Programmieren II

Unit Testing & Test-Driven Development

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from Lars Vogel and T. Bögel)

July 2, 2014

Outline

Outline

OOP Principles

- Minimize Accessibility of Classes and Members (Encapsulation)
- Favor Composition over Inheritance
- Program to an Interface, not an Implementation
- The Open-Closed Principle (open for extension, closed for modification)
- The Single Choice Principle (**Strategy** Pattern)
- The Liskov Substitution Principle

Outline

Motivation for testing

- In 1998, NASA **lost** the USD 655 Million Mars Climate Orbiter
- "The peer review preliminary findings indicate that one team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation."
- Finding: had the interface between the two teams **been extensively tested**, this could have been avoided

`http://thephp.cc/viewpoints/blog/2014/03/disintegration-testing`

Motivation for unit testing

Why should you write unit tests?

- Make sure your code does what you want it to do
- Speed up development and optimization/refactoring
- Create better interfaces and functionality
- Get other people to use your contribution
- Make sure nobody else breaks your feature
- (Also, make sure **you** don't break your feature)

`http://ed.agadak.net/2008/03/why-you-should-write-unit-tests`

Good testing is not trivial

- Appropriate number of test cases?
- Implementing tests is tedious (stubs, mock objects etc.)

Categories of test cases

- Logical test cases: value ranges for input/output
- Concrete test cases: specific values for input/output

Typical test methods

Black box approach

- Testing an object *as an interface*
- No control about the internal structure of the test object
- Interface knowledge
- Examples: equivalence classes, boundary cases, state of objects

White box approach

- Testing different flows within a test object
- Exploits knowledge about the internal structure (code)
- Examples: coverage of statements, branches, conditions, paths

Test methods

Positive test

- Correct input (expects correct results)

Negative test

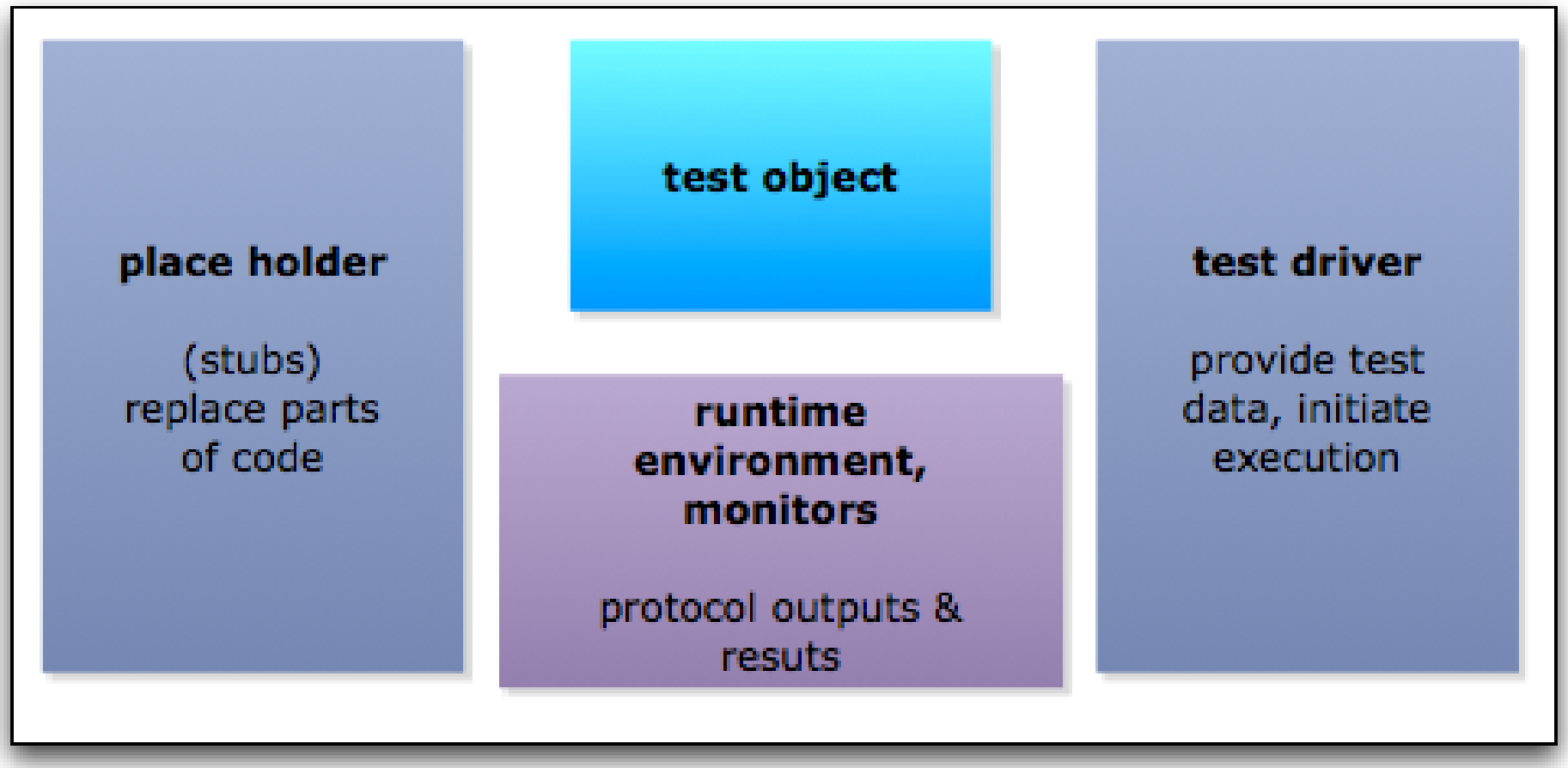
- Invalid input (expects appropriate exception handling)

