Unit testing and Java

Zeger Hendrikse

Cas Stigter

Importance of testing

Testing should occur throughout the development lifecycle. Testing should never be an afterthought. Integrating testing into the development process brings many benefits [5].

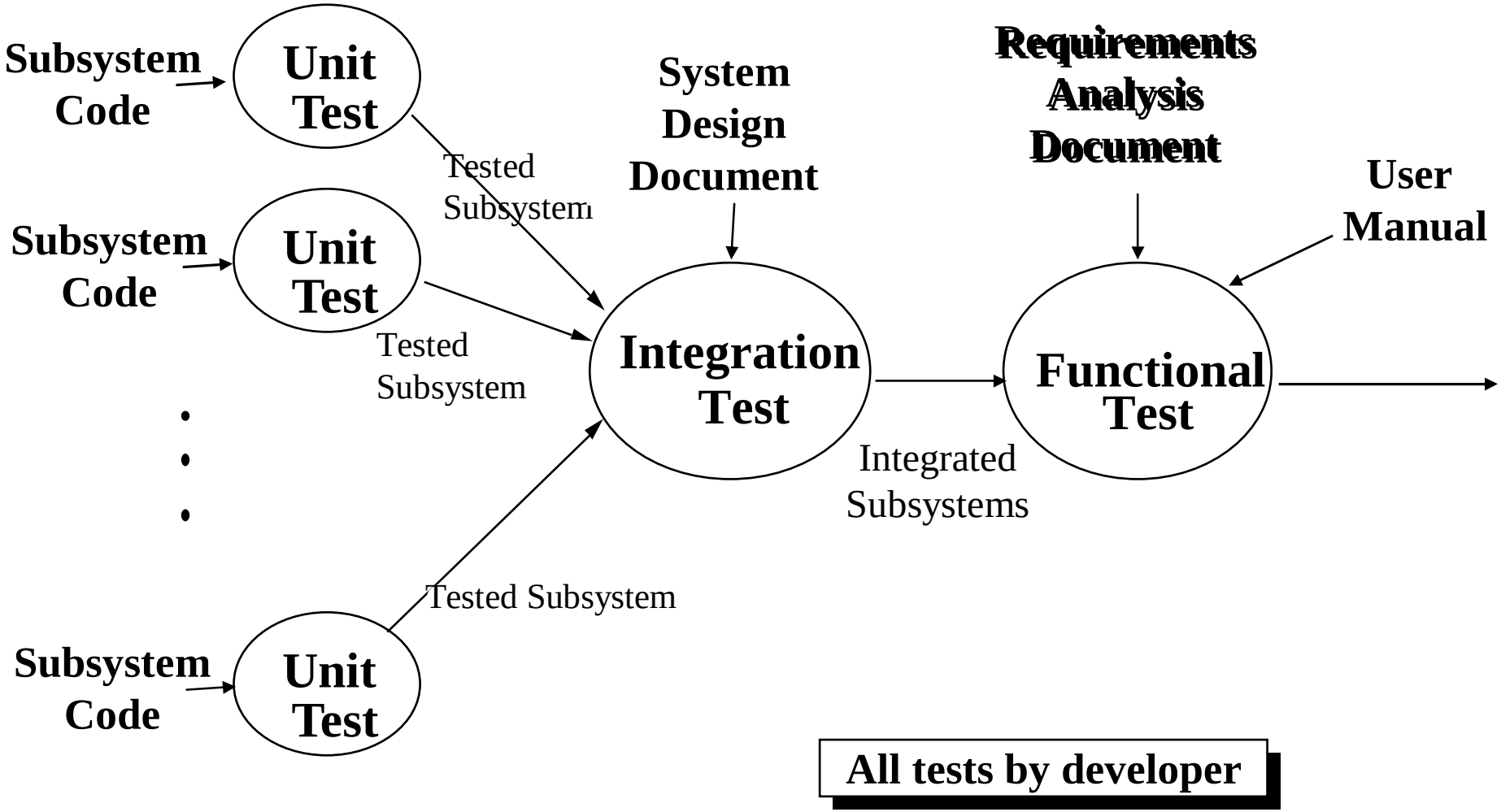
Content

- Part I
  - General motivation & introduction (J)unit testing
  - Testing simple Java classes + demo
  - Integration with IDEs and Ant
- Part II
  - …and beyond
    - In-container testing
    - HttpUnit, DBUnit, Cactus, JUnitPerf, JMeter
- Part III
  - Book reviews demo application
- Part IV
  - Workshop

