Software Architecture

Bertrand Meyer

Lecture 10: Testing Object-Oriented Software

Ilinca Ciupa
(Geoffrey James – *The Zen of Programming*, 1988)

“Thus spoke the master: “Any program, no matter how small, contains bugs.”

The novice did not believe the master’s words. “What if the program were so small that it performed a single function?” he asked.

“Such a program would have no meaning,” said the master, “but if such a one existed, the operating system would fail eventually, producing a bug.”

But the novice was not satisfied. “What if the operating system did not fail?” he asked.
“There is no operating system that does not fail,” said the master, “but if such a one existed, the hardware would fail eventually, producing a bug.”

The novice still was not satisfied. “What if the hardware did not fail?” he asked.

The master gave a great sigh. “There is no hardware that does not fail”, he said, “but if such a one existed, the user would want the program to do something different, and this too is a bug.”

A program without bugs would be an absurdity, a nonesuch. If there were a program without any bugs then the world would cease to exist.”
Agenda for today

- What testing is and what it is not
- Terminology
  - bugs
  - types of tests
  - components of a test
- Partition testing
- Black-box vs white-box testing
- Measuring test quality
  - code coverage
  - mutation testing
- Testing strategy
- Test-driven development
- Test automation
- Contract-based testing
Assignment 3: Testing

- [Link](http://se.inf.ethz.ch/teaching/ss2006/0050/exercises/exercise3.pdf)
- Issued: 7 June 2006
- Due: 20 June 2006
“Software testing is the execution of code using combinations of input and state selected to reveal bugs.”

“Software testing [...] is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs.”

What does testing involve?

- Determine the **parts** of the system to be tested

- Find **input values** which should bring significant information

- **Run** the software on the input values

- **Compare** the produced **results** to the expected ones

- (Measure execution characteristics: time, memory used, etc)
"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."


What testing can do: find bugs
What testing cannot do: prove the absence of bugs
What testing is not

- Testing ≠ debugging
  - When testing uncovers an error, debugging is the process of removing that error

- Testing ≠ program proving
  - Formal correctness proofs are mathematical proofs of the equivalence between the specification and the program
Common abbreviations

- IUT – implementation under test
- MUT – method under test
- OUT – object under test
- CUT – class/component under test
- SUT – system under test
Bug-related terminology

- **Failure** – manifested inability of the IUT to perform a required function; evidenced by:
  - Incorrect output
  - Abnormal termination
  - Unmet time or space constraints

- **Fault** – incorrect or missing code
  - Execution may result in a failure

- **Error** – human action that produces a software fault

- **Bug** – error or fault

caused by Errors
result from Failures
result from Faults
Hopper’s bug

9/9

0800&000, a thread started.

1200 (04) & 000, a thread started.

Relays 6-2 in 033 failed special speed test.

In detail:

Relays were changed.

1300 Started Cosine Tape (Sine check).

1525 Started Multi Adder Test.

1545 Relay #70 Panel E (moth) in relay.

First actual case of bug being found.

1700 and entered.

1900 closed form.
We could, for instance, begin with cleaning up our language by no longer calling a bug “a bug” but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer’s own creation. The nice thing about this simple change of vocabulary is that it has such a profound effect. While, before, a program with only one bug used to be “almost correct”, afterwards a program with an error is just “wrong”...

E. Dijkstra, *On the cruelty of really teaching computer science* (December 1989)
Testing scope

- **Unit test** – typically a relatively small executable

- **Integration test** – a complete system or subsystem of software and hardware units
  - Exercises interfaces between units to demonstrate that they are collectively operable

- **System test** – a complete integrated application
  - Focuses on characteristics that are present only at the level of the entire system
  - Categories: functional, performance, stress or load
Intent (1)

- Fault-directed testing – reveal faults through failures
  - Unit and integration testing
- Conformance-directed testing – to demonstrate conformance to required capabilities
  - System testing
- Acceptance testing – enable a user/customer to decide whether to accept a software product
Intent (2)

- **Regression testing** – retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made
- **Mutation testing** – purposely introducing faults in the software in order to estimate the quality of the tests
Components of a test

- **Test case** – specifies:
  - The state of the IUT and its environment before test execution
  - The test inputs
  - The expected result

- **Expected results** – what the IUT should produce:
  - Returned values
  - Messages
  - Exceptions
  - Resultant state of the IUT and its environment

- **Oracle** – produces the results expected for a test case
  - Can also make a pass/no pass evaluation
Test execution

- Test suite – collection of test cases
- Test driver – class or utility program that applies test cases to an IUT
- Stub – partial, temporary implementation of a component
  - May serve as a placeholder for an incomplete component or implement testing support code
- Test harness – a system of test drivers and other tools to support test execution
Partition testing

- Partition – divides the input space into groups which hopefully have the property that any value in the group will produce a failure if a bug exists in the code related to that partition

- Examples of partition testing:
  - Equivalence class – a set of input values so that if any value in the set is processed correctly (incorrectly) then any other value in the set will be processed correctly (incorrectly)
  - Boundary value analysis
  - Special values testing
Choosing values

- **Each Choice (EC):** A value from each set for each input parameter must be used in at least one test case.

