Inf2: SEPP
Lecture 18: Verification, validation and testing: Overview

Cristina Adriana Alexandru

School of Informatics
University of Edinburgh

Last lectures

- Requirements engineering
- Design
- Construction
- Refactoring

This lecture

Verification, validation and testing ("VV&T")

- Motivation
- Definitions
- Essence of testing
- Terminology of what can go wrong
- Approaches to testing, kinds of tests
- How to test:
  - Test-first development
  - Test-driven development
  - Behaviour-driven development
- Evolving tests
- Limitations of testing

Verification, validation and testing: motivation

From Lecture 14 . . .

High quality code does *what it is supposed to do*.

What it is supposed to do means:

- Meets stated requirements
- Meets wider expectations
  (of whoever it was who asked for its development, and ideally of stakeholders)

Problems:

- How can we know this is the case?
- When it is not, how can we isolate the cause?

Verification, validation and testing: definitions

"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?

- Does software meet requirements?

Validation: are we building the right software?

- More general. Does software meet expectations?

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

Other techniques useful for verification:
reviews/inspections/walkthroughs, static analysis

Other techniques useful for validation: prototyping, early releases

Essence of testing

- Generating stimulus for component
- Collecting outputs from component
- Checking if actual outputs are as expected

Often hard to fully test a component in isolation

- Component test environment constructed using *mock objects*

Terminology of what can go wrong

1. Mistake: Human behaviour that produces fault(s)
2. Fault: An incorrect step, process, or data definition in a computer program. A.k.a defect or, informally, a bug
3. Error: A discrepancy between some computed value and the correct value; Captured by tests.
4. Failure: The termination of an intended behavior due to a fault manifestation.

Faults do not necessarily lead to errors

Errors do not necessarily lead to failures

Some approaches to testing

- **Black box**
  - Focusing on the requirements while treating the system as a black box (i.e. not looking into its code)
  - Advantages: helps conduct verification; when refactoring, tests do not need to be changed
  - Disadvantages: may not thoroughly exercise the different ways to execute the code
- **White box**
  - Considers software code; testing that the system does what the developer intended
  - Advantages: helps developers check their work, more through
  - Disadvantages: will miss misinterpreted requirements, refactoring will require updating the tests.
- **Regression testing**: repeat some/all tests after modifications; can help identify bugs and their location quicker.

Kinds of tests

- **Module (or unit) tests**: for each class in OO software, with subset of tests for each of its methods; Isolate causes of errors.
- **Integration tests**: test that components interact properly
- **System tests**: at the level of the whole system, check if requirements met
- **Acceptance tests**: check that system meets user/customer needs (validation); done in real environment with real data
- **Stress tests**: push system to its limits to check that performance degrades gracefully
- **Performance tests**: checking other performance requirements
- **Regression tests** (see regression testing above)

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

How to test

Desirable that tests are:

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

Ideally, tests should be written at the same time as the requirements- Now standard practice

- Tests and requirement features can be cross-referenced
- Use cases can suggest tests

Helps to ensure testability of requirements.

Test-first development (TFD)

Basic idea is

- write tests as informed by and capturing requirements, and **before** writing the code they apply to (**but still having requirements!**)
- write code to pass the tests
- iteratively run tests as code is written

The motivating observation: tests implicity define

- interface, and
- specification of behaviour

for the functionality being developed.

As a consequence:

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

Further advantages of TFD

TFD

1. **clarifies requirements**: trying to write a test often reveals that you don't completely understand exactly what the code *should* do.
   - Discover issues more quickly than if coding first
   - Makes coding easier
2. **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.
3. **ensures adequate time for test writing**: If coding first, testing time might be squeezed or eliminated, which is very risky.

Test-driven development (TDD)

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

- Disadvantage: communication with stakeholders affected

Behaviour-driven development (BDD)

More recent term

- Writing use cases in a more stylised language which can be parsed by a machine and at least partially turned into tests
- Advantages: more interpretable by stakeholders, produce tests
- Disadvantages: still not ideal for stakeholder communication; go deeper into design and implementation and may lose sight of higher level needs.

Evolving tests when new bug is identified

Assume an implementation passes all current tests.

What if a new bug is identified by users 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.

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 or requirements.

Reading

Essential: SWEBOK v4 Ch 5, on Software Testing

Essential: Sommerville SE Ch 8

Suggested: Stevens Ch 19.