Intuitive/empirical test

- Based on knowledge about error situations
- Should be tested alongside systematic test methods

Test execution

- Test framework (often: requirement to run tests)



Component tests

Component tests

- Component: self-contained unit of code, e.g. class, method, module
- White box and black box testing

Common errors

- Code does not terminate
- Erroneous or missing result
- Unexpected or wrong error message
- Inconsistent state of memory
- Unnecessary requirement of resources
- Unexpected behavior (e.g. crash)

Black box component test

- Testing a self-contained unit, e.g. an operation
- Typical test methods
  - Empirical or random tests
  - Equivalence classes (especially borderline cases)
  - State assessments
- Is unable to detect redundant parts of the code

Outline

Example: Test cases

Determine a suitable number of test cases.

```
int search (int[] a, int k)
pre: a.length > 0
post:
(result >= 0 and a[result] ==k) or
(result == -1 and
                  (not exists i: i>=0
                   and i< a.length
                   and a[i] == k)
)
```

Equivalence classes I

Equivalence class Subset of possible input values

Assumption Reaction of the code is identical for all values in the equivalence class

Test cases Cover all equivalence classes: at least one representative for each class

Borderline cases Test borderline cases

Typical equivalence classes

- Valid/invalid ranges of values
- Critical values
- Separation into buckets of similar output values

Equivalence classes II

Multiple input parameter

- Combine all valid equivalence classes of different input parameters
- Combine each invalid equivalence class with other, valid equivalence classes

Simplification

- Frequent combinations
- Testing critical values only
- Restrict test coverage to pairwise combinations
- Minimal: each valid equivalence class occurs in one test case

Example: equivalence classes I

Function

```
int search (int[] a, int k)
pre: a.length > 0
post:
(result >= 0 and a[result] ==k) or
(result == -1 and
                  (not exists i: i>=0
                   and i< a.length
                   and a[i] == k)
)
```

Example: equivalence classes II

Valid equivalence classes

- **Param1**: `a.length` $>$ `0`
- **Param2**: k in a, k not in a
- Refinement for *param1*:
  - `a.length = 1`
  - `a.length` $>$ `1`
- Refinement for *param2*:
  - k is first
  - in the middle
  - last element of a
  - not in a

