CSE 70: Unit Testing and Mock Objects
JUnit and jMock

Ingolf Krueger
Learning Goals for Today
Learning Goals

• Understand the roles and limitations of testing in the software development process

• Be able to distinguish between unit, integration, system and acceptance testing

• Understand the value proposition of test-driven development

• Be able to design basic unit tests

• Be able to implement basic unit tests in jUnit

• Understand the need for Mock Objects

• Be able to comprehend basic unit tests with Mock Objects

• Understand the benefits and limitations of Mock Objects
Focus: Testing in Iterative and Agile Development
Test Process Maturity (after Beizer)

L4  “mental discipline for higher quality software”

L3  “reduce risk of using the software”

L2  “show that the software doesn’t work”

L1  “show that the software works”

L0  Debugging
Evolutionary Development

- Typical for smaller systems/projects
- Increasingly being used also for larger systems
Agile methods

- Rely on team knowledge instead of on documentation
- Iterative: several short cycles
- Incremental: development and delivery in several iterations
- Self organizing: teams decide on the best way to operate
- Emergence: processes, principles and work structure emerge during the project.
Testing in XP
• **Fine-scale feedback**
  − Pair programming
  − Planning game
    (requirements captured on index cards, prioritized)
  − **Test-driven development**
  − Whole team
• …
Test everything that can possibly break!
Testing Activities (1/2)

• **Unit Testing**
  - gain confidence that individual units of source code work as expected
  - in OO: method/intra-class-level tests

• **Module Testing:**
  - often a synonym for Unit testing
  - in OO: test class as a whole

• **Integration Testing:**
  - gain confidence that multiple modules/subsystems work correctly together
  - in OO: interplay between classes/components
Testing Activities (2/2)

- **System Testing:**
  - gain confidence that *all* modules/subsystems work correctly together when assembled into overall system
  - in OO: tests of the built/deployed system

- **Acceptance Testing**
  - gain confidence that the overall system meets its requirements
  - in OO: tests against user requirements

- **Others:**
  - Usability testing
Unit Testing and Test-Driven Development

- Testing **at the method-level** in OO (mostly *intra-class*)
- Isolate fine-grained functionality
- Give code-based “specification” for what the method is supposed to do

**Benefits:**
- Defines contract
- Facilitates regression testing
- Supports refactoring
- Increases confidence in code/refactorings
- Helps identify problems early
- Can be expanded into integration testing
- Can be highly automated
Unit Testing and Test-Driven Development

• Limitations:
  – As with all testing approaches: can only show presence of bugs, not their absence (cf. E. W. Dijkstra)
  – Focus on unit-level
  – Code coverage may require vast number of tests – effectiveness?
  – How do you know when to stop writing tests?
  – How to test non-functional/quality aspects?
  – Are the tests themselves correct?
  – Rigor and discipline necessary in writing and executing tests, as well as in immediately addressing test failures
  – Need to run fast to allow automation and frequent test execution
Test-Driven Development: Basic Workflow

1. Add a Test
2. Run all Tests
3. Write Code
4. Run all Tests
5. Refactor

see new test fail

automated,
see tests succeed
How do you write Tests in Java?

jUnit
JUnit4 – Test Suites, Fixtures, Test Cases

Test Fixture Setup

@BeforeClass

Test Fixture Teardown

@AfterClass

@Before

@Test t1

@After

@Before

@Test t2

@After

...

@Before

@Test t3

@After

Test Case

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

public class StringFormatterTest {
    StringFormatter formatter;

    @BeforeClass
    public static void initialSetup() {
        System.out.println("Testing Starts!");
    }

    @Before
    public void setup() {
        formatter = new StringFormatter();
    }
    ...
}
@Test
dpublic void string() {
   String in = "Hello!";
   String out = "H*e*l*l*o*!";
   assertEquals(out, formatter.formatString(in));
}

@After
dpublic void teardown() {
   formatter = null;
}

@AfterClass
dpublic static void finalTeardown() {
   System.out.println("Testing Ends!");
}
}
Example: jUnit 4

• Test Suite:
  – A class containing a number of test cases
  – manages test fixture

• Test Case:
  – jUnit4 treats any method annotated with @Test as a test case.
  – Test methods must not have parameters.
  – To run each test (method), jUnit creates a new instance of the class defining the test case and calls the test method: tests are isolated from one another
Example: jUnit 4

• Assertions
  – A jUnit test triggers the object under test and then makes assertions about the results
  – Examples:
    • assertEquals(a, b)
    • assertTrue(a)
    • assertFalse(a)
    • …