- **All Combinations (AC):** A value from each set for each input parameter must be used with a value from every set for every other input parameter.
Partition testing: conclusions

- Applicable to **all levels** of testing – unit, class, integration, system, etc.

- Divides the **input space** of the program into partitions, based on the specification and/or on the implementation

- It’s probably what you’re doing unconsciously anyway
<table>
<thead>
<tr>
<th>Black box testing</th>
<th>White box testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses no knowledge of the internals of the SUT</td>
<td>Uses knowledge of the internal structure and implementation of the SUT</td>
</tr>
<tr>
<td>Also known as responsibility-based testing and functional testing</td>
<td>Also known as implementation-based testing or structural testing</td>
</tr>
<tr>
<td>Goal: to test how well the SUT conforms to its requirements (Cover all the requirements)</td>
<td>Goal: to test that all paths in the code run correctly (Cover all the code)</td>
</tr>
</tbody>
</table>
Black box vs white box testing (2)

<table>
<thead>
<tr>
<th>Black box testing</th>
<th>White box testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses no knowledge of the program except its specification</td>
<td>Relies on source code analysis to design test cases</td>
</tr>
<tr>
<td>Typically used in integration and system testing</td>
<td>Typically used in unit testing</td>
</tr>
<tr>
<td>Can also be done by user</td>
<td>Typically done by programmer</td>
</tr>
</tbody>
</table>

Chair of Software Engineering
White box testing

- Allows you to look inside the box
- Some people prefer “glass box” or “clear box” testing
Measures of test quality

- Code coverage
- Data coverage
- Specification coverage
- Mutation testing
Coverage

- General notion expressing a **percentage** of elements (defined by a test strategy) exercised by a test suite
- A certain coverage measure is **achieved** by a test suite if 100% of the required elements have been exercised
  - e.g.: "This test suite achieves statement coverage for method $m$"
  - $\Rightarrow$ every statement in method $m$ is executed by at least one test case in the test suite
Code coverage

- **Code coverage** - how much of your code is exercised by your tests

- **Code coverage analysis** = the process of:
  - Computing a measure of coverage (which is a measure of test suite quality)
  - Finding sections of code not exercised by test cases
  - Creating additional test cases to increase coverage
Code coverage analyzer

- Tool that automatically computes the coverage achieved by a test suite

- Steps involved:
  1. Source code is instrumented by inserting trace statements.
  2. When the instrumented code is run, the trace statements produce a trace file.
  3. The analyzer parses the trace file and produces a coverage report (example).
Basic measures of code coverage

- **Statement coverage** – reports whether each executable statement is encountered
  - Disadvantage: insensitive to some control structures
- **Decision coverage** – reports whether boolean expressions tested in control structures evaluate to both true and false
  - Also known as branch coverage
- **Condition coverage** – reports whether each boolean sub-expression (separated by logical-and or logical-or) evaluates to both true and false
- **Path coverage** – reports whether each of the possible paths in each function has been tested
  - Path = unique sequence of branches from the function entry to the exit point
Mutation testing

- Idea: make small changes to the program source code (so that the modified versions still compile) and see if your test cases fail for the modified versions

- Purpose: estimate the quality of your test suite
Terminology

- Faulty versions of the program = mutants
  - We only consider mutants that are not equivalent to the original program!

- A mutant is said to be killed if at least one test case detects the fault injected into the mutant

- A mutant is said to be alive if no test case detects the injected fault

- A mutation score (MS) is associated to the test set to measure its effectiveness

- A test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants
Mutation operators

- **Mutation operator** = a rule that specifies a syntactic variation of the program text so that the modified program still compiles

- **Mutant** = the result of an application of a mutation operator

- The quality of the mutation operators determines the quality of the mutation testing process.

- **Mutation operator coverage (MOC)**: For each mutation operator, create a mutant using that mutation operator.
Examples of mutants

Original program:

```java
if (a < b)
    b := b - a;
else
    b := 0;
```

Mutants:

```java
if (a < b)
    if (a <= b)
        b := b - a;
    else
        b := b + a;
    b := x - a;
else
    b := 0;
    b := 1;
    a := 0;
```
System test quality (STQ)

- S - system composed of n components denoted \( C_i, \ i \in [1..n] \)
- \( d_i \) - number of killed mutants after applying the unit test sequence to \( C_i \)
- \( m_i \) - total number of mutants of \( C_i \)
- the mutation score \( MS \) for \( C_i \) being given a unit test sequence \( T_i \):
  \[
  MS(C_i, T_i) = \frac{d_i}{m_i}
  \]
- \( STQ(S) = \frac{\sum_{i=1,n} d_i}{\sum_{i=1,n} m_i} \)
- STQ is a combined measure of test suite quality and contract quality
Testing strategy

How to plan and structure the testing of a large program:

- **Who is testing?**
  - Developers / special testing teams / customer
  - It is hard to test your own code

- **What test levels are needed?**
  - Unit, integration, system, acceptance, regression test

- **How is it done in practice?**
  - Manual testing
  - Testing tools
  - Automatic testing
Gosh darn, another bug! What's wrong with you programmers, anyway?

