Introduction to Software Testing

Błażej Pietrzak
blazej.pietrzak@cs.put.poznan.pl

Agenda

- Verification vs Validation
- Software Testing
- Testing axioms
- High level and Low level tests
- Black box vs White box testing
- Testing and debugging
- Inspections and testing

Notorious bugs

- Shooting Down Airbus 320, 1988
  - 290 people dead
  - cryptic and misleading output displayed by the tracking software of USS Vincennes
- Death of cancer patients, 1985-87
  - Due to overdoses of radiation resulting from a race condition between concurrent tasks in Therac-25 software

Notorious bugs

- The Pentium Floating-Point Bug, 1994
  - Intel replaced all defective chips at a cost of ~$475M
- Ariane 5 Rocket, 1996
  - Satellite launcher malfunction was caused by a faulty software exception routine resulting from a bad 64-bit floating point to 16-bit integer conversion
  - The rocket and it's four satellites were uninsured
  - Development ~$7 000 000, losses ~$500 000 000

Verification vs Validation

- Verification
  „Are we building the product right"
  - The software should conform to its specification
- Validation
  „Are we building the right product"
  - The software should do what the user really requires

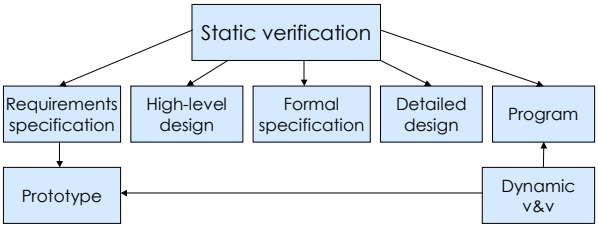
The V & V process

- V & V must be applied at each stage in the software process
- Principal objectives:
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation

Static and dynamic verification

- Software inspections (static)
  - Concerned with analysis of the static system representation to discover problems
- Software testing (dynamic)
  - The system is executed with test data and its operational behavior is observed

Static and dynamic V & V



V & V goals

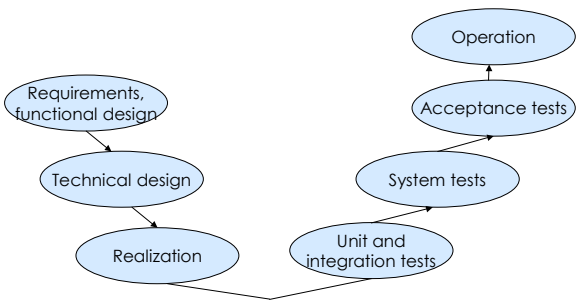
- Verification and validation should establish confidence that the software is fit for purpose
- This does not mean completely free of defects
- It must be good enough for its intended use and the type of use will determine the degree of confidence that is needed

V & V confidence

Depends on system's purpose, user expectations and marketing environment

- Software function
  - The level of confidence depends on how critical the software is to an organization
- User expectations
  - Users may have low expectations of certain kinds of software
- Marketing environment
  - Getting a product to market early may be more important than finding defects in the program

V – model



V & V planning

- Plan early!
- Static verification or testing?
- Defining standards for the testing rather than describing product tests
- Pareto rule (80/20)

The structure of a software test plan

- The testing process
- Requirements traceability
- Tested items
- Testing schedule
- Test recording procedures
- Hardware and software requirements
- Constraints

What is software testing?

Software testing is **running the code** for combinations of states and input data **in order to detect faults**.

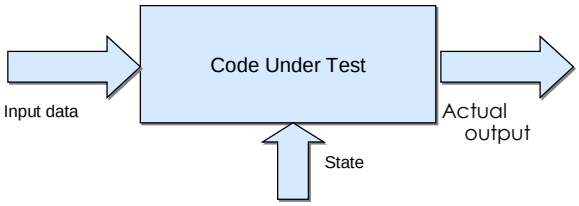
Robert V. Binder: „Testing Object-Oriented Systems. Models, Patterns, and Tools"

Successful test = fault detection

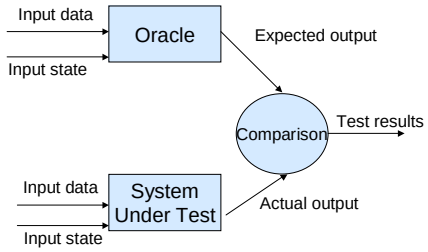
Test effectiveness = ability to find faults

What is software testing?

A test case



What is software testing?