Invalid equivalence class

- **Param1**: `a.length=0`

Test cases

a	**element** k	**result**
`[] (a.length=0)`	x	invalid
`[x] (a.length=1)`	x (in a)	1
`[y] (a.length=1)`	x (not in a)	-1
`[. . . , x, . . . ] (a.length>1)`	x (in the middle of a)	n
`[. . . ] (a.length>1)`	x (not in a)	-1
`[x,. . . ] (a.length>1)`	x (first element of a)	1
`[. . . ,x] (a.length>1)`	x (last element of a)	n

State assessment

- Considers not only input/output, but also history of states of a component
- Example: Stack
- States: initial, empty, filled, full, deleted
- Successful test: cover all states and branches at least once

White box test

- Coverage of statements
- Coverage of branches
- Coverage of paths
- Requirement: control flow diagram

Main advantage

- Each part of the code is tested

Disadvantages

- Difficulty of finding appropriate input values to cover all paths
- Difficulty of finding appropriate output values

$\rightarrow$ usually: **Grey Box** testing

Testing Classes I

Defining a structure of tests

- Constructors
- Getters
- Boolean Methods
- Setters
- Iterators
- Complex computations
- Other methods
- (Destructors)

Testing Classes II

Testing interactions between methods

- Dependencies between methods
  - Non-modal: no dependency
  - Uni-modal: fixed order (e.g. traffic light)
  - Quasi-modal: content dependent order (e.g. Stack)
  - Modal: functional dependency (e.g. bank account)

Fundamentals of Testing I

Principle 1

Complete test coverage is *impossible*.

Principle 2

"Program testing can be used to show the presence of bugs, but never to show their absence!" Edsger Dijkstra

Principle 3

Write tests early! Don't defer them to the end of the development cycle!

Principle 4

Bugs are not evenly distributed across the code. If a component contains multiple bugs, it is likely to contain even more.

Fundamentals of Testing II

Principle 5

Repeating tests doesn't give you new insights. You need to assess, update and modify your test cases.

Principle 6

Testing depends on the context and the application. Security relevant code needs to be tested more thoroughly. (Thanks, Captain Obvious!)

Principle 7

A program without bugs does not necessary fulfill the specification of a client.

Principle 8

Challenge: minimal number of test cases (= invested time) vs. highest quality.

Outline

Test-Driven Development

Test-Driven Development

- **Test-first** approach
- Thinking about tests before implementing functionality
- Writing tests before programming

Advantages

- All code is tested
- Forces you to think about functionality your code needs to provide
- Simple solutions (KISS): writing the smallest amount of code to make the test pass

Alternative programming workflow.

Fear

Test-driven development is a way of managing fear during programming. (Beck, 2002)

- Start simply.
- Write automated tests.
- Refactor to add design decisions one at a time.

The secrets of the ancients – 1960s

It was just assumed that any pro would do a damn good job of this activity, not to be embarrassed by unit bugs found in integration test or system test – or god help us, in production. (Jerry Weinberg)[1]

[1] source: http://secretsofconsulting.blogspot.de/2008/12/how-we-used-to-do-unit-testing.html

The secrets of the ancients – 1970s

It said the way to program is to look at the input tape and manually type in the output tape you expect. Then you program until the actual and expected tapes match.

I thought, what a stupid idea. I want tests that pass, not tests that fail. Why would I write a test when I was sure it would fail. ***Well, I'm in the habit of trying stupid things out just to see what happens, so I tried it and it worked great****.*

Rhythm of TDD

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

Outline

JUnit test framework

Based on the tutorial by Lars Vogel:
`http://www.vogella.com/articles/JUnit/article.html`

JUnit TestClass

- Separate class for testing (*test class*) that contains multiple test methods
- Test methods indicated by annotations
- Requirement: all methods can be executed in an arbitrary order (no dependencies)

Writing a test method

- Annotate a method with `@org.junit.Test` annotation
- Use a JUnit method to check the expected result versus the actual result

JUnit annotations (version 4.x) I

@Test Specifies that a method is a test method

@Before Method is executed before each test, e.g. to prepare the test environment (reading data, initializing classes etc.)