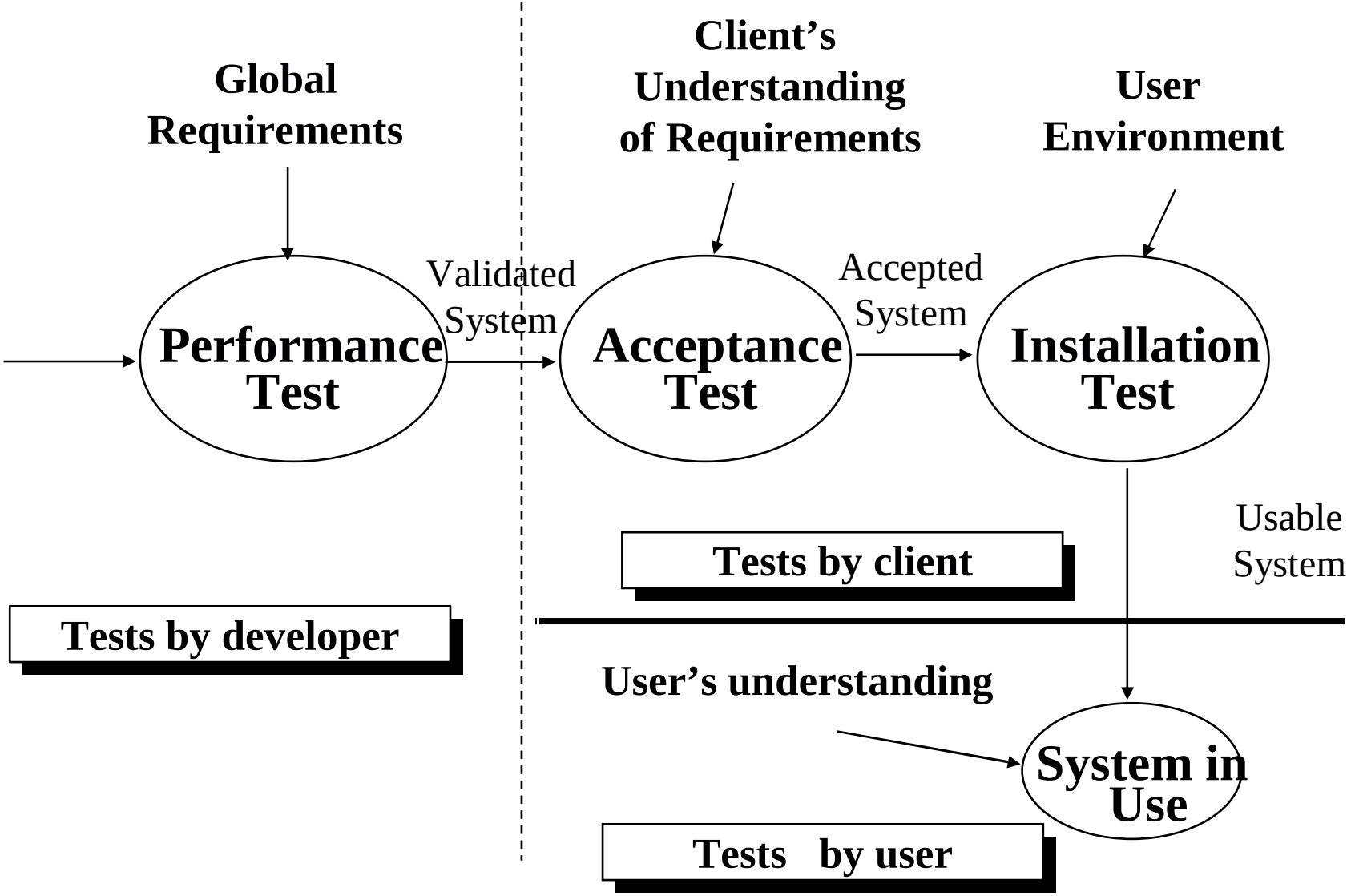
Introduction

PART I

Testing activities (1/2)



Testing activities (2/2)



Unit tests

- **Unit test**
  - test behaviour of single independent unit of work
  - `System.out.println()` is not enough!
- **Unit test** best practices
  - Each test must run independently of all other unit tests
  - Errors must be detected and reported test by test
  - It must be easy to define which unit test will run (automation)
- **JUnit** design goals. The framework must help us
  - write useful tests
  - create tests that retain their value over time
  - Lower the cost of writing tests by reusing code

XP (not exhaustive)

