 Clean and Disposable Test Bed

When preparing automated tests, it is important to make sure that the tests are reproducible. This includes not depending on any specificity of the system where they were first developer, as well as being able to reproduce a test run on any other system. Tests that execute consistently on automated infrastructures and are also easy to reproduce locally by developers are of fundamental importance for effective QA work.

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

In this context, reproducibility clearly implies automation. That automation may incur in variable amounts of effort. Package maintainers must weigh whether the upfront automation effort is worth, even though in the long run it usually is.

Therefore:

Use 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 needed 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.

If the package is not completely ready to be used after a simple installation, the next steps must be automated in the test code. For example, a web application that might need properly configured web server before it can be actually used. Package maintainers can use this automation as a proof-of-concept of the documentation that explains uses how to perform this extra configuration.

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.

`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. This is is useful during test suite development.

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/`

3.4 Acknowledge Known Failures

A package has a extensive test suite.

The majority of the tests pass succesfully, but not all of them.

These test failures can have several causes. Some tests migh assume they are testing the code in the source tree (instead of the installed code). There can can be failures due to known platform incompatibilities. There can also be random failures caused by fragile tests.

Ideally, we want 100% of the tests to pass. Maintainers should usually investigate each failure, and provide patches to fix the issues, be them in in the test code itself, or in the production code.

Whether of not each individual failure if severe enough to stop the package from being consumed by users is a maintainer decision. Not all features and corner cases are equally important, and it is sometimes acceptable to live with issues that will affect a small fraction of the users if the effort to have it fixed is too high.

One one hand, the effort required to investigate the causes of the failures will depend on the size of the test suite, on the number of failing tests, and on the complexity of each of the present issues. On the other hand, maintainers will often have limited time to handle the package. They may feel inclined to disable the test suite altogether.

Therefore:

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 will 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”.

It is important, however, to keep in mind that the ideal solution it to eventually get rid if all test failures. Maintainers can plan to spend time investigating the issues one by one, until eventually they are all resolved. Tolerating known failures is a way of being able to fix these issues gradually, while still benefiting from the known-good tests along the way.

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, when they all pass).

Let’s look at an excerpt of the test runner code:

```
KNOW_FAILURES=$(dirname $(readlink -f $0))/known-failures.txt
# ...
for t in $tests; do
  if ruby2.3 test/runner.rb $t >log 2>&1; then
    echo "PASS $t"
    pass=$(($pass + 1))
  else
    if grep "^$t$" $KNOW_FAILURES; then
      fail_expected=$(($fail_expected + 1))
```

```
10        echo "FAIL (EXPECTED) $t"
11        # ...
12      else
13        fail=$(($fail + 1))
14        echo "FAIL $t"
15        # ...
16      fi
17      # ...
18    fi
19    total=$(($total + 1))
20  done
21  # ...
22  if [ $fail -gt 0 ]; then
23    exit 1
24  fi
```

Every test that fails is checked against the list of known failures (line 8), and counted as an expected failure or otherwise (lines 9 and 13). At the end, the test runner will report an non-success exit code only if there are failures that were not expected (lines 22–24). In all cases, the test runner produces a report like the following:

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

3.5 Automatically Generate Test Metadata

Teams of maintainers will often have several similar packages. For example, teams that maintain packages written in the same programming language (e.g. the Ruby team, the Perl team). A large number of those packages will be able to be tested with very similar code, specially if the upstream community has well-known standards on how test suites are supposed to be executed.

Similar packages tend to have similar or identical test suite specifications.

Having packages with duplicated test definitions makes it difficult to update all of them when some new testing requirement comes along. On the other hand, maintainers will usually not be able to use the exact same definition in all packages; some of them will need small variations.

Therefore:

Replace duplicated test definitions with ones generated automatically at run time.

Having a central tool automatically generate test definitions at runtime allows similar packages to have their tests executed in the same way, without the need of carrying duplicated test definitions on each package. When developers come up with new ways of testing that group of packages, that can be introduced by modifying only that tool, instead of having to modify all packages.

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 the `--autopkgtest` command line option. This makes it possible not to 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@
```

This is supported by the `autodep8`[10] tool, which is used by `autopkgtest` to **automatically generates test suite declarations at run time** 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. Support for new types of packages can be easily added.

3.6 Smoke Tests

Despite automated testing being increasingly common, there are still packages that won't have them. Other packages may have only unit tests for their library code, but no tests for user-facing features like command-line interfaces, or background services.

Some features might not even be provided by the original package, but by the packaging. Examples include services, scheduled tasks, maintainer scripts, etc.

The package maintainer wants to add tests to make sure that high-level functionality works.

Writing tests for the internals of a program will usually require a knowledge of the codebase, and might require more effort than what the maintainer has available. Moreover, if this type of test could be written that should probably be done upstream directly.

Having tests that are specific to the packaging might be justifiable, as long as they do not require a large amount of effort to maintain.

Therefore:

Write smoke tests that exercise the basic functionality of the package and check for expected results.

A smoke test is a test that covers the main functionality of a system with the objective of quickly determining if the system is working correctly. Even though a smoke test usually does not exercise the entire feature set, its failure might indicate a serious issue (think "when there is smoke, there is probably fire"). If very basic functionality does not work, that might also indicate more serious issues.

Even the simplest of the test cases is 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[11] 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.

[10] `https://packages.debian.org/autodep8`
[11] Application Binary Interface; interface between program modules at the machine code level. Includes function calling conventions, parameter sizes, etc. When a library changes its ABI, compiled programs need to be *recompiled* to work correctly against that new version of the library. Thus ABI changes in libraries need to be coordinated with programs that use them.

—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 integration tests that exercise more than that is extremely useful in the long-term maintainability of the package. Such more elaborated tests can assist in detecting a wider range of unexpected changes in its environment (dependencies, underlying operating system semantics, etc).

`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
```

