Test-Driven Development (TDD)

CS 4501 / 6501
Software Testing

[Lasse Koskela, “Test Driven”, Chapters 2-3]
Agile Airplane Testing

Test harness:
- Appearance matches
- Color coding in place
- Fly 6ft (or 2m) over table

Implementation:
- Fold airplanes

Testing:
- Verify with test harness (accept or reject)

[Refer to “Agile Airplane Testing,” class meeting 6-Sep-2018]

[Image from http://www.twilightsoftwares.com]
TDD: Red-Green-Refactor Process

1. Write a test that fails
2. Make only enough code for it to pass
3. Improve code quality

Repeat Process
Overview of Process

1. From requirements to tests
2. Choosing the first test
3. Breadth-first, depth-first
4. Let’s not forget to refactor
5. Adding a bit of error handling
6. Loose ends on the test list
7. Repeat
Exercise: Requirements

• Imagine we are implementing a subsystem for the corporate email application.

• This subsystem is responsible for providing mail-template functionality so that the CEO’s assistant can send all sorts of important, personalized emails to all personnel with a couple of mouse-clicks.

• How would tests drive the development of this subsystem?
1. From Requirements To Tests

The first step in TDD is writing a failing test, we need to figure out what desired behavior we’d like to test for.

- Decomposing requirements
  - Template system as **tasks** – “things we need to do”
    - When completed, lead to satisfying the original requirements
  - Template system as **tests** – “thing we need to verify”
    - When passing, lead to the requirements being satisfied
Example: Tasks vs. Tests

Imagine you are implementing a subsystem for an email application.

<table>
<thead>
<tr>
<th>Template system as <strong>tasks</strong></th>
<th>Template system as <strong>tests</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Write a regular expression for identifying variables from the template</td>
<td>• Template without any variables renders as is</td>
</tr>
<tr>
<td>• Implement a template parser that uses the regular expression</td>
<td>• Template with one variable is rendered with the variable replaced with its value</td>
</tr>
<tr>
<td>• Implement a template engine that provides a public API and uses the template parser internally</td>
<td>• Template with multiple variables is rendered with the appropriate placeholders replaced by the associated values</td>
</tr>
<tr>
<td>• ...</td>
<td>• ...</td>
</tr>
</tbody>
</table>

Idea of what we should do, easy to lose sight of the ultimate goal – *not represent progress* of the produced software

Idea of what should be done – **connect to capabilities** of the produced software

[Koskela, p. 46]
What Are Good Tests Made Of?

- Tests are generally better than tasks for guiding our work, but does it matter what kind of tests we write?
  - Sure it does!

- Two properties of a good test
  - A good test is **atomic**
    - Keeps things small and focused
  - A good test is **isolated**
    - Doesn’t depend on other tests
Programming by Intention

• Given an initial set of tests, pick one that is potentially lead to most progress with least effort

• Write test code
  • How to test something that doesn’t exist without breaking our test?
    • Imagine code exists

• Benefit of programming by intention
  • Focus on what we could have instead of what we do have
2. Choosing the First Test

Restrict focus, do not worry about the whole system

- Before coming up with an initial list of tests, **define** a set of **requirements** for the subsystem under test.

- Example requirements:
  - System replaces variable placeholders like `${first_name}` with values provided at runtime.
  - Attempt to send a template with undefined variables raises error.
  - System ignores variables that are not in the template.

- Example corresponding tests:
  - Evaluating template "Hello, ${name}" with value "Reader" results in "Hello, Reader".
  - Evaluating "${greeting}, ${name}" with "Hi" and "Reader" result in "Hi, Reader".
  - Evaluating "Hello, ${name}" with "name" undefined raises MissingValueError.
Writing The First Failing Test

• We got a list of tests that tell us exactly when the requirements have been fulfilled. Now, we start working through the list, making them pass one by one

• Consider the following test
  • Evaluating template “Hello, ${name}” with value “Reader” results in “Hello, Reader”

• Now, let’s create a JUnit test
Example

• Step 1: Creating a **skeleton** for our tests

```java
public class TestTemplate {
}
```

• Step 2: Adding a **test method**

```java
import org.junit.Test;

public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {

    }
}
```
Example

• Step 3: Writing the **actual test**

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

public class TestTemplate {

    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }
}
```

assuming that the implementation is there (even though it isn’t)
Example

- Now, the compiler points out that there is no such constructor for `Template` that takes a `String` as a parameter.

- Step 4: **Satisfying the compiler** by adding empty methods and constructors.