- Simplicity
  - advocates doing “the simplest thing that could possibly work”
  - avoid wasted effort (YAGNI=You Aren’t Going To Need It)
- TDD (Test Driven Development)
  - *Develop tests before actual code*!
  - If bug is found → create test case → reproduce it → fix it
  - **Code that hasn’t been tested, doesn’t exist**
- Incrementally develop (and refactor!) your code
  - IMHO only viable by above feature
- Pair programming
- …

Unit testing and its relation to logging

- Offers “persistent debugging” (vs. session in debuggers)
- “…unit tests are valuable in indicating *what* may be wrong with an object, but won’t necessarily indicate *where* the problem is...” [5]
- “Code isn’t ready for production unless it is capable of generating log messages and its log output can easily be configured” [5] → As important as unit tests.
- No `System.out`: console output cannot be configured!
  - Switched off
  - Redirected
  - Performance degradation

JUnit structure

- `TestCase`
  - Extends JUnit `TestCase` class
  - Contains one or more `testXXX()` methods
- `TestSuite`
  - Group of tests
- `TestRunner`
  - Launcher of a suite of tests

JUnit example (1/3)

```
Package nl.amis.demo.calculator;

public class Calculator
{
  public double add(double x,
    double y)
  {
    return x + y;
  }
}
```

1. Useful for separating test and app. Code
2. We will test this method

JUnit example (2/3)

```
package nl.amis.demo.calculator;

import junit.framework.TestCase;

public class CalculatorTest
   extends TestCase
{
  public void testAdd()
  {
    Calculator calculator =
      new Calculator();
    double result =
      calculator.add(12, 34);
    assertEquals(46, result, 0);
  }
}
```

1. Tests in different folder, but in same package!
2. Import JUnit stuff…
3. Extend from TestCase
4. Method must be called testXXX() (reflection will find it)
5. Create an instance of the class to test
6. Generate test result…
7. … and check result.

JUnit example (3/3)

- Suite
  - We need a container to contain all the tests/test cases
  - A new instance of the `TestCase` class for each `testXXX()` method
  - TestRunner automatically creates a `TestSuite` if you don't provide one
    - `TestSuite` scans your class for `testXXX()` methods, so default is:

```
public static Test suite()
{
    return new TestSuite(TestCalculator.class);
}
```

Depending on available time:

- Demo setting up this project in JDeveloper

More realistic example

```
public static String[]
    commaDelimitedListToStringArray(String s)
```

- Testing must encompass the following conditions
  - Ordinary inputs (words and characters separated by commas)
  - A null string, we expect an empty array on null input
  - A single string (output single element array containing string)
- We simplify the test code by
  - Including a private method that does the verification

Test practices

JUnit best practices:

1. Write tests to interfaces
2. Don’t bother testing Java bean properties
3. Maximize test coverage
4. Don’t rely on the ordering of test cases
5. Avoid side effects
6. Read test data from the `classpath`, not the file system
7. Avoid code duplication in test cases
8. TDD: test should fail first, to assure that the test is ok!

The 7 core classes of the JUnit test framework

1. `Assert`
   - `AssertEquals()`, `AssertNotNull()`, ...
2. `TestResult`
   - Contains collection of errors and or failures
3. *`Test`*
   - Can be run and passed to `TestResult`
4. *`TestListener`*
   - Invoked of events during testing
5. `TestCase`
   - Defines an environment (fixture, `setup()`, `tearDown()`) for running tests
6. `TestSuite`
   - Runs a collection of `TestCase`'s
7. `BaseTestRunner`
   - Superclass for all test runners

Testing for Exceptions

- To test if the right handling takes place when e.g. the database is down
- Example: retrieve an article from the DB that does not exist:

```
public void testGetFinderErrorResponse()
   {
       // Ask for an article that isn't there and see if
       // we get a FinderException
       try
       {
           org.bibtexml.session.ArticleFacade bean = getHome().create();
           ArticleDTO data = bean.readArticle(PUB_ID_3);
           // this should have failed and we should be in the catch clause
           fail("The article should not have been found here!");
       }
       catch (Exception e)
       {
           // We expect a FinderException here
           assertEquals(FinderException.class, e.getClass());
       }
   }
```

