Object-Oriented Software Development

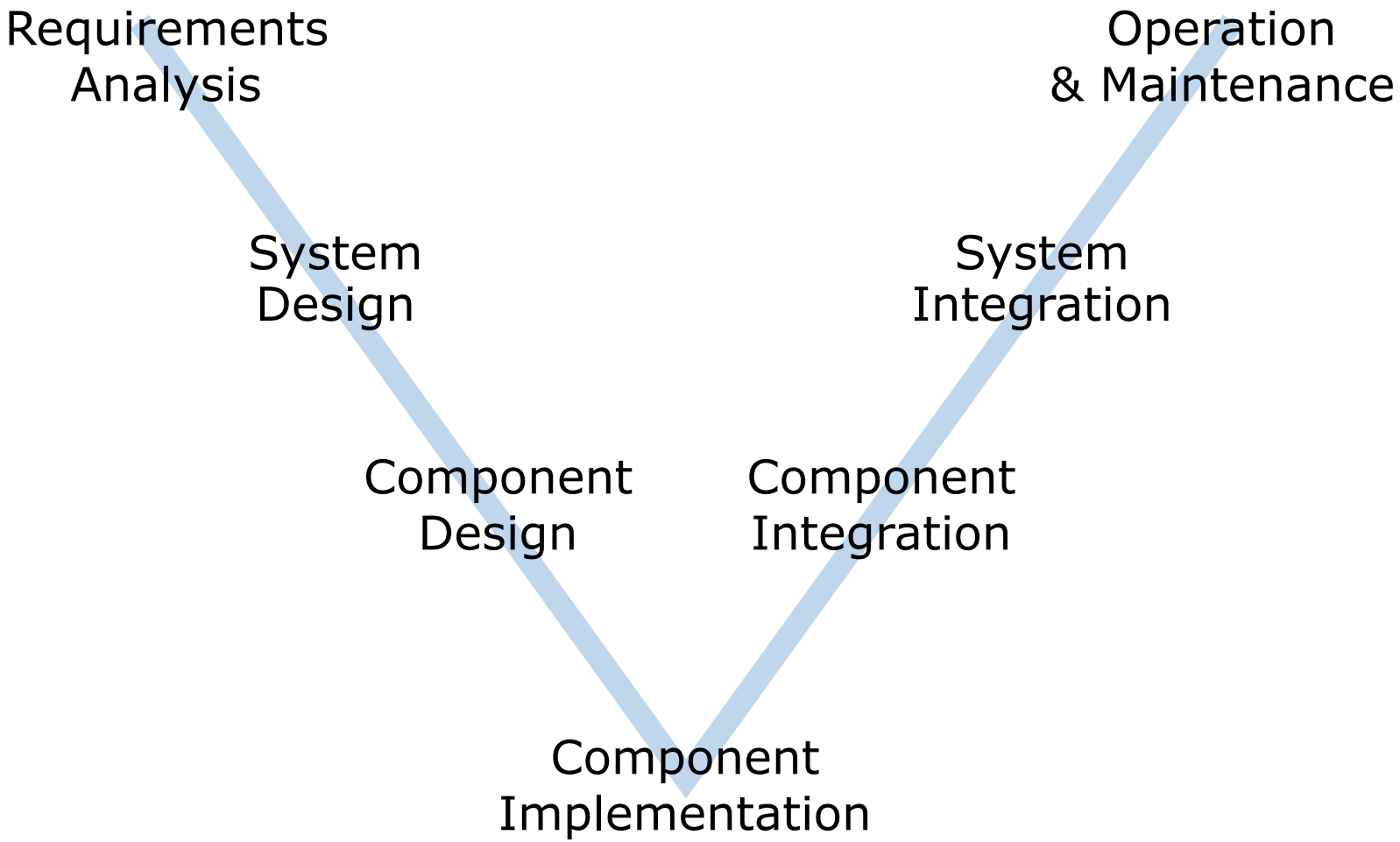
Testing

Overview

Contents

- Testing in theory
- Testing in practice
- Testing in object-oriented software development
- JUnit

Testing in Theory



Testing in Theory

- Types of testing
  - Verification – have we built the software right? (specification)
  - Validation – have we built the right software? (customer)
- Testing levels
  - Unit tests
    - Testing of classes or components
  - Integration test
    - Testing the interaction of components
  - System tests
    - End-to-end testing, use-case testing
- Test procedures
  - Regression testing
  - Production testing
  - Performance testing
  - Acceptance testing, etc.

