

Testing, Debugging, and Verification

Testing, Part II

Srinivas Pinisetty ¹

09 November 2018

¹Slides based on material from Wolfgang Aherndt...

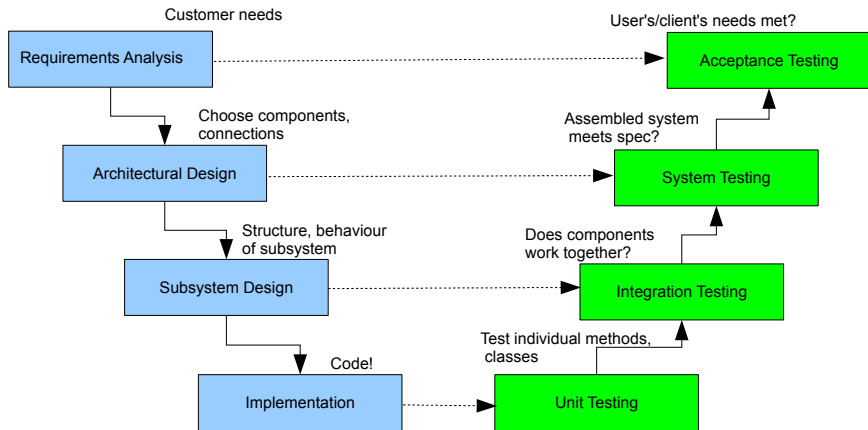
Lab and exercise sessions

- ▶ First lab session next week Monday
- ▶ First exercise session next week Thursday
- ▶ please bring laptops
- ▶ install relevant tools before
 - ▶ topic: testing
 - ▶ install JUnit beforehand
(version JUnit4 upwards)

Overview of this Lecture

- ▶ Testing levels and the V-model
- ▶ Focus on Unit Testing
- ▶ Terminology: Test case, test set, test suit, oracle
- ▶ Introduction to JUnit: a framework for rapid unit testing
- ▶ Extreme Testing using JUnit

Recall: The V-Model



(many variants!)

Pentium Bug (-94)

- ▶ Wrong result on floating point divisions.
- ▶ Missing entries in the lookup table.
- ▶ Rarely happened (on system level).
- ▶ Easy catch in **unit test**.

Pentium Bug (-94)

- ▶ Wrong result on floating point divisions.
- ▶ Missing entries in the lookup table.
- ▶ Rarely happened (on system level).
- ▶ Easy catch in **unit test**.

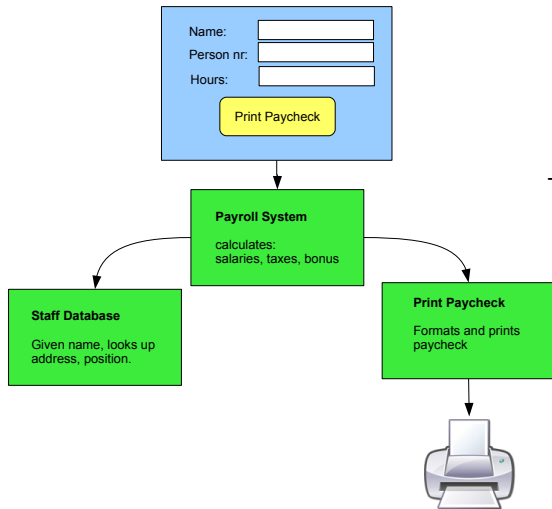
Ariane 5 Rocket (-96)

- ▶ Exploded 5 secs. after takeoff.
- ▶ Used guidance system from Ariane 4.
- ▶ Flight trajectory was different. Lacked **system testing**.

Mars Lander (-99)

- ▶ Crashed on landing.
- ▶ Mismatch in units: imperial vs. metric.
- ▶ Lacking in **integration testing**.

Discussion: Testing Levels of a System for Printing Paychecks



Think of examples of:

- ▶ System Tests
- ▶ Integration Tests
- ▶ Unit Tests

Some examples of Tests

- ▶ System Tests
 - ▶ Enter data in GUI, does it print the correct paycheck, formatted as expected?

Some examples of Tests

▶ System Tests

- ▶ Enter data in GUI, does it print the correct paycheck, formatted as expected?

▶ Integration Tests

- ▶ Payroll asks database for staff data, are values what's expected? Maybe there are special characters (unexpected!).
- ▶ Are paychecks formatted correctly for different kinds of printers?

Some examples of Tests

▶ System Tests

- ▶ Enter data in GUI, does it print the correct paycheck, formatted as expected?