Definitions: fixtures, stubs & mock objects

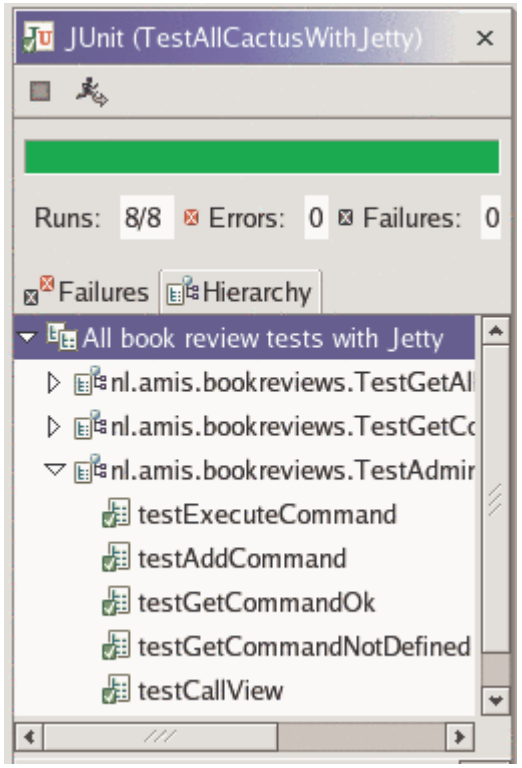
- Fixture: “The set of common resources or data that you need to run one or more tests” [1]
- Example: set up a database connection before testing

- Stub: “portion of code that is inserted at runtime in place of the real code…to replace a complex behavior with a simpler one that allows independent testing of some portion of the real code” [1]
- Mock object: “is created to stand in for an object that your code will be collaborating with.” [1]

JUnit and Ant

- Make unit testing an integral part of your project: Ant!

```
<junit printsummary="yes" haltonfailure="yes">
  <classpath refid="master-classpath" />
  <formatter type="plain" />
  <!-- JUnit classes are XXXTestSuite.java by convention -->
  <batchtest fork="yes"
             todir="${reports.dir}/name-${DSTAMP}-${TSTAMP}">
    <fileset dir="${name-test.dir}">
      <include name="**/*TestSuite.java" />
     </fileset>
  </batchtest>
</junit>
```



Slogan Unit tests:

Keep the bar green to keep the code clean!

Design patterns

Intermezzo I

Intermezzo I: JUnit & Design Patterns (1/2)

- Composite pattern
  - "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly" [GoF].
  - JUnit's `Test` interface to run a test *or* a suite of tests *or* a suite of a suite of tests.
- Command pattern
  - "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations" [GoF].
  - JUnit's `Test` interface provides a command `run()` method for both the `TestSuite` and `TestCase`.

Intermezzo I: JUnit & Design Patterns (2/2)

- Collecting parameter
  - Collect results over different methods by passing an object
  - JUnit's `TestResult` class is passed as collecting parameter object.
- Observer (in JUnit, but we skip `TestListener` here)
  - "Define a one to many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically" [K. Beck, *Smalltalk Best Practice Patterns*]
  - The `TestRunner` registering as a `TestListener` with the `TestResult` is an example of the observer design pattern.

Stubs, mocks and in-container testing

PART II

In-container testing (vs. mock objects)

Mock objects:

- Do *not* require a container
- *Pure* unit tests
- Delivers real value if mocks aren't too complex
- "improves domain code by preserving encapsulation, reducing global dependencies and clarifying the interaction between classes"
- Difficulty: "...the discovery of values and structures for parameters that are passed into the domain code"

In-container testing:

- Interactions *with* container too!
  - So less "unit" than mock objects…
- Integration + deployment too!
  - So less "unit" than mock objects…
- Still: Cactus tests individual components as opposed to HttpUnit (=acceptance testing).

Testing business objects

- **Testing business objects implemented without using EJB**
  - Simply use JUnit, container services may be mocked
- **Testing EJBs**
  - Write a test that is a remote client of the EJB container (suitable for EJBs with remote interfaces)
  - Write and deploy a test that executes within the application server (requires additional test framework (e.g. cactus) + complicates deployment)
  - Work with stub objects (only feasible when EJBs have simple requirements of the EJB container

Testing database interaction

- Using mock objects
- Using JDBC helper classes (to set/verify/undo the results in the database)
- Using DBunit, which we'll discuss later in more detail