Testing in Theory

- Testing may not be sufficient for quality assurance:
  - Development process
  - Prototyping
  - Reviews
  - Analysis tools (e.g. code coverage, quality measures, …)

- Early error detection
  - The later an error is detected the more costly it is to mend

Testing in Practice

- Testing tends to be the stepchild of software development
  - Everybody knows how important testing is, but…
  - Testing is not taken seriously
  - Ad-hoc testing
  - Testing is started too late
    - If it is getting tight people are tempted to cut down on testing first
  - Testing of code that has been written days/weeks/months ago is difficult

- Test environment
  - Insufficient quality
  - Insufficient availability / stability
  - Reproducibility

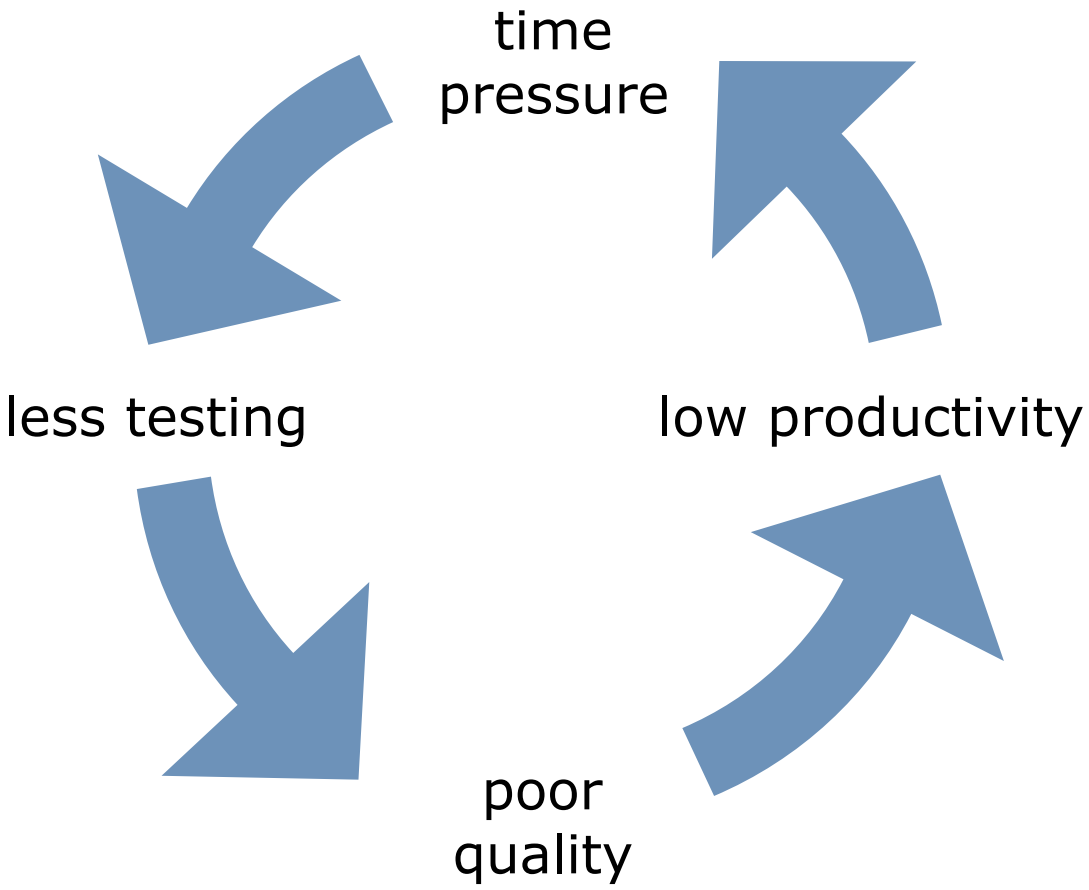
Testing in Practice

- Manual testing
  - Expensive
  - ➜ Testing is not done thoroughly enough
  - Avoiding regression testing

- Code reviews rather than testing the executable
- Integration testing is running late
- Testing is a psychological challenge

➜ Poor software quality

Testing in Practice



Testing – Basics

- Tool support
  - Manual testing is time consuming and does not scale
  - Test automation
- Responsibilities
  - Developers
  - Test team
  - Project management
- Training
  - Testing is challenging
  - Desire to creatively destroy
  - Fathom the limits of a product
- Goal: find errors rather than trying to “prove” the absence of errors!