@After Method is executed after each test. Used to clean up the test environment, e.g. delete temporary data, free memory

@BeforeClass Method is executed once before the start of all tests. Use cases: connection to a database etc. Note: methods need be be declared as `static`

@AfterClass Method is executed once after all tests have been finished. Note: methods need to be declared as `static`

@Ignore Ignore a method

@Test (expected = Exception.class) Fails, if the method does not throw the named exception

@Test(timeout=100) Fails, if the method takes longer than 100 milliseconds

Assert statements I

Assert statements

- Special methods of the `Assert` class to test for certain conditions
- Required: expected result and actual result
- `[message]`: error message (optional)
- **Write meaningful error messages!**

- **fail(String)**: Let the method fail (e.g. to check that a certain part of the code is not reached)
- **assertTrue([message], boolean condition)** Checks that the boolean condition is true
- **assertsEquals([String message], expected, actual)** Tests whether two values are the same

Assert statements II

- **assertsEquals([String message], expected, actual, tolerance)** Test for matching floating point values. Tolerance: number of decimals that need to be identical
- **assertNull([message], object)** Checks that an object is `null`
- **assertNotNull([message], object)** Checks that an object is not `null`
- **assertSame([String], expected, actual)** Checks that both variables refer to the same object
- **assertNotSame([String], expected, actual)** Tests whether both variables refer to different objects

Test Suite

Test suites

- Multiple test classes can be combined into a *test suite*
- Running a test suite runs all corresponding test classes

Example: two test classes

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({TestClassA.class, TestClassB.class})
public class AllTests {
}
```

Practical workflow

Outside of eclipse

- Add JUnit `jar` file (library) to the class path
- Method `runClasses()` in `org.junit.runner.JUnitCore`: run test classes
- Returns object of type `org.junit.runner.Result`
- `Result` provides information about failures, successful tests etc.

Unit testing with eclipse

- Eclipse supports most of the workflows necessary for JUnit
- You should use Eclipse

Class to test: Money.java I

```
public class Money {
    private final int amount;
    private final String currency;

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

    public int getAmount() {
        return amount;
    }

    public String getCurrency() {
        return currency;
    }

    public boolean equals(Object anObject) {
        if (anObject instanceof Money) {
```

Class to test: Money.java II

```
            Money money = (Money) anObject;
            return money.getCurrency().equals(getCurrency())
            && getAmount() == money.getAmount();
        }
        return false;
    }
}
```

Testing Money.java I

Things to test

- Constructor
- `equals()` method

Testing Money.java II

Testing the constructor

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

public class MoneyTest {

    @Test
    public void constructorShouldSetAmountAndCurrency() {
        Money m = new Money(10, "USD");

        assertEquals(10, m.getAmount());
        assertEquals("USD", m.getCurrency());
    }
}
```

Testing for exceptions I

- Modification of the constructor
- If amount < 0 or an invalid currency is used: throw `IllegalArgumentException`

Testing for exceptions II

Modified constructor

```
public Money(int amount, String currency) {
    if (amount < 0) {
        throw new IllegalArgumentException(
            "illegal amount: [" + amount + "]");
    }

    if (currency == null || currency.isEmpty()) {
        throw new IllegalArgumentException(
            "illegal currency: [" + currency + "]");
    }
    this.amount = amount;
    this.currency = currency;
}
```

Testing for exceptions III

Testing for the occurrence of an exception

```
@Test(expected = IllegalArgumentException.class)
    public void constructorShouldThrowIAEForInvalidAmount() {
        Money m = new Money(-10,"USD");
    }
```

Testing for exceptions IV

Running the tests from the command line

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MoneyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MoneyTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

Creating and running tests from eclipse

- Simply do File $->$ New $->$ JUnit test
- It will ask you for which class file you want to write a test
- Type in the test, and run it to see if it passes

Client and Address

Client and Address

- `Client`: a client stores a collection of `Address` objects
- `Address`: one address of a client
- We start with the test

