Verification, validation and testing

Nigel Goddard

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):

- 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. 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 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.
- *JUnit Tutorial*//`http://www.vogella.de/articles/JUnit/article.html`

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.

Wake-up question

1. Verification is the process of checking the requirements match what the customer wants
2. Validation is the process of checking that the software does what the customer wants
3. Verification is the process of checking that the software does what the customer wants
4. Validation is the process of checking that the software matches the requirements
5. Verification is the process of checking that the software matches the requirements

Which of the above statements are true?

A 1 and 4
B 1 and 2
C 2 and 5
D 3 and 4

Answer:

Wake-up question

1. Verification is the process of checking the requirements match what the customer wants
2. Validation is the process of checking that the software does what the customer wants
3. Verification is the process of checking that the software does what the customer wants
4. Validation is the process of checking that the software matches the requirements
5. Verification is the process of checking that the software matches the requirements

Which of the above statements are true?

A 1 and 4
B 1 and 2
C 2 and 5
D 3 and 4

Answer: C: valid - right soft.; verif. - soft. right

Limitations of testing

- Time-consuming to write tests, so
- Coverage almost always limited: may happen not to exercise a bug.
- Difficult/impossible to emulate live environment perfectly, so 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.

NB easier to get value out of tests if they are written early, so that they also help clarify and document requirements, than if they are written after the code.

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 `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.

Review question

Reasonable references from pre and postconditions are:

1. precondition refers to the result of the method and to the arguments; postcondition refers to the arguments
2. pre and post condition both refer to the arguments
3. pre refers to the arguments, post to the arguments and the result

Which of the above statements are true?

A 1 and 2

B 2 and 3

C 1 and 3

D 1, 2 and 3

Answer:

Review question

Reasonable references from pre and postconditions are:

1. precondition refers to the result of the method and to the arguments; postcondition refers to the arguments
2. pre and post condition both refer to the arguments
3. pre refers to the arguments, post to the arguments and the result

Which of the above statements are true?

A 1 and 2

B 2 and 3

C 1 and 3

D 1, 2 and 3

Answer: B

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

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.)

FindBugs is relatively widely used: looks for lots of common errors.

Contemplate Limited is a spinout company from Informatics which provides these kinds of tools.

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: Sommerville Ch22-24 and/or Stevens Ch19.