Testing in Object-Oriented Software Development

- Classes are the starting point for testing
  - Classes are the ideal „unit“ for unit testing
    - What is the unit in procedural programming? Procedure? Module?
  - Each class should be able to test itself
  - Production code and test code are written together
  - Unit may comprise several classes (i.e. component)

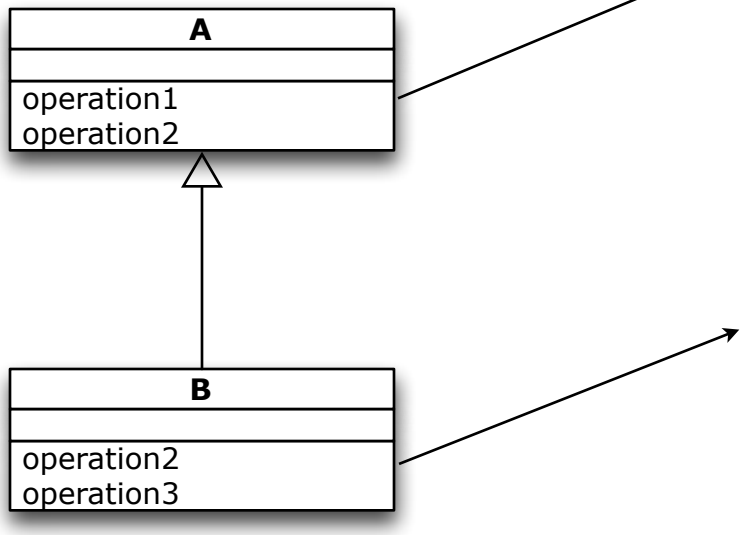
➜ Locality of errors

Testing in Object-Oriented Software Development

- Challenges
  - Polymorphism: combinatorial explosion of testing scenarios
  - Inheritance: changes to a class ➜ clients and subclasses have to be tested

- Goal: testing should be as easy as compiling
  - Automated testing
  - Fast ➜ incentive to run tests after each modification

Testing & Inheritance



Test cases for operation1 and operation2

Test cases for operation2 and operation3

If operation1 calls operation2 in A

➜ write new test cases for operation1!

Unit tests are white box tests

➜ implementation of base classes must be accessible too!

JUnit

- Framework for „codified“ testing
- Foundation for repeatable unit and function tests
  - Automation
  - Infrastructure for assertions
  - Tests can be aggregated in test suites
- User interface
  - Text based
  - IDE integration
- ✷Unit
  - Available for most common programming languages

A Simple Test

- Code to be tested

```
public class Statistics {

    private List<Integer> fElements = new ArrayList<Integer>();

    public int sum() {
        int sum = 0;
        for (int i : fElements) {
            sum = sum + i;
        }
        return sum;
    }

//…
}
```

A Simple Test

- Test code

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

class StatisticsTest {

    @Test
    public void sum() {
        Statistics m = new Statistics();
        m.addValue(3);
        m.addValue(4);
        m.addValue(5);
        assertEquals(3, m.size());
        assertEquals(12, m.sum());
    }
}
```

- Execution

```
java org.junit.runner.JUnitCore StatisticsTest
```

Test Case

- Structure
  - Instance variables for test objects
  - Code for creating test objects
  - Test execution
  - Verification of the results
    - Result: return value of a method or side effect (e.g. object state)
- Test do not have any side effects!
  - Testing sequence does not have any impact
- Fixture
  - @Before and @After are called before/after each test

    ➜ Factorizing of test buildup / teardown
  - @BeforeClass and @AfterClass are called once for each test class

Test Suite

- Test suite aggregates several test classes

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

@RunWith(Suite.class)
@Suite.SuiteClasses({
    StatisticsTest.class,
    OtherTest.class
    })
public class AllStatisticsTests {}
```