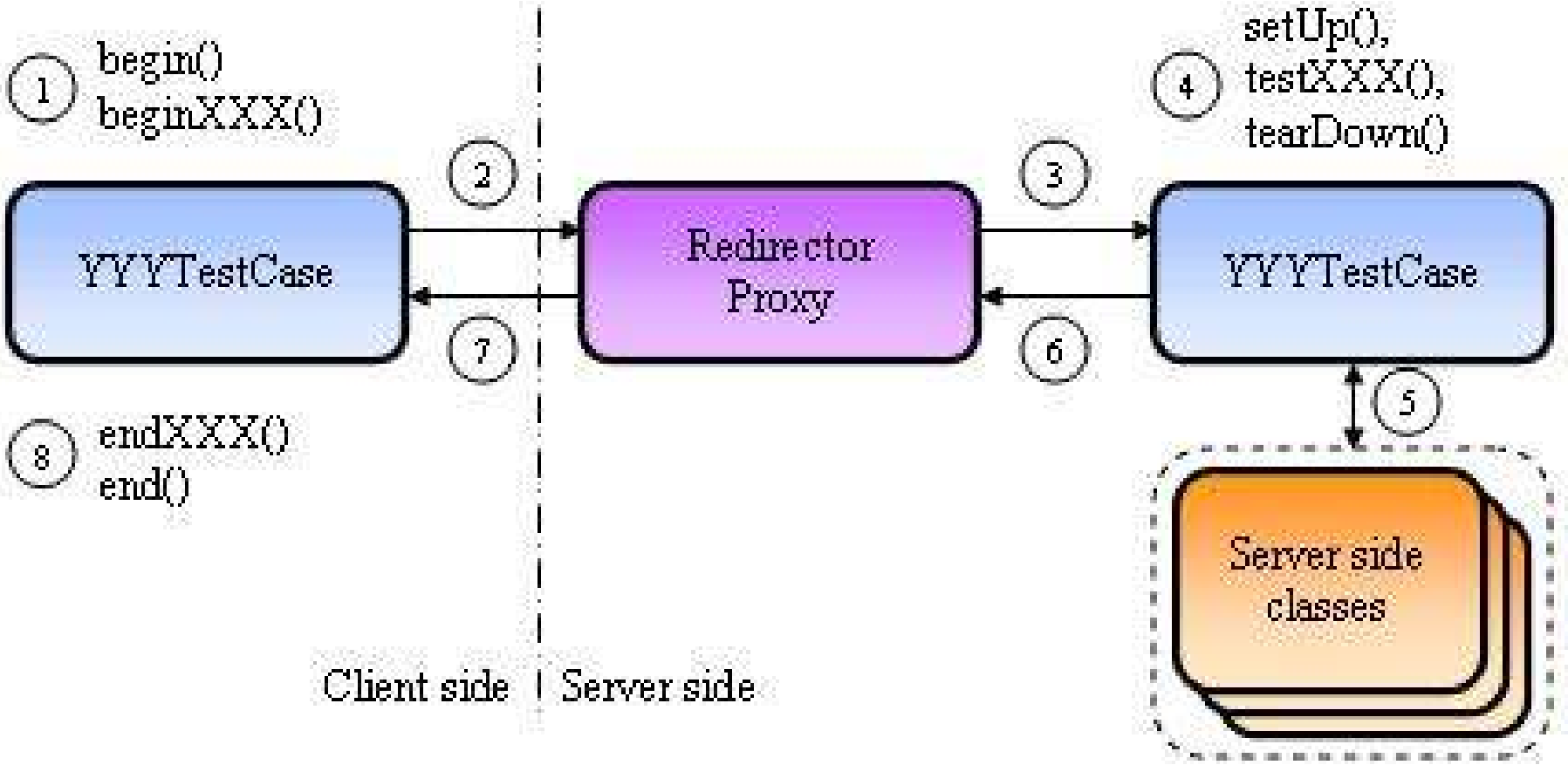
Testing web interfaces

- **Testing web-tier components**
  - Testing outside the container with Servlet Unit
  - In-container testing with Cactus
- **Acceptance testing web interfaces**
  - HttpUnit allows us to write acceptance tests that run outside of the container
- **Load testing web interfaces**
  - Microsoft's Web Application Stress Tool (WAS), easy to use and freely available
  - JMeter

In-container testing: Cactus (1/4)

- Cactus is an open source test framework, based on JUnit, that allows in-container testing of EJBs, servlets, JSP pages and servlet filters.
- Originally J2EEUnit, but legal issues with Sun's J2EE…
- Generates implicit objects available in a web server
  - e.g. `HttpServletRequest`, `PageContext`, `FilterChain`
- **Transparently** mimics client & server in one go

In-container testing: Cactus (2/4)



