Trusted Components
Bertrand Meyer, Manuel Oriol

Lecture 7: Testing Object-Oriented Software
Ilinca Ciupa, Andreas Leitner, Bertrand Meyer
A (rather unorthodox) introduction (1)

(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

- Why test?
- Test basics
- Unit testing (JUnit)
- Specification-based testing
- Test case generation
- Measuring test quality
Agenda for today

- Why test?
- Test basics
- Unit testing (JUnit)
- Specification-based testing
- Test case generation
- Measuring test quality
Here’s a thought...

- "Imagine if every Thursday your shoes exploded if you tied them the usual way. This happens to us all the time with computers, and nobody thinks of complaining."

  Jef Raskin, Apple Computer, Inc.
NIST report on testing (May 2002)

- Financial consequences, on developers and users, of “insufficient testing infrastructure”: $59.5 B.
  - Finance $3.3 B
  - Car and aerospace $1.8 B. etc.
Relative cost to correct a defect

Agenda for today

- Why test?
- **Test basics**
  - Unit testing (JUnit)
  - Specification-based testing
  - Test case generation
  - Measuring test quality
Test basics: topics

- Definition
- Components of a test
- Types of tests
  - With respect to scope
  - With respect to intent
  - White-box vs. black-box
- How to find the inputs: partition testing
- Testing strategy
- Testing and bug prevention
Definition: testing

“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 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
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

Errors ➔ Faults ➔ Failures

Faults ➔ Failures

Failures ➔ Caused by Errors
Hopper’s bug

0800 Anton started
1000 - Anton stopped
1330 1304.76,416.995,502.5 1.23457.893.215.678.910.112
0303 1204.76415.4678910.112
0307 1204.76415.4678910.112

Relays 5-2 in 033 failed special speed test
in relay

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

1545 Relay #70 Panel F
Moth in relay.

First actual case of bug being found.
1700 Anthony started.
1900 closed form.
Dijkstra’s criticism of the word “bug”

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. W. Dijkstra, *On the cruelty of really teaching computer science* (December 1989)
What does testing involve?

- Determine **which parts** of the system you want to test
- 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)
Components of a test

- **Test case** – specifies:
  - The state of the implementation under test (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
Types of tests w.r.t. scope

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

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

- **System test** – scope: a complete, integrated application
  - Focuses on characteristics that are present only at the level of the entire system
  - Categories:
    - Functional
    - Performance
    - Stress or load
Types of tests w.r.t. intent

- **Fault-directed testing** – intent: reveal faults through failures
  - Unit and integration testing

- **Conformance-directed testing** – intent: to demonstrate conformance to required capabilities
  - System testing

- **Acceptance testing** – intent: enable a user/customer to decide whether to accept a software product
Types of tests w.r.t. intent (continued)

- **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.
Testing and the development phases

- Unit testing – implementation
- Integration testing - subsystem integration
- System testing - system integration
- Acceptance testing – deployment
- Regression testing - maintenance
## Black box vs white box testing (1)

<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>
White box testing

- Allows you to look inside the box
- Some people prefer “glass box” or “clear box” testing
Partition testing

- If you can’t test every value of the input domain, how do you choose the inputs for your tests?
- One solution is partition testing
- **Partition** – divides the input space into sets which hopefully have the property that any value in the set will produce a failure if a bug exists in the code related to that partition
- A partition must satisfy two properties:
  - Completeness: the partition must cover the entire domain
  - Disjointness: the sets must not overlap
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.
Testing strategy

How do we 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 do we need?
  - Unit, integration, system, acceptance, regression test

- How do we do it in practice?
  - Manual testing
  - Testing tools
  - Automatic testing
Tom Van Vleck,
ACM SIGSOFT
Software
Engineering Notes,
14/5, July 1989
Testing and bug prevention

“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?”
Testing basics: literature

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, in preparation
Agenda for today

- Why test?
- Test basics
- **Unit testing (JUnit)**
- Specification-based testing
- Test case generation
- Measuring test quality
- 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: 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
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
How to use JUnit

- Requires JDK 5

- **Annotations:**
  - @Test for every method that represents a test case
  - @Before for every method that will be executed before every @Test method
  - @After for every method that will be executed after every @Test method