• Test Fixtures
  – Defined state at the start of a test
  – Goal: Repeatable tests
  – Setup/Teardown with
    • @Before/@After
    • @BeforeClass/@AfterClass
Your Turn!
Will this code pass the test? – Team up for 3 minutes!

```java
public String formatString(String in) {  
    StringBuffer out = new StringBuffer(); 
    for(int i = 0; i < in.length(); i++) {  
        out.append(in.charAt(i));  
        out.append(CharConstantHolder.STAR);  
    } 
    return out.toString();
}

@Test
public void string() {  
    String in = "Hello!";
    String out = "H*e*l*l*o*!";

    assertEquals(out, formatter.formatString(in));
}
```
Recap!
package wednesday;

import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;

public class StringFormatterTest {

    ...
}
Example: StringFormatterTest

public class StringFormatterTest {

    StringFormatter formatter;

    @Before
    public void setup() {
        formatter = new StringFormatter();
    }
}
Example: StringFormatterTest

```java
@After
public void teardown() {
    formatter = null;
}

@Test
public void emptyString() {
    assertEquals("", formatter.formatString(""));
}
```

StringFormatterTest.java
Example: StringFormatterTest

```java
@Test
public void simpleSingleCharString() {
    assertEquals("A", formatter.formatString("A");
}
```
@Test
public void singleCharString() {
    Random random = new Random(Calendar.getInstance().getTimeInMillis());
    char randomChar = (char)(Math.abs(random.nextInt())%255);
    StringBuffer sb = new StringBuffer();
    sb.append(randomChar);
    assertEquals(sb.toString(),
                 formatter.formatString(sb.toString()));
}
Example: StringFormatterTest

```java
StringFormatterTest.java

import com.example.StringFormatter;

@Test
public void string() {
    String in = "Hello!";
    String out = "H*e*l*l*o*!";

    assertEquals(out, formatter.formatString(in));
}
```
## Useful Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@BeforeClass</code></td>
<td>Annotates method to be executed before an entire test suite</td>
</tr>
<tr>
<td><code>@Before</code></td>
<td>Annotates method to be executed before every test case</td>
</tr>
<tr>
<td><code>@Test</code></td>
<td>Annotates method to be a test case</td>
</tr>
<tr>
<td><code>@Test(expected=SomeException.class)</code></td>
<td>Annotates method to be a test case in which you expect exception <code>SomeException</code> to be thrown</td>
</tr>
<tr>
<td><code>@Test(timeout=135)</code></td>
<td>Annotates method to be a test case that will finish before timeout of 135 ms occurs</td>
</tr>
</tbody>
</table>
## Useful Annotations/Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertEquals(Object expected, Object actual)</code></td>
<td>Asserts that two objects are equal</td>
</tr>
<tr>
<td><code>assertSame(Object expected, Object actual)</code></td>
<td>Asserts that two object references refer to the same object</td>
</tr>
<tr>
<td><code>assertNull(Object object)</code></td>
<td>Asserts that object reference is null</td>
</tr>
<tr>
<td><code>assertTrue(Boolean condition)</code></td>
<td>Asserts that condition is true</td>
</tr>
<tr>
<td><code>assertFalse(Boolean condition)</code></td>
<td>Asserts that condition is false</td>
</tr>
<tr>
<td><code>fail(String message)</code></td>
<td>Fail test with message</td>
</tr>
</tbody>
</table>

© Ingolf Krueger, 2010
Testing Terminology
Testing Approaches

• Black-box testing = functional
  – source code is unavailable
  – internal state cannot be directly observed
  – based on requirements, design specs...

• White-box testing = structural
  – source code can be analyzed and the program can be set into arbitrary states
  – Syntactic analysis of algorithms, undocumented features, “tricks”

• Gray box = mixture of two
Testing Coverage Quantification

- **Statement (block)** – for every statement there exist at least one test leading to the execution of that statement

- **Decision (branch)**
  - for every decision there exists at least one test that evaluates it to “true” and another one for “false”
  - short circuit – for language specific optimizations (e.g., a & b & c, when a is false then don’t evaluate b and c)

- **Data definition** – there is at least one test covering the path from the definition to the usage of all entities
Testing Tips
Be conscious about what to test, and what approach to use – limit tests to everything that could possibly break your system.
Supply many unit tests for complex code.