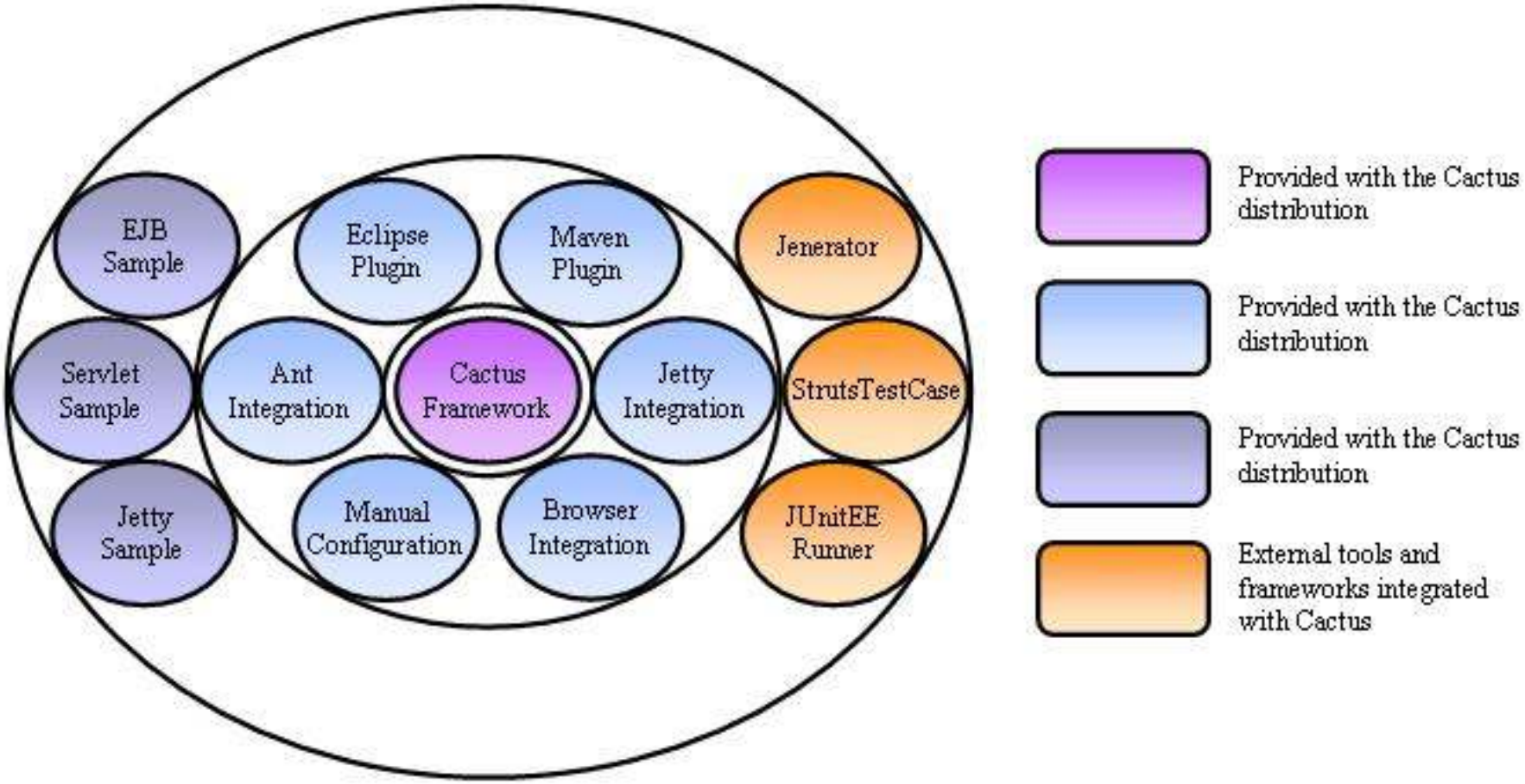
- `YYYTestCase=(ServletTestCase | FilterTestCase | JspTestCase)`
- XXX=name of the test case. Each `YYYTestCase` class contains several test cases.

© http://jakarta.apache.org/cactus/

In-container testing: Cactus (3/4)

1. beginXXX(), pass information to the redirector
2. YYYTestCase opens the redirector connection
3. creation of server-side YYYTestCase instance
   - assignment of `HttpServletRequest` etc. to instance
   - executes test, results in `ServletConfig` servlet object
4. call setUp(), testXXX and tearDown() of YYYTestCase
5. client-side executes endXXX()
   - you can assert cookies, HTTP header and/or HTTP content
6. Gathering the test result from `ServletConfig` and pass it through to the JUnit test console/GUI.