▶ Integration Tests

- ▶ Payroll asks database for staff data, are values what's expected? Maybe there are special characters (unexpected!).
- ▶ Are paychecks formatted correctly for different kinds of printers?

▶ Unit Tests

- ▶ Does payroll system compute correct tax-rate, bonus etc?
- ▶ Does the Print Paycheck button react when clicked?
- ▶ ...

Regression Testing

Orthogonal to the above testing levels:

Regression Testing

- ▶ Testing that is done **after changes** in the software.
- ▶ **Purpose:**
gain confidence that the change(s) did not cause (new) failures.
- ▶ Standard part of the maintenance phase of software development.

E.g. Suppose Payroll subsystem is updated. Need to re-run tests (which ones?).

Unit Testing

Rest of testing part of the course: **focusing largely on unit testing**

recall: **unit testing** = procedure testing = (in oo) **method testing**

major issues in unit testing:

1. unit test cases (**'test cases'** in short)
2. order in which to test and integrate units

start with 1.

The science of testing is largely the science of test cases.

What does a test case consists of?

(to be refined later)

Test case

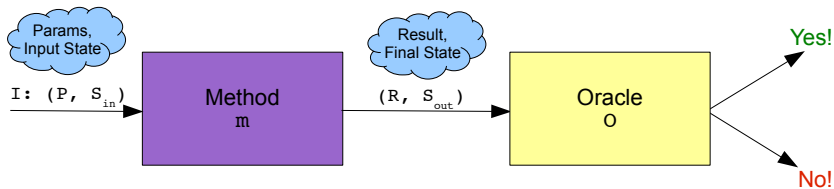
- ▶ Initialisation (of class instance and input arguments)
 - ▶ Call to the method under test.
 - ▶ Decision (oracle) whether the test **succeeds or fails**
-
- ▶ two first parts seem enough for a test case,
 - ▶ but test oracle is vital for *automated* evaluation of test

'Success' vs. 'Failure' of Tests

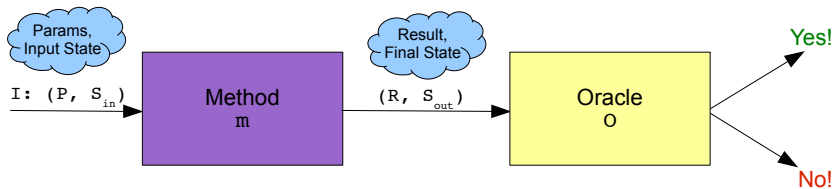
What does it mean for a test to **succeed**?

... or **fail**?

Test Cases, more precise



Test Cases, more precise



More formally...

A **test case** is a tuple $\langle m, I, O \rangle$ of **method** m , **input** I , and **oracle** O , where

- ▶ m is the method under test
- ▶ I is a tuple $\langle P, S_{in} \rangle$ of **call parameters** P and **initial state** S_{in}
- ▶ $O(R, S_{out}) \mapsto \{pass, fail\}$ is a **function** on **return value** R and **final state** S_{out} , telling whether they comply with **correct behaviour**

A **test set** TS^m for a (Java) method m consists of n test cases:

$$TS^m = \{\langle m, I_1, O_1 \rangle, \dots, \langle m, I_n, O_n \rangle\}$$

In general, O_i is **specific** for **each** test case!

A **test suite** for methods m_1, \dots, m_k is a union of corresponding test sets:

$$TS^{m_1} \cup \dots \cup TS^{m_k}$$

Automated and Repeatable Testing

Basic idea: write **code that performs the tests**.

- ▶ By using a tool you can automatically run a large collection of tests
- ▶ The testing code can be integrated into the actual code, thus stored in an organised way
- ▶ side-effect: **documentation**
- ▶ **After debugging, the tests are rerun** to check if failure is gone
- ▶ Whenever code is extended, all old test cases can be rerun to check that nothing is broken (**regression testing**)

Automated and Repeatable Testing (cont'd)

We will use **JUnit** for writing and running the test cases.