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.

3.7 Record Interactive Session

Some packages have been around for a long time, since a time when automated testing was not so common place as it is now. Even today automated testing is not always 100% guaranteed. Also, when developing something experimental, it is not always easy to design automated tests because one might not know exactly what is being developed. And experimental software will often be shipped, sometimes to the despair of their developers.

You want to provide tests for a package that provides none.

Some programs will have clear boundaries with its environment. Candidate boundaries are – in order of difficulty to test – command-line interfaces, listening server sockets, and graphical user interfaces.

Therefore:

Record sample interactions with the program in a way that they can be "played back" later as automated tests.

Install the package on a clean testbed. Exercise it user interface, and verify that the results match the expected ones. Issue commands and verify whether their output matches the expected behavior. Then copy-and-paste the commands and their output into a test case.

The expected behavior will hopefully be explicitly documented. If that is not the case you may have to rely on your experience or on the experience of others about how the program in question is expected to behave.

Additionally, the exit status code of a program is an important part of its expected behavior, but it is usually not represented in console output. Make sure to take note 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 in use. This pattern is *one way* of producing Smoke Tests.

`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.

GUI programs can be tested in similar way by using tool that can simulate user interactions. Depending on the technique used, and on whether the testing tool has access to the GUI internals, such tests may be more or less fragile.

4. KNOWN USES OF AUTOMATED AS-INSTALLED TESTS

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

Ubuntu[12], 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[13], 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[14] project also makes use of as-installed tests. Tempest[15] 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 it. New OpenStack releases are also required to pass the Tempest test suite, as are e.g. the OpenStack packages provided in Debian.

5. RELATED PATTERNS

Automate First (J. Yoder, Wirfs-Brock, and Washizaki 2016) describes aspects related to test automation in agile development processes, and contains precious insights on what and when to automate processes, including testing.

Many, if not all, of the XUnit Test Patterns (Meszaros 2006) and patterns for Test-Driven Development (E. Guerra et al. 2013) can be applied in the context of testing packages. The Test Smells listed by Meszaros (Meszaros 2006) are also a factor to keep in mind on all testing activities.

There are also sets of patterns for testing specific types of systems, such as Web Applications (M. Aniche, Guerra, and Gerosa 2014) and distributed systems (E. Guerra et al. 2014).

6. CONCLUSION

This paper presented a pattern language 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.

Acknowledgements

This paper has improved significantly since its first draft due to contributions by several people. Ayla Rebouças shepherded the paper leading up to SugarloafPLoP'16 and provided valuable feedback. Pedro Martos, Renato Cordeiro Ferreira, Alessandro Leite, Tiago Boldt Sousa, Joe Yoder, and Ralph Johnson provided comprehensive feedback during the SugarloafPLoP'16 writers' workshop.

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

References

Aniche, M., E. Guerra, and M. Gerosa. 2014. “A Set of Patterns to Improve Code Quality on Automated Functional Tests of Web Applications.” *21th Conference on Pattern Languages of Programs (PLoP)*.

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

Elssamadisy, Amr, and Jean Whitmore. 2006. “Functional Testing: A Pattern to Follow and the Smells to Avoid.” In *PLoP Pattern Languages of Programs*, 27–21. PLoP ’06. New York, NY, USA: ACM. doi:10.1145/1415472.1415504[16].

Guerra, Eduardo, Paulo Bittencourt Moura, Felipe Meneses Besson, Ayla Rebouças, and Fabio Kon. 2014. “Patterns for Testing Distributed Systems Interaction.” In *Proceedings of the 21st Conference on Pattern Languages of Programs*, 12–11. PLoP ’14. USA: The Hillside Group. `http://dl.acm.org/citation.cfm?id=2893559.2893571`.

Guerra, Eduardo, Joseph Yoder, Maurício Finavaro Aniche, and Marco Aurélio Gerosa. 2013. “Test-Driven Development Step Patterns for Designing Objects Dependencies.” *Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP)*.

Meszaros, Gerard. 2006. *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Yoder, J.W., R. Wirfs-Brock, and H. Washizaki. 2016. “QA to Aq Part Six: Being Agile at Quality.” In *HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 23*.

[16] `https://doi.org/10.1145/1415472.1415504`