In-container testing: Cactus (4/4)



The Cactus Ecosystem

© http://jakarta.apache.org/cactus/

Cactus set-up

Info from [5]

1. Ensure that the classpath is set-up correctly!
2. Edit the web application's `web.xml` to define the Cactus "servlet redirector" servlet
   - routes requests from the remote tests to the server-side test instances
3. Include the test classes in the `WAR`.
4. Configure the Cactus client
   - All necessary libraries for both client and server

Cactus/Eclipse integration

- "The Eclipse plugin is a work in progress. In the past it was working, but since we moved to the new Ant integration it has been broken […] in the meantime, we have removed the plugin download."
- Solved by using Cactus/Jetty integration (Jetty plug-in)
  - Jetty is embedded Web AS
    - supports servlets, JSPs
    - Used in stubbing the web server's resources
  - Advantages:
    - Very fast,
    - IDE independent
    - Can set breakpoints for debugging
  - Wrap tests in `JettyTestSetup` class provided by Cactus

DBUnit

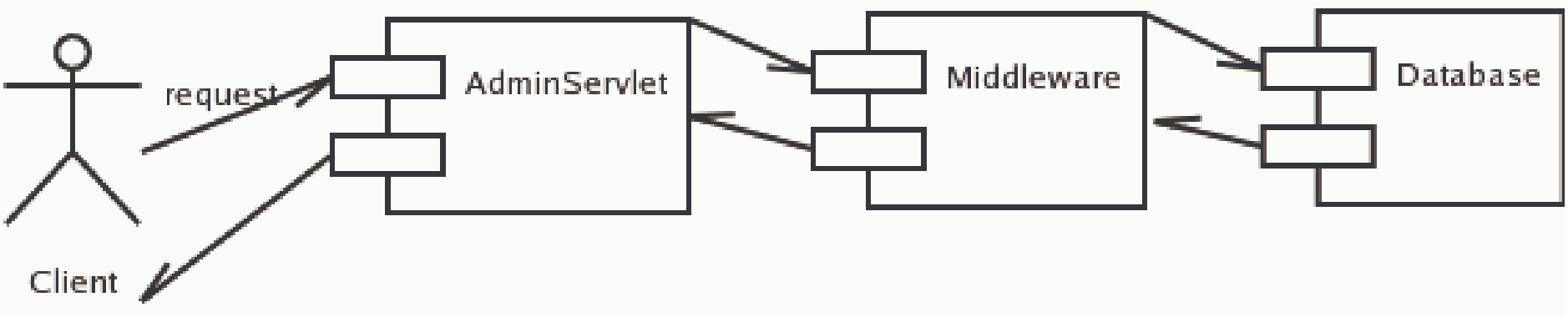
- DB set-up for testing can be done with Ant but:
  - DbUnit adds in-container testing if DB is accessed by container, e.g. J2EE container
  - Verification of contents of DB after tests
  - The database is reset with the contents of the XML file after every test
- DB set-up is defined in flat XML file