```java
public class Template {
    
    public Template(String templateText) {
    
    }
    
    public void set(String variable, String value) {
    
    }
    
    public String evaluate() {
        return null;
    }
    
}
Example

• Step 5: Running test
  • Yes, the test fails – not surprisingly, because we haven’t implemented the methods yet
  • Benefit: to check that the test is executed, not the test result

The red phase of the TDD cycle

What we have now tell us when we are done with this particular task

“when the test passes, the code does what we expect it to do”
Example

• Step 6: Making the first test pass
  • Passing as quickly as possible with a hard-coded return statement

```java
public class Template {
    public Template(String templateText) {
    }
    public void set(String variable, String value) {
    }
    public String evaluate() {
        return "Hello, Reader";
    }
}
```

The green phase of the TDD cycle

2 dimensions to move forward:
• Variable
• Template text
Example

• Step 7: Writing another test

• Passing as quickly as possible with a hard-coded return statement

```java
public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}"");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentValue() throws Exception {
        Template template = new Template("Hello, ${name}"");
        template.set("name", "someone else");
        assertEquals("Hello, someone else", template.evaluate());
    }
}
```

Forcing out the hard-coded return statement with another test

The hard-coded evaluate method in the Template class will no longer pass this test

How to make the test pass
- Step 8: **Revising code** (to make the second test pass by storing and returning the set value)

```java
public class Template {
    private String variableValue;

    public Template(String templateText) {
    }

    public void set(String variable, String value) {
        this.variableValue = value;
    }

    public String evaluate() {
        return "Hello, " + variableValue;
    }
}
```

Our test passes again with minimal effort.

Our test isn’t good enough yet because of the hard-coded part.

To improve the test’s quality, follow three dimensions to push our code: **variable, value, template**.
Example

• Step 9: **Revising test**

```java
public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentTemplate() throws Exception {
        Template template = new Template("Hi, ${name}");
        template.set("name", "someone else");
        assertEquals("Hi, someone else", template.evaluate());
    }
}
```

- Rename test to match what we’re doing
- Squeeze out more hard coding
- Hard-coded return from the production code won’t work anymore
  - Parsing
  - Rendering
3. Breadth-First, Depth-First

- What to do with a “hard” red phase?
  - Issue is “What to fake” vs. “What to build”

- “Faking” is an accepted part of TDD
  - That is, “deferring a design decision”
Breadth-First

• Implement the higher-level functionality first by faking the required lower-level functionality
• Implement the **lower-level** functionality first and only compose the higher-level functionality once all the ingredients are present.
Back to Our Example

- Assume we are dealing with “Hello, ${name}”
- We can fake the lower-level functionality
- Do breath-first
Faking Details a Little Longer

• Handling variables as variables

```java
public class Template {
    private String variableValue;
    private String templateText;

    public Template(String templateText) {
        this.templateText = templateText;
    }

    public void set(String variable, String value) {
        this.variableValue = value;
    }

    public String evaluate() {
        return templateText.replaceAll("\\$\\{name\\}", variableValue);
    }
}
```

Store the variable value and the template text somewhere

Make evaluate replace the placeholder with the value
Proceed with the TDD Cycle

• Run the tests
• All tests are passing
• Now, add more test to squeeze out the fake stuff
Squeezing Out The Fake Stuff

• Writing test for **multiple variables** on a template

```java
@Test
public void multipleVariables() throws Exception {
    Template template = new Template("${one}, ${two}, ${three}");
    template.set("one", "1");
    template.set("two", "2");
    template.set("three", "3");
    assertEquals("1, 2, 3", template.evaluate());
}
```

The red phase

This test fails

To get the test passing as quickly as possible, do the search-and-replace implementation
import java.util.Map;
import java.util.HashMap;
import java.util.Map.Entry;

public class Template {
    private Map<String, String> variables;
    private String templateText;

    public Template(String templateText) {
        this.variables = new HashMap<String, String>();
        this.templateText = templateText;
    }

    public void set(String name, String value) {
        this.variables.put(name, value);
    }

    public String evaluate() {
        String result = templateText;
        for (Entry<String, String> entry : variables.entrySet()) {
            String regex = "\\$\{" + entry.getKey() + "\}\";
            result = result.replaceAll(regex, entry.getValue());
        }
        return result;
    }
}

Store variable values in HashMap
Loop through variables
Replacing each variable with its value
Run tests again, Nothing’s broken!
Special Test Case

- Evaluating template “Hello, ${name}” with values “Hi” and “Reader” for variables “doesnotexist” and “name”, results in the string “Hello, Reader”

```java
@Test
public void unknownVariablesAreIgnored() throws Exception {
    Template template = new Template("Hello, ${name}");
    template.set("doesnotexist", "Hi");
    template.set("name", "Reader");
    assertEquals("Hello, Reader", template.evaluate());
}
```

If we set variables that don’t exist in the template text, the variables are ignored by the Template class.