Code coverage

How much code is exercised by tests

Statement coverage

- Every statement is exercised

Branch coverage

- Every branch has been taken in every possible direction (true and false)

Statement coverage – an example

How can tests achieve full statement coverage?

```
void func(int count) {
  if ((count % 2) == 0)
    System.out.println("count is even.");
  for (; count < 5; count++)
    System.out.println("count " + count);
}
```

Statement coverage – an example

How can tests achieve full statement coverage?

```
void func(int count) {
  if ((count % 2) == 0)
    System.out.println("count is even.");
  for (; count < 5; count++)
    System.out.println("count " + count);
}
```

count = 4

Branch coverage

How can tests achieve full branch coverage?

```
void func(int count) {
  if ((count % 2) == 0)
    System.out.println("count is even.");
  for (; count < 5; count++)
    System.out.println("count " + count);
}
```


Branch coverage

How can tests achieve full branch coverage?

```
void func(int count) {
  if ((count % 2) == 0)
    System.out.println("count is even.");
  for (; count < 5; count++)
    System.out.println("count " + count);
}
```

Two tests: count =4, count=13

Branch coverage – cont.

- The *if* statement would need count both even and odd
- *for* loop must evaluate at least once and false at least once
- Two tests: count =4, count=13

Dark side of code coverage

```
long multiply(int arg1, int arg2) {
    return arg1 + arg2;
}
```

arg1 = 0 and arg2 = 0 produce correct result, coverage 100%.

Sometimes 100% coverage could not be achieved (i.e. dead code)

Dark side of code coverage

```
long multiply(int arg1, int arg2) {
    return arg1 + arg2;
    System.out.println(„Dead code");
}
```

Sometimes 100% coverage could not be achieved (i.e. dead code)

Mutation testing

- A scheme for testing tests, by gauging their effectiveness of a given test.
- Assume that we have a test T which the code passes.
- If the mutated code also passes test T, then T is less-likely to be regarded as a good test.

Mutated code

```
long add(int arg1, int arg2) {
    return arg1 * arg2;
}
```

Test cases:
arg1 = 0 and arg2 = 0
arg1 = 1 and arg2 = 1

Mutated code

```
long add(int arg1, int arg2) {
    return arg1 * arg2;
}
```

Test cases:
arg1 = 0 and arg2 = 0
should be removed or adjusted
arg1 = 1 and arg2 = 1
produce incorrect result and stays

Testing limits

„Testing can show the presence of errors, but never their absense"
Dijkstra

Testing limits – cont.

- The only validation technique for non-functional requirements
- Static verification + testing = full V &V coverage

Testing limits – cont.

Exhaustive testing is impossible (intractable)

```
for (int i = 0; i < n; i++) {
    if (a.get(i) == b.get(i))
        x[i] = x[i] + 100;
    else
        x[i] = x[i] / 2;
}
```

n	Path No.
1	3
2	5
10	1025
60	1 152 921 504 606 847 200

Testing limits – cont.

„Mój wyrób spełni założenia, jeśli spełni je każda z jego części składowych i jeśli zmontuję je właściwie"

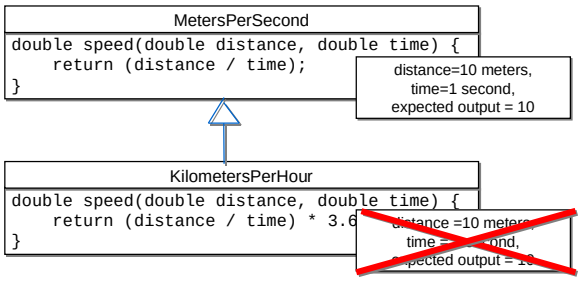
Nie można założyć, że z poszczególnych poprawnych części zawsze powstaje poprawna całość

Weyuker

Antiextensionality axiom – cont.

- A test suite that covers one implementation of a specification does not necessarily cover a different implementation of the same specification (many ways of implementation)
  - Quicksort vs Heapsort
- A test suite that covers base class's method may not be sufficient if the the method is derived in the subclass

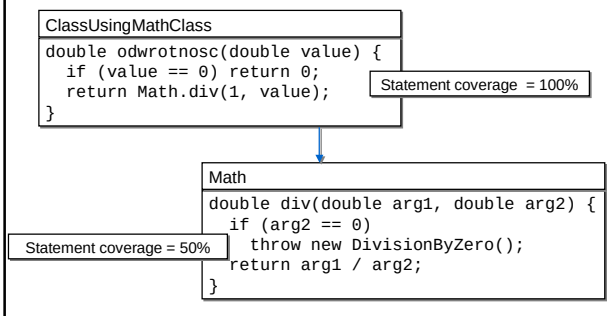
Antiextensionality axiom



Antidecomposition axiom – cont.

The coverage achieved for a module under test is not necessarily achieved for modules that it calls.

Antidecomposition axiom



Anticomposition axiom

- **Test suites that are individually adequate for segments within a module are not necessarily adequate for the module as a whole.**
  - **Individual method scope test suites does not guarantee that all class interactions will be tested.**
  - **We cannot be confident in a subclass that has passed only subclass scope testing, even if both superclasses and subclasses methods have already passed their own suites.**

Low-level tests

Unit test

- test, executed by the developer in a labolatory environment, that should demonstrate that the program (method, class, or cluster of classes) meets the requirements set in the design specifications.

