

Re: Code correctness, and testing strategies

Source: <http://coding.derkeiler.com/Archive/Python/comp.lang.python/2008-06/msg01226.html>

- *From:* David <wizzardx@xxxxxxxxx>
 - *Date:* Wed, 11 Jun 2008 10:39:49 +0200
-

Thanks again for an informative reply :-)

I finished that small app I mentioned last time (before reading the last reply to this thread). A few points (apologies for the length):

I added a few integration tests, to test features which unit tests weren't appropriate for. The main thing they do is call my main object's 'run_once()' method (which iterates once) instead of 'run()' (which runs forever, calling run_once() endlessly), and then check that a few expected things took place during the iteration.

Those integration tests helped me to find a few issues. There were a few 'integration' tests which were actually acceptance/functional tests because integration tests weren't good enough (eg: run script in unix daemon mode and check for pid/lock/etc files). I left those under integration because it's annoying to change to 3 different directories to run all of the tests. I should probably split those off and automate the testing a bit more.

After doing all the automated tests, I did some manual testing (added 'raise UNTESTED' lines, etc), and found a few more things. eg, that one of my (important) functions was being unit tested but not ever being run by the main app. Fixed it with BDD, but it is something I need to be aware of when building bottom-up with BDD :-). Even better: develop top down also, and have acceptance tests from the start, like you mentioned before.

Another thing I found, was that after installing on a test system there were more failures (as expected). Missing dependencies which had to be installed. Also Python import errors because the files get installed to and run from different directories than on my workstation. Not sure how I would use BDD to catch those.

Finally – I didn't use BDD for the debianization (checking that all the shell scripts & control files work correctly, that the debian package had all the files in the correct places etc). I figured it would be too much trouble to mock the Debian package management system, and all of the linux utilities.

Re: Code correctness, and testing strategies

Should a BDD process also check 'production' (as opposed to 'under development') artifacts? (installation/removal/not being run under a version control checkout directory/missing dependencies/etc)?

In the future I'll probably create 'production' acceptance tests (and supporting utilities) following BDD before debianization. Something automated like this:

- 1) Build the installer package from source
- 2) Inspect the package and check that files are in the expected places
- 3) Package should pass various lint tests
- 4) Install the package under a chroot (reset to clear out old deps, etc)
- 5) Check that files etc are in the expected locations
- 6) Run functional tests (installed with the package) under the chroot
- 7) Test package upgrades/removal/purges/etc under the chroot.

There are a few existing Debian utilities that can help with this.

On Wed, Jun 11, 2008 at 4:36 AM, Ben Finney
<bignose+hates-spam@xxxxxxxxxxxxxxxx> wrote:

David <wizzardx@xxxxxxxx> writes:

[...]

Does this mean that you leave out the formal 'integration' and 'systems' testing steps? By actually running the app you are doing those things more or less.

I'm not sure why you think one would leave out those steps. The integration and systems tests should be automated, and part of some test suite that is run automatically on some trigger (such as every commit to the integration branch of the version control system).

It sounded that way because when I asked about integration tests originally, you said to use approval testing. Which seems to completely skip the automated 'integration' and 'systems' testing steps I was expecting.

A Python distutils 'setup.py' is a very common way to set up the build

Re: Code correctness, and testing strategies

parameters for an application.

I don't make setup.py, because my work projects are always installed via apt-get onto Debian servers. setup.py is a bit redundant and less functional for me :-)

Might be a bad practice on my part. Usually you start with a non-Debian-specific install method. Which then gets run to install the files into a directory which gets packaged into a Debian installer. I cheat with Python apps because I'm both the upstream author and the Debian maintainer. I setup my Debian control files so they will copy the .py files directly into the packaged directory, without running any non-Debian-specific install logic.

I should probably have a setup.py anyway... :-)

But I will be making a Debian installer a bit later.

That's an important part of the build process, but you should write (and test via your automated build process) a 'setup.py' before doing that.

Should I have a setup.py if it won't ever be used in production?

I've considered setting up a centralised build server at work, but currently I'm the only dev which actually builds & packages software, so it wouldn't be very useful.

It's extremely useful to ensure that the automated application build infrastructure is in place early, so that it is easy to set up and automate. This ensures that it actually gets done, rather than put in the "too hard basket".

Thanks for the advice. I'll look into setting it up :-)

At the moment I have ad-hoc projects in my home directory, each with it's own git repo. There's at least 100 of them. Usually I run a local script to build deb files from one of those dirs, and push them to our internal debian repo. I'll setup a system where public git repos (with hook scripts) get setup on the build server. Then I'll push my changes to the server when I want a package to be built & uploaded. I also

Re: Code correctness, and testing strategies

want the same system on my workstation for testing, and in case there are network problems.

The benefits are many even for a single-developer project: you have confidence that the application is simple to deploy somewhere other than your development workstation, you get early notification when this is not the case, you are encouraged from the start to keep external dependencies low, your overall project design benefits because you are thinking of the needs of a deployer of your application, and so on.

I already think this way because I manage our installation infrastructure (I have a developer hat, a package maintainer hat, and a release manager hat). But it might be useful for other devs in the future, if I can get them to use the system ;-). Also so they can make new packages etc while I'm on leave without having to use my workstation :-). Hasn't happened yet. Usually they would just SCP updated PHP files into production without using Debian installers. Their work is important, but it doesn't perform any critical function :-). (as in bugs = lost mega bucks per minute).

We do have other devs (PHP mostly), but they don't even use version control :-/.

Fix that problem first. Seriously.

I've tried a few times, but haven't succeeded yet. It's not an easy concept to sell to people who aren't interested, and there is more work involved than simply hacking away at source and making occasional copies to a new 'bpk' or 'old' sub-directory.

There have been a few cases where the devs were working on code, when clients came in and needed to see an earlier version (also not in production yet). I've saved their bacon by restoring the correct version from our daily backups. We didn't have much in the way of backups either before I set them up :-)

David.

.