- Every @Test method must contain some check that the actual result matches the expected one – use asserts for this
  - assertTrue, assertFalse, assertEquals, assertNull, assertNotNull, assertSame, assertNotSame
Example: basics

```java
package unitests;

import org.junit.Test;  // for the Test annotation
import org.junit.Assert;  // for using asserts
import junit.framework.JUnit4TestAdapter;  // for running

import ch.ethz.inf.se.bank.*;

public class AccountTest {
    @Test
    public void initialBalance() {
        Account a = new Account("John Doe", 30, 1, 1000);
        Assert.assertEquals("Initial balance must be the one set through the constructor", 1000, a.getBalance());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTest.class);
    }
}
```

To declare a method as a test case.

To compare the actual result to the expected one.

Required to run JUnit4 tests with the old JUnit runner.
Example: set up and tear down

```java
package unitests;

import org.junit.Before; // for the Before annotation
import org.junit.After; // for the After annotation
// other imports as before...

public class AccountTestWithSetUpTearDown {
    private Account account;

    @Before
    public void setUp() {
        account = new Account("John Doe", 30, 1, 1000);
    }

    @After
    public void tearDown() {
        account = null;
    }

    @Test
    public void initialBalance() {
        Assert.assertEquals("Initial balance must be the one set through the constructor",
                          1000,
                          account.getBalance());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AccountTestWithSetUpTearDown.class);
    }
}
```

To run this method before any `@Test` method

To run this method after any `@Test` method

Must make account an attribute of the class now

To run this method before any `@Test` method
@BeforeClass, @AfterClass

- A method annotated with @BeforeClass will be executed once, before any of the tests in that class is executed.
- A method annotated with @AfterClass will be executed once, after all of the tests in that class have been executed.
- Can have several @Before and @After methods, but only one @BeforeClass and @AfterClass method respectively.
Checking for exceptions

- Pass a parameter to the `@Test` annotation stating the type of exception expected:

```java
@Test(expected=AmountNotAvailableException.class) public void overdraft () throws AmountNotAvailableException {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.withdraw(1001);
}
```

- The test will fail if a different exception is thrown or if no exception is thrown.
Pass a parameter to the `@Test` annotation setting a timeout period in milliseconds. The test fails if it takes longer than the given timeout.

```java
@Test(timeout=1000) public void testTimeout () {
    Account a = new Account("John Doe", 30, 1, 1000);
    a.infiniteLoop();
}
```
Agenda for today

- Why test?
- Test basics
- Unit testing (JUnit)
- Specification-based testing
- Test case generation
- Measuring test quality
Design by Contract: applications

- Built-in correctness
- Automatic documentation
- Testing and debugging
- Get inheritance right
- Get exceptions right
- Give managers better control tools
Design by Contract: language support

- Eiffel
  www.eiffel.com

- For Java: numerous tools including JML
  ww.cs.iastate.edu/~leavens/JML/

- Spec# (Microsoft)
  research.microsoft.com/specsharp/
Contracts for testing and debugging

- Contracts express implicit assumptions behind code
- A bug is a discrepancy between intent and code
- Contracts state the intent!

- In EiffelStudio: select compilation option for run-time contract monitoring at level of:
  - Class
  - Cluster
  - System

- May disable monitoring when releasing software
- A powerful form of quality assurance
Run-time contract monitoring

A contract violation always signals a bug:

- Precondition violation: bug in client
- Postcondition violation: bug in routine
When testing a certain method:

- We have to satisfy its **precondition** (so that we can execute it)
- If it does not fulfill its **postcondition** → **BUG**

```plaintext
class ARRAYED_LIST [G] ...  
put (v: like item) is  
  -- Replace current item by `v'.  
  -- (Synonym for `replace')  
  require  
    extendible: extendible  
  do  
    ...  
  ensure  
    item_inserted: is_inserted (v)  
    same_count: count = old count  
end
```

precondition

body

postcondition
Assertions as built-in test (BIT)

- Must be executable
- An executable assertion has 3 parts:
  - A predicate expression
    - In Eiffel: boolean expression + 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
Agenda for today

- Why test?
- Test basics
- Unit testing (JUnit)
- Specification-based testing
- Test case generation
- Measuring test quality
Testing is tedious

Facts from a survey of 240 software companies in North America and Europe:

- 8% of companies release software to beta sites without any testing.
- 83% of organizations' software developers don't like to test code.
- 53% of organizations' software developers don't like to test their own code because they find it tedious.
- 30% don't like to test because they find testing tools inadequate.
Test automation

- Testing is so difficult and time consuming...
- So why not do it automatically?
- What is most commonly meant by “automated testing” currently is automatic test execution
- But actually...
Degrees of Automation

- No automation
- Automated execution
- Automated input generation
- Automated oracle
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 system under test and the tool does the rest (test generation, execution and result evaluation)
Challenges of Automated Testing

- Vast input space
- Is this input good?
  - Precondition
- Is this output good?
  - Postcondition

- The quality of the test is only as good as the quality of the assertions
Vast Input Space

- Input space typically unbounded
- Even when finite, very large
- Exhaustive testing impossible
- Number of test cases increases exponentially with number of input variables

```plaintext
foo (c: CHARACTER) is
do
  ...
end

bar (c1: CHARACTER, c2: CHARACTER) is
do
  ...
end
```
AutoTest

- Fully automated testing framework
  - Actual strategies are extensions
- Based on Design By Contract
- Robust execution
- Integration of manual unit tests

Available with source code from
http://se.inf.ethz.ch/people/leitner/auto_test/
The tool

AutoTest

System under test → Test results
Test scope → Counter-examples
Parameters →
Agenda for today

- Why test?
- Test basics
- Unit testing (JUnit)
- Specification-based testing
- Test case generation
- Measuring test quality
Measuring test quality: topics

- Code coverage
- Data coverage
- Mutation testing
Coverage

- General notion expressing a percentage of elements (defined by a test strategy) exercised by a test suite
- When we say that a certain coverage measure is achieved by a test suite, we mean 100% of the required elements have been exercised
  - e.g.: “This test suite achieves statement coverage for method \( m \)” \( \iff \) 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:
  - Source code is instrumented by inserting trace statements.
  - When the instrumented code is run, the trace statements produce a trace file.
  - 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
Code coverage tools

- **Emma**
  - Java
  - Open-source

- **JCoverage**
  - Java
  - Commercial tool
  - [http://www.jcoverage.com/](http://www.jcoverage.com/)

- **NCover**
  - C#
  - Open-source

- **Clover, Clover.NET**
  - Java, C#
  - Commercial tools
Dataflow-oriented testing

- Focuses on how variables are defined, modified, and accessed throughout the run of the program

- **Goal**: to execute certain paths between a definition of a variable in the code and certain uses of that variable
Access-related bugs

- Using an uninitialized variable
- Assigning to a variable more than once without an intermediate access
- Deallocating a variable before it is initialized
- Deallocating a variable before it is used
- Modifying an object more than once without accessing it
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
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:

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

Mutants:

```plaintext
if (a < b)
    if (a <= b)
    if (a > b)
    if (c < b)
        b := b - a;
    b := b + a;
    b := x - a;
else
    b := 0;
    b := 1;
    a := 0;
```
OO mutation operators

- Visibility-related:
  - **Access modifier change** – changes the visibility level of attributes and methods

- Inheritance-related:
  - **Hiding variable/method deletion** – deletes a declaration of an overriding or hiding variable/method
  - **Hiding variable insertion** – inserts a member variable to hide the parent’s version
OO mutation operators (continued)

- Polymorphism- and dynamic binding-related:
  - Constructor call with child class type – changes the dynamic type with which an object is created

- Various:
  - Argument order change – changes the order of arguments in method invocations (only if there exists an overloading method that can accept the changed list of arguments)
  - Reference assignment and content assignment replacement
    - example: list1 = list2.clone()
System test quality (STQ)

- S - system composed of n components denoted C_i
- d_i - number of killed mutants after applying the unit test sequence to C_i
- m_i - total number of mutants
- 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) = \[ \frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} m_i} \]
- In general, STQ is a measure of test suite quality
- If contracts are used as oracles, STQ is a combined measure of test suite quality and contract quality
Mutation tools

- muJava - [http://ise.gmu.edu/~ofut/mujava/](http://ise.gmu.edu/~ofut/mujava/)
Measuring test quality: literature

Discussion

- Is testing a way of increasing trust in the software?
- How much testing is enough?
- How do you decide which modules to test more thoroughly than others?
End of lecture 7