Book reviews demo application

PART III

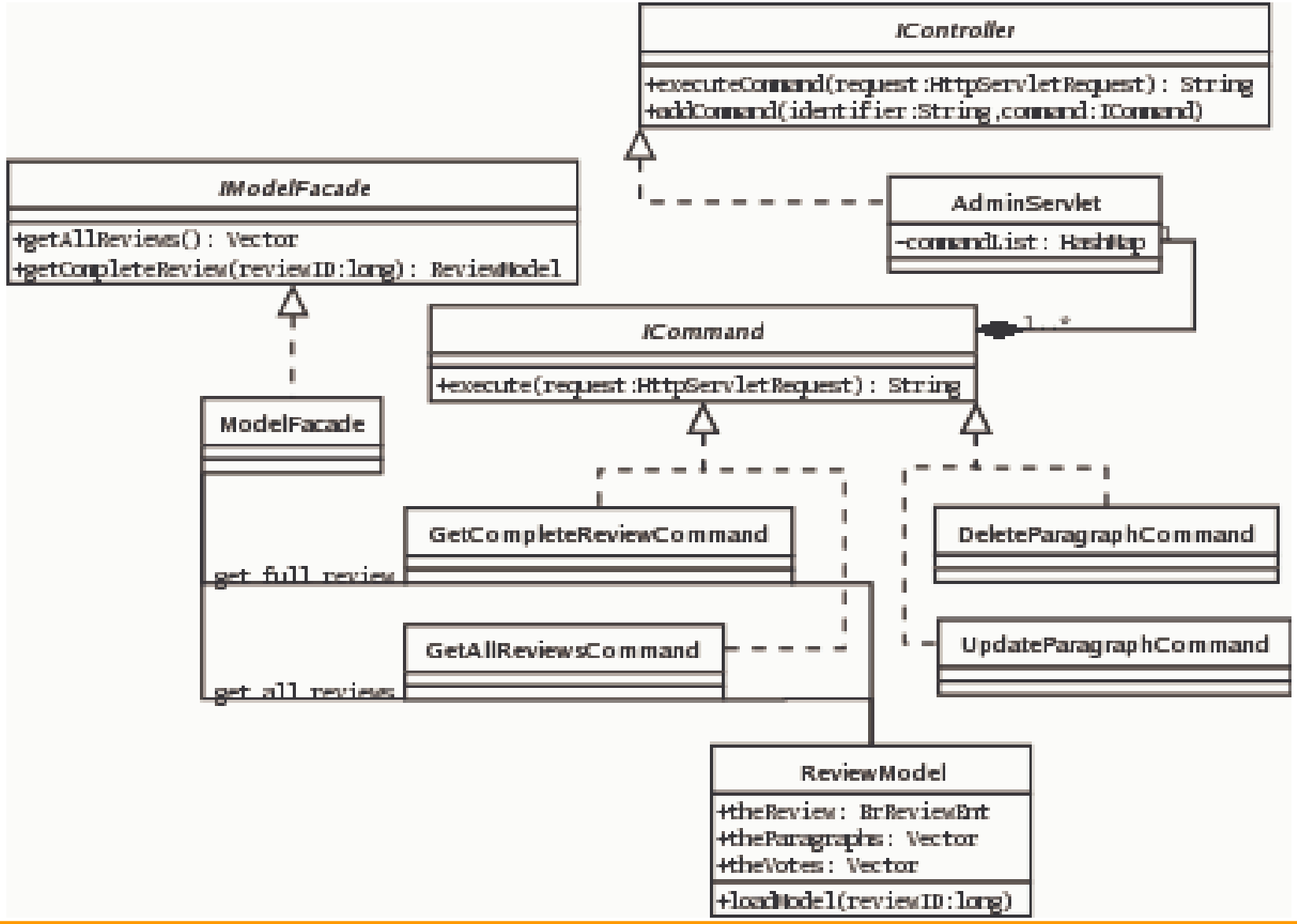
(J)Unit testing demo app. (1/3)

- BookReview web application
  - Book review data model stored in Oracle DB (Ludo)



- Get all reviews or one complete review
- Read a review and/or modify it by updating/adding a paragraph
- Built with servlets, JSPs, Java classes and Oracle data base

(J)Unit testing demo app. (2/3)



(J)Unit testing demo app. (3/3)

- Shows (J)Unit testing in real example
  - Application of in container testing (Cactus) & mock objects
  - Shows use of
    - JUnit
    - Cactus
    - HttpUnit (to validate output of JSPs)
    - DBunit

Design patterns

Intermezzo II

Intermezzo II: Design Patterns revisited (1/2)

Design patterns used in book review demo app:

- Strategy (see Intermezzo I)
  - Used here for request handling
  - **Key** design pattern in Struts and Java Server Faces (JSF) frameworks
- Inversion of control
  - Handler object registers with the controller for an event
  - In case of an event, the callback of the appropriate handler is invoked
  - Lets one properly manage event life cycle and plug in of custom handlers
  - IMPORTANT FOR UNIT TEST CASES

Intermezzo II: Design Patterns revisited (2/2)

- MVC = Model (1) + View (2) + Controller (3)
  - **Key** design pattern in Struts and JSF frameworks
  1. Model is the data retrieved from the DB via façade interface
  2. View is given to JSPs
     1. Code still in JSPs, should be improved by using custom tag library
  3. Controller is de AdminServlet
- Façade
  - Offers a simple interface to (complex interacting) subsystems
  - Here: retrieve all book reviews and/or a complete review

Conclusions and references

Conclusions

- “Testing should occur throughout the software life cycle. testing should be a core activity of software development” [5]
- Unit testing is usually the most important type of testing
- Not easily “unit-testable” may be a strong argument against that particular technology, e.g. EJBs
- TDD is a core feature of XP
- Unit testing is fun!

Workshop

PART IV

References

1) *"JUnit in Action"*, V. Massol
2) "*Java tools for eXtreme Programming*", R. Hightower, N. Lesiecki
3) http://www.junit.org/
4) "*Eclipse in Action*", D. Gallardo et al.
5) "*Expert one-on-one J2EE design and Development*" by Rod Johnson, Wrox 2002.