Patterns for Testing Debian Packages

Antonio Terceiro
terceiro@debian.org
A brief intro to Debian CI

- autopkgtest created back in 2006 (!)
- 2014: Debian CI launches
- Goal: provide automated testing for the Debian archive (i.e. run autopkgtest for everything)
- Plans: gate migrations from unstable to testing
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Duration</th>
<th>Status</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7</td>
<td>2017-08-04 15:16:30 UTC</td>
<td>0h 5m 45s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-04 12:01:59 UTC</td>
<td>0h 6m 14s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-04 08:09:16 UTC</td>
<td>0h 6m 10s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-04 02:28:44 UTC</td>
<td>0h 5m 54s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-03 22:10:34 UTC</td>
<td>0h 6m 14s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-03 11:19:56 UTC</td>
<td>0h 6m 8s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-03 03:13:20 UTC</td>
<td>0h 6m 17s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-02 00:31:22 UTC</td>
<td>0h 6m 42s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-01 20:02:27 UTC</td>
<td>0h 6m 27s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-01 17:09:19 UTC</td>
<td>0h 5m 48s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-01 05:39:36 UTC</td>
<td>0h 6m 24s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-08-01 00:51:37 UTC</td>
<td>0h 6m 16s</td>
<td>pass</td>
<td>debci log</td>
</tr>
<tr>
<td>1.7</td>
<td>2017-07-31 01:01:48 UTC</td>
<td>0h 6m 9s</td>
<td>pass</td>
<td>debci log</td>
</tr>
</tbody>
</table>
~8k source packages
~28% of the archive

~21 packages/day
since January 2014
As a CI proponent, I have read and written tests for several packages. I started to notice, and suggest, similar solutions to recurring problems ... and thought they could/should be documented.
Patterns
A **pattern** is a re-usable, documented solution to a recurring problem.

Often used in design disciplines, such as architecture and software engineering.
Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch
SugarLoaf PLoP 2016
Tango Edition

11TH LATIN AMERICAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS
Buenos Aires, Argentina
This talk is based on the following paper:


PDF: [https://deb.li/pattestdeb](https://deb.li/pattestdeb)
Documenting patterns

- Common elements:
  - Title
  - Context
  - Problem
  - Forces
  - Solution
  - Consequences
  - Examples
- Several different styles/templates
A note about Patterns conferences

- A breath of fresh air for those used to traditional academic conferences
- Discussion instead of presentation
- Dedicated reading time → people actually read your stuff
A brief introduction to DEP8
DEP8

Goal: test a package in a context as close as possible from a system where the given package is properly installed
$ cat debian/tests/control
Tests: test1, test2

Tests: test3
Depends: @, shunit2

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

$ grep Testsuite: debian/tests/control
Testsuite: autopkgtest
# added for you by dpkg-source from stretch+
# if debian/tests/tests/control exists
Tooling: autopkgtest

```
$ autopkgtest foo_1.2.3-1.dsc -- null
$ autopkgtest foo_1.2.3-1_amd64.changes -- null
$ autopkgtest -B . -- null

$ autopkgtest ... -- lxc --sudo autopkgtest-sid-amd64
$ autopkgtest ... -- qemu /path/to/img
```
Pattern #1
Reuse Existing Tests
Upstream provides tests. They are intended to run against the source tree, but still they are useful to verify whether the package works (*context*)

However, **there are no "as-installed" tests** (*problem*)
• maintainer might lack time or skills to write tests ...
• but upstream already wrote some tests

(forces)
Therefore:

Implement as-installed tests as a simple wrapper program that calls the existing tests provided by upstream (solution)
Reusing **unit tests** is very useful for library packages

Reusing **acceptance tests** is useful for applications
#!/bin/sh
#
#
set -eu
#
for testbin in /usr/bin/lxc-test-*; do
    STRING="lxc-tests: $testbin"
    [ ! -x "$testbin" ] && continue
    # ...
    OUT=$(mktemp)
    $testbin >$OUT 2>&1 && pass "$STRING" \|| fail "$STRING" "$testbin" "$OUT"
    rm $OUT
done

[ "$TEST_FAIL" != "0" ] && exit 1

exit 0
Pattern #2
Test the Installed Package
The goals of DEP-8/autopkgtest is to test the package as installed.