Keep code simple.
Classify I/O Data Types and Test Each Class
Understand and test for corner cases.
Test for both success and failure.
Challenges for Unit Testing and TDD
Challenges for Unit Testing and TDD

• What if the program spreads over multiple processes or nodes in a network?
• Can we move a continuously running server into a consistent start state for the test?
• What if we don’t have access to some of the components we must test against?
• What if the setup/teardown costs are large?
  – Examples:
    • Databases
    • Web Services
    • Mass file operations
  – Slows down the test process/software development rhythm
  – Reduces attractiveness of “always-on” testing
Example: Chat System

public interface ChatClient {
    public void notify(String m);
}

public interface ChatServer {
    public void sign_on(String client_id, ChatClient client);
    public void publish(String message);
    public boolean isPresent(String client_id);
}
ChatClient and ChatServer

server: ChatServer

client1: ChatClient
client1 Signs Up

server: ChatServer

1. sign_up

client1: ChatClient
client2 Signs Up

server:
ChatServer

2. sign_up

client1:
ChatClient

client2:
ChatClient
Message Gets Published

3. publish

server:
ChatServer

client1:
ChatClient

client2:
ChatClient
Clients Get Notified

server: ChatServer

4. notify

client1: ChatClient

5. notify

client2: ChatClient
Scenario we want to test: two clients sign up, a message gets published, both clients are notified about that message.
Now Let’s Test the ChatServer: First Attempt

ChatServerTest.java

```java
@Test
public void publishMessage() {
    ChatClient client1 = new ChatClient();
    ChatClient client2 = new ChatClient();

    ChatServer server = new ChatServer();

    String message = "Hooray";

    server.sign_on("ikrueger", client1);
    server.sign_on("mamenari", client2);
    ...
}
```

Now the test is set up!
Now Let’s Test the ChatServer: First Attempt

// trigger events on target object
server.publish(message);

// Now find out that the clients have been called!
// But how?

assertTrue(client1.hasReceived(message));

// what's wrong with this???
// remember what unit we are testing?
...

ChatServerTest.java
How can we keep tests localized and focused? 
Enter: Mock Objects
Mock Objects

Object under test, with implementation

server: ChatServer

client1: ChatClient

client2: ChatClient

Mock objects, only interface known
Testing with Mock Objects

set up test fixture, called “Mockery”

server: ChatServer

specify expectations about interplay

hide implementation details of Mock Objects
Mock Objects Basic Template

1. Create any required mock objects

2. Create any real objects needed for the test, including the target object

3. Specify the “interaction protocol” between the target and the mock objects

4. Trigger the event(s) on the target object

5. Assert any resulting values and that all the expected calls have been made

Now Let’s Really Test the ChatServer with Mockery

```java
import static org.junit.Assert.*;
import org.jmock.integration.junit4.*;
import org.junit.runner.*;
import org.junit.*;
import org.jmock.Mockery;
import org.jmock.Expectations;

@RunWith(JMock.class)
public class ChatServerMockTest {
    private final Mockery context = new JUnit4Mockery();
    ...
```

ChatServerTest.java
Now Let’s Really Test the ChatServer with Mockery

\[
\begin{align*}
\text{@Test} \\
\text{public void publishMessage()} { \\
\hspace{1em} \text{final ChatClient client1 = context.mock} \\
\hspace{2em} \text{(ChatClient.class);} \\
\hspace{1em} \text{final ChatClient client2 = context.mock} \\
\hspace{2em} \text{(ChatClient.class,} \\
\hspace{3em} \text{"client2")}; \\
\hspace{1em} \text{// create real objects (target is the server)} \\
\hspace{1em} \text{ChatServerImp server = new ChatServerImp();} \\
\hspace{1em} \text{final String message = "Hooray";} \\
\hspace{1em} \text{server.sign_on("ikrueger", client1);} \\
\hspace{1em} \text{server.sign_on("mamenari", client2);} \\
\hspace{1em} ... \\
\end{align*}
\]
Now Let’s Really Test the ChatServer with Mockery

```java
// set expectations
context.checking(new Expectations() {
    one (client1).notify(message);
    one (client2).notify(message);
});
```

ChatServerTest.java
Now Let’s Really Test the ChatServer with Mockery

```java
// trigger events on target object

server.publish(message);

// assert resulting values

assertTrue(server.isPresent("ikrueger"));
assertTrue(server.isPresent("mamenari"));

}
```
Mock Objects: Benefits & Limitations

• Benefits:
  – Allows TDD to focus on one class at a time
  – Decouples tests from heavyweight partner objects
  – Supports deterministic tests for “nondeterministic” units

• Limitations:
  – Interaction protocols can become implementations specific (order/number of messages/parameter values) and thus brittle
  – Refactorings on those protocols can break the tests easily
  – Results in increased maintenance to keep tests current
  – Sometimes too much decoupling rather than straightforward unit tests with jUnit
  – If interaction protocol is captured incorrectly, then the test may pass although the integration tests will fail
What have you learned today?
Learning Goals

• Understand the roles and limitations of testing in the software development process

• Be able to distinguish between unit, integration, system and acceptance testing

• Understand the value proposition of test-driven development

• Be able to design basic unit tests

• Be able to implement basic unit tests in jUnit

• Understand the need for Mock Objects

• Be able to comprehend basic unit tests with Mock Objects

• Understand the benefits and limitations of Mock Objects