Testing Client I

```
public class ClientTest {
    private Address addressA = new Address("street A");
    private Address addressB = new Address("street B");

    @Test
    public void afterCreationShouldHaveNoAddress() {
        Client client = new Client();
        assertEquals(0, client.getAddresses().size());
    }

    @Test
    public void shouldAllowToAddAddress() {
        Client client = new Client();
        client.addAddress(addressA);
        assertEquals(1, client.getAddresses().size());
        assertTrue(client.getAddresses().contains(addressA));
    }

    @Test
```

Testing Client II

```
    public void shouldAllowToAddManyAddresses() {
    Client client = new Client();
    client.addAddress(addressA);
    client.addAddress(addressB);
    assertEquals(2, client.getAddresses().size());
    assertTrue(client.getAddresses().contains(addressA));
    assertTrue(client.getAddresses().contains(addressB));
    }
}
```

→ problematic: code duplication (`Client` is instantiated in each method).

Creating Address

Address.java

- Address only needs to store an address as a String
- Constructor expects the string

```
public class Address {
    private String address;

    public Address(String add) {
        this.address = add;
    }

    public String getAddress() {
        return this.address;
    }
}
```

Creating `Client` I

Implementing Clients

- Collection of `Address` objects
- Size of collection is 0 after instantiation
- Provides methods for adding an address and getting all addresses

Creating Client II

Client.java

```
public class Client {

    private List<Address> addresses;

    public Client() {
        this.addresses = new ArrayList<Address>();
    }

    public List<Address> getAddresses() {
        return addresses;
    }

    public void addAddress(Address newAddress) {
        this.addresses.add(newAddress);
    }

}
```

Running initialization before testing

- To avoid to repeat the same code for each test: `@Before` annotation

```
public class ClientTest {
    private Address addressA = new Address("street A");
    private Address addressB = new Address("street B");
    private Client client;

    @Before
    public void setUp() {
        client = new Client();
    }

    // the rest of the code identical to the previous listing
    ...
}
```

assert vs. assertTrue

- Java defines an assert statement which is only actually checked if you give the jvm the "-ea" argument
  - The (good) intention is to allow programmers to develop using "-ea" and then to leave it away when code is in production
  - Unfortunately, in practice people don't use this!
- Instead use methods like assertTrue from JUnit in your code (everywhere, not only in tests)

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

public void myMethod(int i) {

  // code here modifies i, should never result in i being
      less than 1

  assertTrue("myMethod: i not greater than 0", i>0);
}
```

Outline

What is mocking?

Motivation

- Java classes usually depend on other classes
- *Mock object*: dummy implementation for interface or class
- User-defined output of certain method calls
- Benefit: testing without dependencies
- This is a commonly used form of **integration testing** (think of Mars Climate Observer)
- Example: data provider (database replacement)

Creating mock objects

- Manually (via code)
- Mock framework: create mock objects at runtime and define their behavior

Mock frameworks

jMock

`http://jmock.org/`

EasyMock

`http://easymock.org/`

Mockito

`http://code.google.com/p/mockito/`

Example: EasyMock I

Object instantiation

```
// ICalcMethod is the object which is mocked
ICalcMethod calcMethod = EasyMock.createMock(ICalcMethod.class)
    ;
```

Example: EasyMock II

Specifying output

- **expect()**: simulate a method with certain arguments
- **andReturn()**: return value of the method for specified parameters
- **times()**: how often the Mock object will be called

```
// setup the mock object
expect(calcMethod.calc(Position.BOSS)).andReturn(70000.0).times
    (2);
expect(calcMethod.calc(Position.PROGRAMMER)).andReturn(50000.0)
    ;
// Setup is finished need to activate the mock
replay(calcMethod);
```

Literature

Vogel, Lars.
JUnit - Tutorial
http://www.vogella.com/articles/JUnit/article.html

Ullenboom, Ch.
Javainsel-Blog: Java Tests mit Junit
http://www.tutego.de/blog/javainsel/2010/04/junit-4-tutorial-java-tests-mit-junit

Beck, Kent.
Test Driven Development: By Example.
Addison-Wesley Professional, 2002.