Not so fast, young programmer. Have you ever heard of the three questions?

No, is it a rock group?

Old geezer, probably still writes in FORTRAN.

The first question is: Have I made this error anywhere else?

I guess I should check the other index calculations.

The second question is: What happens when I fix the bug?

Will my fix cause a bug later?

The third question is: How can I change my ways to make this kind of bug impossible?

This stuff should really be done in a define.

Hi-yo Pascal, awaaaah!

Who was that old hacker anyway?
“Three questions about each bug you find” (Van Vleck):

- “Is this mistake somewhere else also?”
- “What next bug is hidden behind this one?”
- “What should I do to prevent bugs like this?”
Test-driven development (TDD)

- Software development methodology
- One of the core practices of extreme programming (XP)
- Steps:
  1. Write a small test.
  2. Write enough code to make the test succeed.
  3. Clean up the code.
  4. Repeat.
- The testing in TDD is *unit testing* + *acceptance testing*
- Always used together with xunit
Test-driven development (TDD)

- Evolutionary approach to development
- Combines
  - Test-first development
  - Refactoring
- Primarily a method of software design
  - Not just method of testing
TDD 1: test-first development (TFD)

1. Add a test
2. Run the tests
   - [Pass]
   - [Fail]
3. Make a little change
4. [Development continues]
5. Run the tests
   - [Fail]
   - [Development stops]

Copyright 2003 Scott W. Ambler
TDD 2: refactoring

A change to the system that leaves its behavior unchanged, but enhances some non-functional qualities:

- Simplicity
- Understandability
- Performance

Refactoring does not fix bugs or add new functionality.
TDD & traditional testing

TDD is a programming technique that ensures that source code is thoroughly unit tested

Need remains for:

- Functional testing
- User acceptance testing
- System integration testing

XP suggests these tests should also occur early
xunit

- The generic name for any test automation framework for unit testing
  - **Test automation framework** – provides all the mechanisms needed to run tests so that only the test-specific logic needs to be provided by the test writer
- Implemented in all the major programming languages:
  - JUnit – for Java
  -/cppunit – for C++
  - SUnit – for Smalltalk (the first one)
  - PyUnit – for Python
  - vbUnit – for Visual Basic
JUnit: Overview

- Provides a **framework for running test cases**

- Test cases
  - Written manually
  - Normal classes, with annotated methods

- Input values and expected results defined by the tester

- Execution is the only automated step
JUnit: Resources

- Unit testing framework for Java
- Written by Erich Gamma and Kent Beck
- Open source (CPL 1.0), hosted on SourceForge
- Current version: 4.0
- Available at: [www.junit.org](http://www.junit.org)


Push-button testing

- Never write a test case, a test suite, a test oracle, or a test driver
- Automatically generate
  - Objects
  - Feature calls
  - Evaluation and saving of results
- The user must only specify the SUT and the tool does the rest (test generation, execution and result evaluation)
“Design by Contract implemented with assertions is a straightforward and effective approach to built-in test. Not only does this strategy make testing more efficient, but it is also a powerful bug prevention technique.”

A contract violation always signals a bug:

- Precondition violation: bug in client
- Postcondition violation: bug in routine
Assertions as built-in test (BIT)

- Must be executable
- An executable assertion has 3 parts:
  - A predicate expression
    - In Eiffel: boolean expression + \text{old} notation
  - An action
    - Executed when an assertion violation occurs
  - An enable/disable mechanism
Benefits and limitations of assertions as BIT

- **Advantages:**
  - BIT can evaluate the internal state of an object without breaking encapsulation
  - Contracts written before or together with implementation

- **Limitations inherent to assertions**
  - Frame problem

- **The quality of the test is only as good as the quality of the assertions**
Errors in the testing system

- Bugs in test design
- Bugs in oracle (faulty contracts)
  - Unsatisfiable contracts
  - Omissions in contracts
  - Incorrect translation of the specification into contracts
- Bugs in test driver
Reference texts for testing (1)

- OO testing “bible”:
- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, in preparation
- Writing unit tests with JUnit:
  Erich Gamma and Kent Beck: *Test Infected: Programmers Love Writing Tests*
- Code coverage:
  [http://www.bullseye.com/coverage.html](http://www.bullseye.com/coverage.html)
- Mutation testing:
Reference texts for testing (2)

- Test-driven development:
Testing tools references

- JUnit:  http://www.junit.org/index.htm
- Gobo Eiffel Test:
  http://www.gobosoft.com/eiffel/gobo/getest/
- AutoTest:
  http://se.inf.ethz.ch/people/leitner/auto_test/
End of lecture 10

Many thanks to Andreas Leitner for providing some of the slides used in this lecture