Integration test

- test, executed by the developer in a labolatory environment, that should demontstrate that a logical series of programs meets the requirements set in the design specifications.

High-level tests

System test

- test, executed by the developer or independent test team in (properly controllable) labolatory environment, that should demonstrate that the developed system or subsystems meet the requirements set in the functional and quality specifications.

Acceptance test

- test, executed by the user(s) and system manager(s) in an environment simulating the operational environment to the greatest possible extent, that should demonstrate that the developed system meets the functional and quality requirements.

Regression testing

Regression testing

- When changes are made, re-test previous test cases at ascertain that no new errors were introduced in the changes

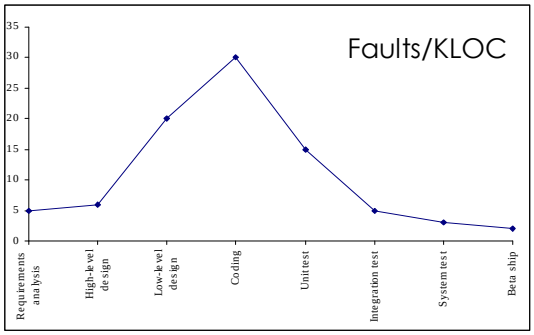
„Smoke test"

- A coarse form of regression test to determine that the product doesn't simply crash as a result of recent changes.

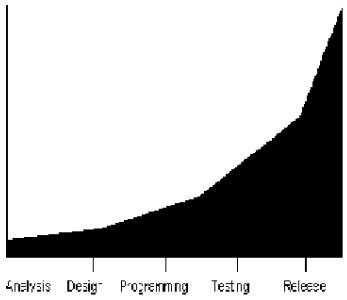
Fault vs Failure

- Fault – an error in the code
- Failure – the manifestation of a fault at runtime
- During testing we look for failures
  - A failure may be caused by many faults
  - A fault may cause many failures
  - Many to many relationship
- Flaw – an error in the design

Typical fault profile



Relative Cost of Fixing Errors



Relative Cost of Fixing Errors

Y2K problem

- Cost ~1/1000 (ignoring inflation) as much to fix at design/coding time as it actually took to fix in the field

Black-Box Testing

- Also called behavioral testing, functional testing, data-driven or input/output-driven testing
- Unconcerned about the internal program structure
- Test planning- may start when the spec is circulated
- All kinds of testing e.g.
  - Acceptance/qualification testing – check whether the program is stable enough to be tested
  - Function test and system test – check it against spec
  - Beta testing – get user feedback
  - Release testing – check all things that will go to the customer /manufacturer, e.g. test the ad copy and sales materials
  - Final acceptance testing by the customer

White-Box Testing

- Logic-driven: examine the internal program structure
- Part of coding stage
  - Unit and low-level components are often tested by the structural strategies
  - Integration testing

Result-oriented testing

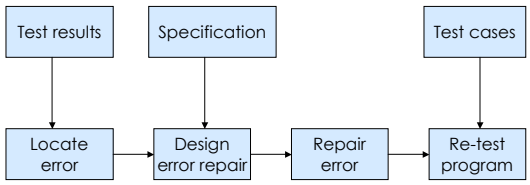
- Scope-specific test design (unit, integration, system) and techniques (white/blackbox) are organized into a coherent whole.
- Design test cases using scope-appropriate responsibility (functionality) -based patterns.
- Develop efficient test suites that exercise many responsibilities and component interfaces with just a few test cases.
- If the components are not trusted, choose an integration cycle based on dependencies to control introduction of untrusted parts.
- Develop and execute the test suite according to this cycle.
- Determine responsibility coverage by analysis and implementation coverage by instrumentation.
- Stop testing when the modeled responsibilities and part interfaces have been covered.

Testing and debugging

- Defect testing and debugging are distinct processes
- V & V is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

The debugging process

Testing != debugging



Inspections and testing

- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering errors

Inspections and testing – cont.

- Many different defects may be discovered in a single inspection
- In testing, one defect may mask another so several executions are required
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V&V process
- Inspections can check conformance with a specification, but not conformance with the customer's real requirements
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Summary

- Verification
  „*Are we building the product right?*"
- Validation
  „*Are we building the right product?*"
- Software testing is **running the code** for combinations of states and input data **in order to detect faults**.
- „Nie można założyć, że z poszczególnych poprawnych części zawsze powstaje poprawna całość"
- Testing != debugging
- Testing != Inspections

Literature and Links

- Robert V. Binder: „Testowanie systemów obiektowych. Modele wzorce i narzędzia." WNT 2003
- Software Test Automation – Effective use of test execution tools; Mark Fewster and Dorothy Graham
- Software horror stories
  http://www.cs.tau.ac.il/~nachumd/horror.html

Quality Assessment

Thank You for your attention ☺

- What is your general impression (1-6)
- Was it too slow or too fast?
- What important did you learn during the lecture?
- What to improve and how?