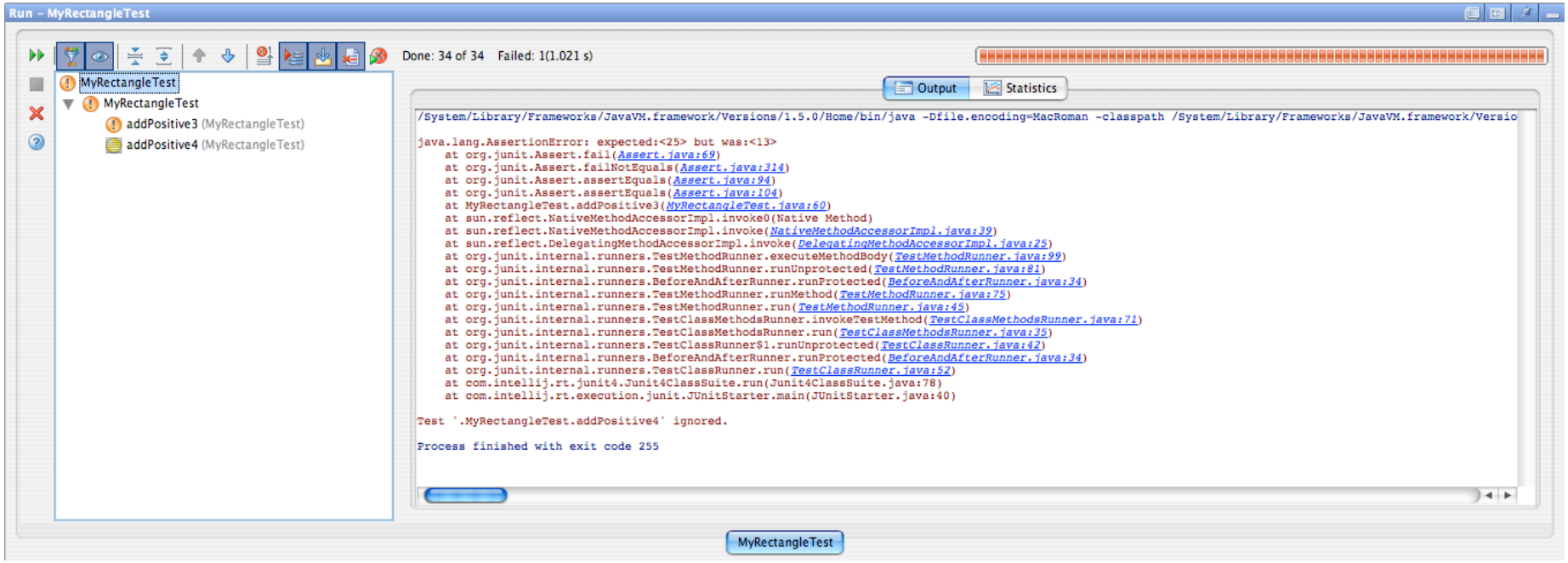
- Usually a test suite for each package
- Suite is a composite (suite of suites)

Additional Testing Infrastructure

- Parameterized tests
  - Repeated execution of a test with different data sets
- Ignoring tests
  - @Ignore ignores a test
  - Ignored tests will be listed as "ignored" in the test result
- Timeout
  - @Test(timeout=10)
  - If the test is not finished after 10 ms it will be terminated and listed as failed
- Exceptions
  - @Test(expected = NullPointerException.class)
  - A test may require a certain exception