JUnit: small tool offering

- ▶ some functionality repeatedly needed when writing test cases
- ▶ a way to annotate methods as being test cases
- ▶ a way to run and evaluate test cases automatically in a batch

- ▶ JAVA testing framework to write and run automated tests
- ▶ JUnit features include:
 - ▶ Assertions for testing expected results
 - ▶ Annotations to designate test cases
 - ▶ Sharing of common test data
 - ▶ Graphical and textual test runners
- ▶ JUnit is widely used in industry
- ▶ JUnit used from command line or within an IDE (e.g., Eclipse)

(Demo)

Reflection: Extreme Testing

- ▶ JUnit designed for **Extreme Testing** paradigm
- ▶ Extreme Testing part of **Extreme Programming**
(but not depending on that)

Reflection: Extreme Testing (cont'd)

A few words Extreme Programming

(no introduction here, but see [Myers], Chapter 8)

- ▶ Extreme Programming (XP) invented by Beck (co-author of JUnit)
- ▶ Most popular agile development process
- ▶ Must create tests first, then create code basis
- ▶ Must run unit tests for every incremental code change

Reflection: Extreme Testing (cont'd)

A few words Extreme Programming

(no introduction here, but see [Myers], Chapter 8)

- ▶ Extreme Programming (XP) invented by Beck (co-author of JUnit)
- ▶ Most popular agile development process
- ▶ Must create tests first, then create code basis
- ▶ Must run unit tests for every incremental code change
- ▶ Motivation:
 - ▶ oo programming allows rapid development
 - ▶ still, quality is not guaranteed
 - ▶ aim of XP: create quality programs in short time frames
- ▶ XP relies heavily on unit and acceptance testing

modules (classes) must have unit tests before coding begins

benefits:

- ▶ You gain confidence that code will meet specification.
- ▶ You better understand specification and requirements.
- ▶ You express end result before you start coding.
- ▶ You may implement simple designs and optimise later **while reducing the risk of breaking the specification.**

Extreme Testing Example: Class Money

```
class Money {
    private int amount;
    private Currency currency;

    public Money(int amount, Currency currency)
    {
        this.amount = amount;
        this.currency = currency;
    }
    public Money add(Money m) {
        // NO IMPLEMENTATION YET, WRITE TEST FIRST
    }
}

class Currency {
    private String name;
    public Currency(String name) {
        this.name = name;
    }
}
```

Write a Test Case for add()

```
import org.junit.*;
import static org.junit.Assert.*;

public class MoneyTest {

    @Test public void simpleAdd() {
        Currency sek = new Currency("SEK");
        Money m1 = new Money(120, sek);
        Money m2 = new Money(160, sek);
        Money result = m1.add(m2);
        Money expected = new Money(280, sek);
        assertTrue(expected.equals(result));
    }
}
```

@Test is an **annotation**, turning **simpleAdd** into a **test case**

Example: Class Money

Now, implement the method under test, and **make sure it fails**

```
class Money {  
    private int amount;  
    private Currency currency;  
  
    ....  
  
    public Money add(Money m) {  
        return null;  
    }  
}
```

Compile and Run JUnit test class

- ▶ JUnit reports failure
- ▶ Produce first 'real' implementation

Example: Class Money

First real attempt to implement the method under test

```
class Money {
    private int amount;
    private Currency currency;

    public Money(int amount, Currency currency)
    {
        this.amount = amount;
        this.currency = currency;
    }

    public Money add(Money m) {
        return new Money(amount+m.amount,
            currency);
    }
}
```


Compile and Run JUnit test class

- ▶ JUnit will **still** report failure
- ▶ Fix possible defects, until test passes.
 - ▶ Can you spot it?
- ▶ What if we have different currencies?

Extend Functionality

Extend Money with Euro-exchange-rate **first in test cases**

Extend Functionality

Extend Money with Euro-exchange-rate **first in test cases**

```
public class MoneyTest {  
    @Test public void simpleAdd() {  
        Currency sek = new Currency("SEK", 9.01);  
        Money m1 = new Money(120, sek);  
        ....  
    }  
    @Test public void addDifferentCurr() {  
        Currency sek = new Currency("SEK", 9.01);  
        Money m1 = new Money(120, sek);  
        Currency nok = new Currency("NOK", 7.70);  
        Money m2 = new Money(160, nok);  
        Money result = m1.add(m2);  
        Money expected = new Money(  

```

Extend Functionality

Extend Money with Euro-exchange-rate **first in test cases**

```
public class MoneyTest {  
    @Test public void simpleAdd() {  
        Currency sek = new Currency("SEK", 9.01);  
        Money m1 = new Money(120, sek);  
        ....  
    }  
    @Test public void addDifferentCurr() {  
        Currency sek = new Currency("SEK", 9.01);  
        Money m1 = new Money(120, sek);  
        Currency nok = new Currency("NOK", 7.70);  
        Money m2 = new Money(160, nok);  
        Money result = m1.add(m2);  
        Money expected = new Money(307, sek);  
        assertTrue(expected.equals(result));  
    }  
}
```

