Verification, validation and testing

Perdita Stevens

School of Informatics
University of Edinburgh

Verification, validation and testing

These are the main techiques for eliminating all kinds of bugs in software (including design bugs).

Verification: are we building the software right?

Validation: are we building the right software?

Testing is a useful technique for both.

“Bug”: or more precisely:

From IEEE610.12-90 (IEEE Standard Glossary of Software Engineering Terminology):

- Error: A difference between a computed result and the correct result
- Fault: An incorrect step, process, or data definition in a computer program
- Failure: The [incorrect] result of a fault
- Mistake: A human action that produces an incorrect result

The common term “defect” usually means fault.

Testing

Ways of testing: black box (specification-based) and white box (structural).

Different testing purposes:

- Module (or unit) testing
- Integration testing
- System testing
- Acceptance testing
- Stress testing
- Performance testing

and many more. i.e., large area: whole third-year course on testing. Basics only here.

Why test?

Testing has three main purposes:

- to help you find the bugs
- to convince the customer that there are no/few bugs
- to help with system evolution.

Crucial attitude: *A successful test is one that finds a bug.*

How to test

Tests often have a contractual role. For this and other reasons, they must be:

- repeatable
- documented (both the tests and the results)
- precise
- done on configuration controlled software

Ideally, test spec should be written at the same time as the requirements spec: this helps to ensure that the requirements are highly testable. It may seem backwards to consider testability when writing requirements but it's now standard.

E.g. may write requirements in numbered sentences and keep a tally of which test(s) tests which requirement(s). Use cases may help structure tests.

Test-first development

Basic idea is to write tests before the code that it's testing.

When you discover a bug, you ask yourself why none of your tests revealed the bug. Is there a bug in an existing test? Or do you need another test?

1. Fix or create a test to catch the bug.
2. Check that the test fails.
3. Fix the bug
4. Run the test that should catch this bug: check it passes
5. Rerun *all* the tests, in case your fix broke something else.

Advantages of test-first development

If you had a completely explicit, fully detailed specification, there wouldn't be nearly so much point in TFD. However, you almost never do. In the real world:

1. Trying to write a test often reveals that you don't completely understand exactly what the code *should* do. It's best to find this our early.
2. If you code first, and requirements are not completely specified, it's tempting to settle ambiguities based on what's easiest to code. A user doesn't know or care what's easy to code, so this can lead to user-hostile software.
3. Occasionally, you write a test and find that it already passes, e.g. because you didn't understand how much your colleague had already done!
4. If tests are written after the code, then under time pressure you may end up never writing them. That way lies madness.

Test-driven development

A subtly different term, covers the way that in Extreme Programming detailed tests *replace* a written specification.

JUnit

Recall that JUnit is an open-source framework for Java testing.

Now require to do enough investigation of JUnit to be confident writing basic JUnit tests, as well as understanding ideas of test-first development. Lots of possible sources, e.g.:

- `http://www.junit.org`
- *Using JUnit with Eclipse IDE* `http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html` (good introduction, details may not be quite right for the version we have)
- *Writing and running JUnit tests* from the Eclipse help documentation, Java Development User Guide, Getting Started, Basic Tutorial.

Best way: try it for real on something.

Assertions

Assertions allow the Java programmer to do 'sanity checks' during execution of the program.

Suppose `i` is a integer variable, and we are writing a bit of code where we 'know' that `i` must be even (because of what we did earlier). We can write

```
assert i % 2 == 0;
```

to check this – if `i` is odd, an `AssertionError` exception is raised.

Assertion checking can be switched off. Therefore, **never** do anything with side-effects in an assertion.

Preconditions, postconditions, invariants

Particularly common types of assertion about methods and classes are:

Precondition: a condition that must be true when a method (or segment of code) is invoked.

Postcondition: a condition that the method guarantees to be true when it finishes.

Invariant: a condition that should always be true of objects of the given class.
What does always mean? In all *client-visible* states: that is, whenever the object is not executing one of its methods.

Writing these conditions would be tedious – and we might want to write richer conditions than can be expressed in Java.

Java Modeling Language

The Java Modeling Language is a way of annotating Java programs with assertions. It uses Javadoc-style special comments.

Preconditions: `//@ requires x > 0;`

Postconditions: `//@ ensures \result % 2 == 0;`

Invariants: `//@ invariant name.length <= 8;`

General assertions: `//@ assert i + j = 12;`

JML allows many extensions to Java expressions. For example, quantifiers `\forall` and `\exists`. And much, much more.

Tools and further reading

The tools jmlc, jmlrac etc. compile and run JML-annotated Java code into bytecode with runtime assertion checking. (Quantifiers?)

They got out of date wrt Java, but development now seems to be active again.

Required/Recommended Reading: Leavens and Cheon 'Design by Contract with JML', via `http://www.cs.iastate.edu/~leavens/JML/documentation.shtml`. Section 1 is Required, the rest is Recommended. You should be able to read and write simple examples, like those in Section 1.

ESC/Java2 – static analysis for JML

Runtime checking is all very well – but you might never hit the problem cases.

Static analysis is analysis of source code. Can hope to *prove* that assertions are always satisfied, not just check the cases that work. (But often better to think of it as compile-time debugging!)

ESC/Java2 is a tool for static analysis of JML-annotated programs. It uses theorem-proving techniques to establish assertions. However, it is neither sound nor complete (completeness is theoretically impossible). It also works with Java 1.4.

There are other more rigorous theorem proving environments for JML which are sound.

Reading

Required: GSWEBOK Ch11, on Software Quality (could delay till after discussion of process)

Required: some JUnit information, see above.

Required/Suggested: Design by Contract with JML (Section 1 required), see above

Suggested: GSWEBOK Ch5, on Software Testing

Suggested: Sommerville Ch22-24 and/or Stevens Ch19.

Quotes of the day

Beware of bugs in the above code; I have only proved it correct, not tried it.

(Donald Knuth)

Law 1. Every non-trivial program contains at least one bug.
Law 2. Every non-trivial program can be simplified by at least one line of code.

We deduce: Every non trivial program can be simplified to one line of code, and it will contain a bug.

(Anon)