Verification, validation and testing

Paul Jackson

School of Informatics
University of Edinburgh

Verification, validation and testing

"VV&T" generally refers to all techniques for improving product quality, e.g., by eliminating bugs (including design bugs).

Verification: are we building the software right?

Validation: are we building the right software?

Testing is a useful (but not the only) technique for both.

Other techniques useful for verification:

- static analysis
- reviews/inspections/walkthroughs

Other techniques useful for validation:

- prototyping/early release

“Bug”: or more precisely:

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

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

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 premature 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.

Evolving tests when they don't catch new bugs

Assume an implementation passes all current tests.

What if a new bug is identified by customer or by code review?

A good discipline is:

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.

Test-first development

The motivating observation: tests implicity define

- interface, and
- specification of behaviour

for the functionality being developed.

Basic idea is

- write tests **before** the code that they testing.
- run tests as code is written,

As a consequence:

- bugs found at earliest possible point
- bug location is relatively easy

Further advantages of test-first development

TFD

1. **ensures adequate time for test writing**: If coding first, testing time might be squeezed or eliminated. That way lies madness.
2. **clarifies requirements**: trying to write a test often reveals that you don't completely understand exactly what the code *should* do.
3. **avoids poor ambiguity resolution**: if coding first, ambiguities might be resolved based on what's easiest to code. This can lead to user-hostile software.

Test-driven development

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

Test automation and JUnit

Automation of tests is essential, particularly when tests must be re-run frequently.

JUnit is a framework for automated testing of Java programs. It's use is required in Coursework 3.

Lots of sources of help. 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.
- *JUnit Tutorial* `//http://www.vogella.de/articles/JUnit/article.html`

Limitations of testing

- **Writing tests is time-consuming**
- **Coverage almost always limited**: may happen not to exercise a bug.
- **Difficult/impossible to emulate live environment perfectly**
  - e.g. *race conditions* that appear under real load conditions can be hard to find by testing.
- **Can only test executable things**, mainly code, or certain kinds of model – not high level design.

Reviews/walkthroughs/inspections

One complementary approach is to get a group of people to look for problems.

This can:

- find bugs that are hard to find by testing;
- also work on non-executable things, e.g. requirements specification
- tease out issues where the problem is that what's "correct" is misunderstood
- spot unmaintainable code.

Of course the author(s) of each artefact should be looking for such problems – but it can help to have outside views too.

For our purposes reviews/walkthroughs/inspections are all the same; there are different versions with different rules. "Review" for short.

Reviews, key points

A review is a meeting of a few people,

which reviews one specific artefact (e.g., design document, or defined body of code)

for which specific entry criteria have been passed (e.g., the code compiles).

Participants study the artefact before the meeting.

Someone, usually the main author of the artefact, presents it and answers questions.

The meeting does not try to fix problems, just identify them.

The meeting has a fixed time limit.

Static and dynamic analysis

Neither testing nor reviews are reliable ways to find subtle technical problems in code. Formal tool-supported analysis not limited to specific test cases can help. Roughly

- Dynamic analysis involves running the code (or at least, simulating it)
- Static analysis does not: the tool inspects the code without running it.

Both require clear specification of what is being checked. This can be given generically ("this code does not deadlock", "no null-pointer exception will ever be raised") or in terms of annotations of the code ("for every class, the class invariant written in the code will always be true when a public method is invoked"). Let's look at such annotations.

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 (and should, for release) 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.

Dynamic analysis

Running the program with assertion checking turned on is a basic kind of dynamic analysis. But more is possible.

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

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

Static analysis

NB even type-checking during compilation is a kind of static analysis!

Static analysis has advanced a lot in recent years, and is much more practical for relatively "clean" languages like Java than for e.g. C++ (pointer arithmetic makes everything harder!).

Tools vary in what problems they address, e.g.

- common coding oversights such as failing to check return values for error codes
- correctness of pre/post-condition specification of methods
- concurrency bugs e.g. race conditions

Think of what these tools do as findings bugs, just like testing. Their methods are not usually sound (every problem flagged guaranteed to be a real error) or complete (guaranteed to find every error) – undecidability looms.

Static analysis tools for Java

OpenJML project provides support for static analysis of JML-annotated programs.

Treats programs as mathematical objects and automatically proves assertions.

Project also supports dynamic analysis tools.

FindBugs is relatively widely used: looks for *bug patterns*, code idioms that are often errors

ThreadSafe from the Informatics spinout *Contemplate* focusses on finding concurrency bugs

Reading

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

Required: some JUnit information, see slide above. You must know how to create and run a JUnit 4 test for a method of a class - the best way to learn this is to do it.

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

Suggested: GSWEBOK Ch5, on Software Testing

Suggested: Stevens Ch19.

Suggested: Sommerville Ch 8 (9th and 10th Ed), Chs 22-24 (8th Ed)