Change, **and test** implementation

Common Parts into Test Fixture

```
public class MoneyTest {
    private Currency sek;
    private Money m1;

    @Before public void setUp() {
        sek = new Currency("SEK",9.01);
        m1 = new Money(120, sek);
    }

    @Test public void simpleAdd() {
        Money m2= new Money(140, sek);
        ....
    }

    @Test public void addDifferentCurr() {
        Currency nok = new Currency("NOK",7.70);
        Money m2 = new Money(160, nok);
        ...
    }
}
```

Testing a unit may require:

Stubs to replace **called** procedures

- ▶ Simulate behaviour of component not yet developed.
- ▶ E.g. test code that calls a method not yet implemented.

Drivers to replace **calling** procedures

- ▶ Simulate environment from where procedure is called.
- ▶ E.g. test harness.

Incremental Testing: Top-Down and Bottom-Up

Explore *incremental* test strategies, following call hierarchy:

Top-Down Testing

Test main procedure, then go down the call hierarchy

- ▶ requires stubs, but no drivers

Incremental Testing: Top-Down and Bottom-Up

Explore *incremental* test strategies, following call hierarchy:

Top-Down Testing

Test main procedure, then go down the call hierarchy

- ▶ requires stubs, but no drivers

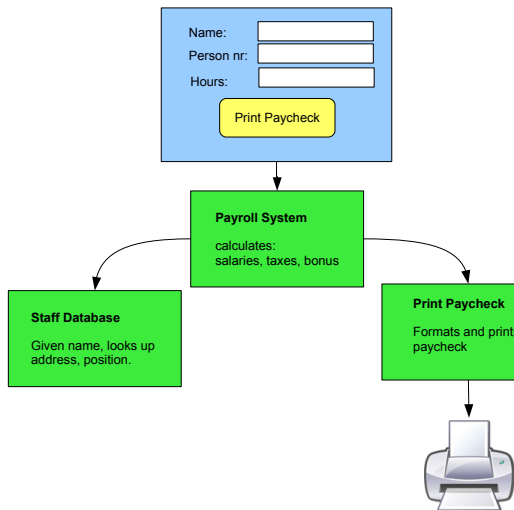
Bottom-Up Testing

Test leaves in call hierarchy, and move up to the root.

Procedure is not tested until all 'children' have been tested.

- ▶ requires drivers, but no stubs

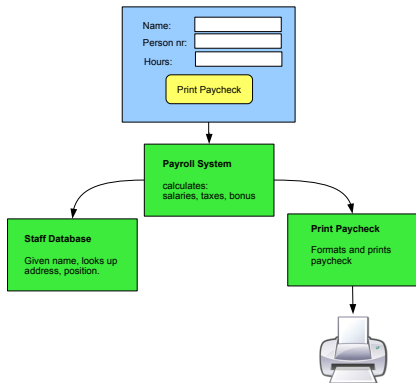
Discussion: Top-down vs Bottom-up Testing



How would you go about testing the Paycheck system

- ▶ Bottom-up?
 - ▶ Which **drivers** do you need?
- ▶ Top-down?
 - ▶ Which **stubs** do you need?

Discussion: Top-down vs Bottom-up Testing



- ▶ Bottom-up?
start from e.g. Print Paycheck, Staff Database
 - ▶ Which drivers do you need?
 - ▶ Driver replacing the caller, here Payroll System, Interface.
- ▶ Top-down?
start from e.g. Interface, then Payroll System
 - ▶ Which stubs do you need?
 - ▶ Stubs replacing called procedures, i.e. Payroll System, Staff Database, Print Paycheck.

Top-Down Testing: Pros and Cons

Advantages of Top-Down Testing

- ▶ Advantageous if major flaws occur toward top level.
- ▶ Early skeletal program allows demonstrations and boosts morale.

Disadvantages of Top-Down Testing

- ▶ Stubs must be produced (often more complicated than anticipated).
- ▶ Judgement of test results more difficult.
- ▶ Tempting to defer completion of testing of certain modules.

Bottom-Up Testing: Pros and Cons

Advantages of Bottom-Up Testing

- ▶ Advantageous if major flaws occur toward bottom level.
- ▶ Judgement of test results is easier.

Disadvantages of Bottom-Up Testing

- ▶ Driver units must be produced.
- ▶ The program as an entity does not exist until the last unit is added.