JUnit IDE Integration

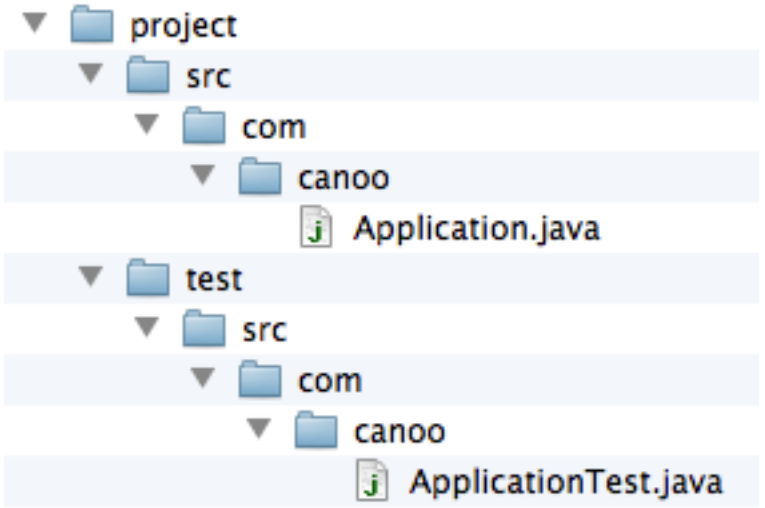
- Example: IntelliJ Idea



Code Layout

- Test code in a separate package
  - Simple layout
  - Example: code: myapp.util, test code: myapp.utiltest
  - Clear separation between production code and test code
  - No access to „package private“ variables and methods

- Test code in the same package
  - Access to „package private“ variables and methods
  - No clear separation between production code and test code
  - Solution:
    - Two separate source directories
    - Both “src”-directories are on the classpath



Testability

- Class interface must support testing
- Isolation of class to be tested
  - Interfaces!
  - Configuration of class (dependency injection, factories)
    - Do not try to subclass for testing purposes!
    - Avoid work in constructor (separate creation from business logic!)
  - Class context (i.e. collaborators) must be replaceable
- Small and focused methods
- Avoid global mutable state
- Avoid deep inheritance hierarchies

Testability

- Information hiding and encapsulation
  - Unit tests are usually white box tests
    - Ignore information hiding
  - Test should not break encapsulation
    - Tests use public API only
    - Otherwise resort to reflection rather than opening class interface
      - E.g. for verifying the object state
- Low testability leads to difficult, expensive and brittle testing
- Test-Driven-Development:
  - Test is written prior to implementing a class
  - ➜ Testability is almost for granted

Application of JUnit

- Unit tests
  - Developers write test and code
  - Important: Each test must be able to run on its own

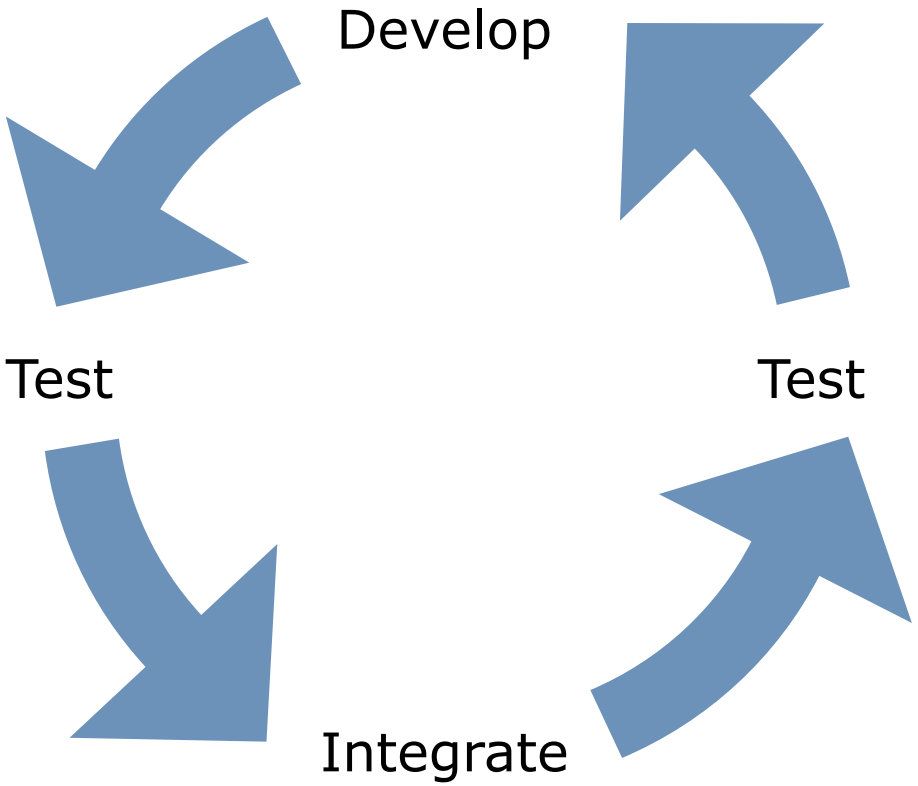
    ➜ avoid dependencies
  - Unit tests are usually white box tests
    - Avoid using the private API, though