Tests that exercise the source tree do not effectively reproduce users' systems.
• Some test suites will rely on absolute file paths (bad)
  • `__FILE__` in Ruby
  • `__file__` in Python
• Some test suites will rely on the testing framework in use to setup the environment
Therefore:

Remove usage of programs and library code from the source tree in favor of their installed counterparts.
• Programs can be called directly by name (they are in $PATH)
• Libraries can be imported/linked against without any extra effort (they are in the standard places)
• No build is necessary (maybe only the test themselves)
if [ -z "$ADTTMP" ]; then
    # if *not* running under autopkgtest,  
    # use programs and libraries from the
    # source tree,
    export PATH="$SOURCE_ROOT/bin:$PATH"
    export LD_LIBRARY_PATH="$SOURCE_ROOT/lib"
fi
# THIS IS AN ABERRATION
require File.expand_path(__FILE__, '../../lib/library')

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

require 'library'
Pattern #3
Clean and disposable test bed
We want reproducible tests, so everything the test needs to work must be explicit.

Tests must reproduce the environment a user gets when installing the package on a clean system.
• Reproducibility comes from automation
• Automation has an upfront cost (usually worth it in the long run)
Therefore:

Use virtualization or container technology to provide fresh test systems
• Package dependencies must be correct
• Packages needed for the test but not for normal usage must be specified in the control file
• Further automation can be scripted in test scripts (e.g. web server setup)
• While writing the tests themselves it is useful to run them against a "dirty" system; but you should test on a clean one before uploading
Examples

• autopkgtest supports different virtualization options, including none (*null*)
• Debian CI uses LXC. QEMU will be used in the future
• Ubuntu autopkgtest uses QEMU and LXC
Pattern #4
Acknowledge Known Failures
A package has an extensive test suite

The majority of tests pass successfully, but some fail
• a test may fail for several reasons
• of course, ideally we want 100% of the tests passing
• Failures needs to be investigated
• how severe is each failure?
  • are all features and corner cases equally important?
• how much effort is required to fix broken tests?
Therefore:

Make known failures non-fatal
• Passing tests act as regression test suite
• list of non-fatal failures can be used as a TODO list
• one should probably not postpone fixing the underlying issues forever
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))
      echo "FAIL (EXPECTED) $t"
      # ...
    else
      fail=$((($fail + 1))
      echo "FAIL $t"
      # ...
    fi
    # ...
  fi
  # ...
fi
total=$((($total + 1))
done
# ...
if [ $fail -gt 0 ]; then
  exit 1
fi
Pattern #5
Automatically Generate Test Metadata
Teams have large amounts of similar packages which could be tested with similar code
Upstream communities usually have conventions on how to run tests

Similar packages tend to have similar or identical test control files
• duplicated test definitions are bad
• Some packages will need slight variations
Therefore:

*Replace duplicated test definitions with ones generated automatically at runtime.*
• automatically generated definitions can be updated centrally
• handling test environments is also managed centrally
  • e.g. making sure the tests are running against the installed package

we do this with autodep8(1)
# package: ruby-foo

$ grep ^Testsuite debian/control
Testsuite: autopkgtest-pkg-ruby

$ autodep8
Test-Command: gem2deb-test-runner \ 
   --autopkgtest \ 
   --check-dependencies 2>&1
Depends: @, «build-dependencies», \ 
gem2deb-test-runner

Also supported:
Perl, Python, NodeJS, DKMS, R, ELPA, Go
Pattern #6
Smoke Tests
• Not all packages provide tests
• Sometimes features are provided by the packaging and not by upstream (e.g. maintainer scripts, service definitions)
The package maintainer wants to add tests to make sure that high-level functionality works.
• Testing internals may be hard (and should be done upstream)
• Packaging-specific tests might be justifiable
Therefore:

Write smoke tests that exercise functionality of the package and check for expected results.
A *smoke test* covers the main and/or most basic functionality of a system.

smoke \(\rightarrow\) fire
Even the simplest test case (e.g. `myprogram --version`) could catch:

- Silent ABI changes
- Issues in dependencies
- Invalid instructions
- Packaging issues
  (myprogram: command not found)
chef-solo -c debian/tests/config.rb -j debian/tests/node.json

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

. shunit2
Pattern #7
Record Interactive Session
• Some packages predate the pervasiveness of automated testing
• Sometimes writing automated tests upfront is not so easy (e.g. experimental interfaces)
You want to provide tests for a package that provides none.
some programs will have a clear boundary with its environment, e.g. 
CLIs
GUIs
listening server sockets
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 the interface, and verify results match expected/documented behavior
• record that interaction in an executable format (YMMV)
$ cat examples/cut.txt
$ 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
$ 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
Final remarks
• These patterns document solutions for autopkgtest-related design issues
• hopefully they are useful for you
• Some patterns solve the same problem
• Can you identify other patterns?
plug: ci/autopkggtest BoF
Friday 15:30 — "Bo" room
Learn more

Paper PDF
https://deb.li/pattestdeb

Debian CI documentation
https://ci.debian.net/doc/

Tutorial: Functional testing of Debian packages (DC15 talk; transcription at Debian CI docs)