- Functional tests
  - Users / developers write functional tests (along use cases)
  - Dependencies cannot always be avoided
  - Functional tests are black box tests

Application of JUnit

- When do we implement a test case for some piece of code?
  - Interface is not yet clear ➜ start with a test case
  - Imlementation is not yet clear ➜ start with a test case
  - Complex interface ➜ executable specification by means of test cases
  - Bug pops up ➜ try to reproduce the bug with a test case
  - Refactoring ➜ “document” the old systems with test cases
- How do we implement a test case?
  - Provoke errors (test border cases)
  - Start with a test which causes an error
    - Failed tests provide confidence!
- When is a test suite complete?
  - No further non-trivial tests are conceivable

Best Practice



Best Practice (Approach)

- Implementing a unit
  - First develop the test cases then implement the unit
  - The unit is completed as soon as all tests pass
- Debugging
  - For each bug popping up a test case is implemented
  - Debug and fix until the test passes
- Integration
  - Aggregate test cases into test suites
  - Automated testing is crucial for integration
- Challenge: large test suites may take quite some time to execute
  - Not all test cases can be run with each minor change
  - Tests should be executed at least once a day:
    - Overnight, during meetings, over lunch, etc.

Testing Context

- Testing needs to decouple from production context
  - Collaborating objects (dependencies) may require resources which are not available in a testing environment (e.g. db, services)
  - Collaborating objects may not provide API to verify behavior
- Abstract coupling is prerequiste for decoupling
  - Use interfaces!
- Subclassing production classes is cumbersome (if possible at all!)
  - Hand-coding mock or stub classes is expensive
- Mock: verify behavior
- Stub: verify state
- Mocking frameworks help to replace dependencies with test classes

Mockito

- Library for creating mocks or stubs with a DSL
- Mock class with Mockito:

```
List mockedList = mock(List.class);

//testing
mockedList.add("one");
mockedList.clear();

//verifying
verify(mockedList).add("one");
verify(mockedList).clear();
```

Mockito

- Stub:

```
List mockedList = mock(LinkedList.class);

//stubbing
when(mockedList.get(0)).thenReturn("first");

//testing
System.out.println(mockedList.get(0));   //prints "first"
System.out.println(mockedList.get(999)); //prints null
```

Functional Testing with JUnit

- Challenge
  - Creating and maintaining a test environment
  - Simulation of the run-time / production environment
    - Databases
    - Services
- Approach
  - Definition of the service / database interfaces
  - Test and production environment implement the interface
  - Context-specific creation of the environment using a factory method / abstract factory / dependency injection

Testing User Interfaces

- Functional tests
  - Test cases are based on use cases
- Challenges
  - Automation
  - Checking results
  - Timing
- Web applications
  - HtmlUnit: models Web document (page, form, table, …)
  - WebTest: models user interaction (XML), based on HtmlUnit
- Swing
  - Jemmy: library for simulating user interaction
  - Alternatives: Pounder, JFCUnit, Abbot, Robot (java.awt.Robot), FEST

Automated Testing – Impact

- Tight integration of testing in software development
- Short-term
  - Code quality (structure and behavior)
  - Confidence in code
  - Productivity (significantly less debugging)
- Long-term
  - System is (economically) viable
  - Automated testing enables radical changes / refactorings
- How many errors are permitted?
  - Unit tests: none
  - Functional tests: depends on effort and priorities

Summary

- Testing is a integrated part of software development
- Testing has to start as early as possible
- Automated testing
- JUnit is supportive rather than interfering
- Continuous integration is crucial

Literature

- JUnit: http://www.junit.org
- HtmlUnit: http://htmlunit.sourceforge.net/
- WebTest: http://webtest.canoo.com
- Jemmy: http://jemmy.netbeans.org/
- E. Kit: Software Testing in the Real World
- K. Beck: Test-Driven Development