This test passes without any changes to the Template class.
Why Red Then Green

• We intentionally fail the test at first just to see that
  • Our test execution catches the failure
  • We are really executing the newly added test
    • Then proceed to implement the test and see the bar turn green again
4. Let’s Not Forget To Refactor

- At this point, it might seem that we didn’t add any code and there is nothing to refactor

  Refactoring applies to code and test code

- Though we didn’t add any production code, we added test code, and that is code – just like any other
  - We don’t want to let our test code rot and get us into serious trouble later

- What could we do about our test code?
  - Identify any potential refactoring
  - Decide which of them we’ll carry out
```java
public class TestTemplate {

    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}"");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentTemplate() throws Exception {
        Template template = new Template("Hi, ${name}"");
        template.set("name", "someone else");
        assertEquals("Hi, someone else", template.evaluate());
    }

    @Test
    public void multipleVariables() throws Exception {
        Template template = new Template("${one}, ${two}, ${three}"");
        template.set("one", "1");
        template.set("two", "2");
        template.set("three", "3");
        assertEquals("1, 2, 3", template.evaluate());
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception {
        Template template = new Template("Hello, ${name}"");
        template.set("doesnotexist", "Hi");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

}
Potential Refactoring in Test Code

• All tests are using a Template object
  • Solution: extract it into an instance variable rather than declare it over and over again, use fixtures

• The evaluate method is called several times as an argument to assertEquals
  • Solution: write a method that calls the evaluate method

• The Template class is instantiated with the same template text in two places
  • Solution: remove the duplicate by using fixtures (with some unified values)

Remove redundant tests
Revisit Current Test Class

Let's consider duplication between these tests

multipleVariables covers oneVariable and differentTemplate
-- thus, get rid of them

unknownVariablesAreIgnored can use the same template text as multipleVariables
public class TestTemplate {
    private Template template;

    @Before
    public void setUp() throws Exception {
        template = new Template("${one}, ${two}, ${three}");
        template.set("one", "1");
        template.set("two", "2");
        template.set("three", "3");
    }

    @Test
    public void multipleVariables() throws Exception {
        assertEquals(template EvaluateTo("1, 2, 3"));
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception {
        template.set("doesnotexist", "whatever");
        assertEquals(template EvaluateTo("1, 2, 3"));
    }

    private void assertTemplateEvaluateTo(String expected) {
        assertEquals(expected, template.evaluate());
    }
}
5. Adding a Bit of Error Handling

• Add exception test, using try/catch block with \( \text{fail()} \)

```java
@Test
public void missingValueRaisesException() throws Exception {
    try {
        new Template("${foo}").evaluate();
        fail("evaluate() should throw an exception if " +
             "a variable was left without a value!");
    } catch (MissingValueException expected) {
    }
}
```

```java
public class MissingValueException extends RuntimeException {
    // this is all we need for now
}
```
Adding a Bit of Error Handling (2)

- Add exception test, using `@Test (expected = ...)`

```java
@Test (expected=MissingValueException.class)
public void testmissingValueRaisesException() throws Exception {
    new Template("${foo}").evaluate();
}
```

```java
public class MissingValueException extends RuntimeException {
    // this is all we need for now
}
```

Except test – either try/catch with `fail()` or `@Test (expected=...)` fails. That means, we have to somehow check the missing variables.

Let’s make the test pass
How to get to the green phase as quickly as possible?
Writing Code To Make The Test Pass

• How do we know inside evaluate, whether some of the variables specified in the template text are without a value?

• Checking for remaining variables after the search-and-replace

```java
public String evaluate()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$\{" + entry.getKey() + "\\}\";
        result = result.replaceAll(regex, entry.getValue());
    }

    if (result.matches(".*\\$\{.+\\}.+\")
        throws new MissingValueException();

    return result;
}
```

Does it look like we left a variable in there?
Refactoring Toward Small Methods

- `evaluate()` is doing too many different things
  - Replacing variables with values, checking for missing values
- **Extracting** the check for missing variables into its own method

```java
public String evaluate()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$\\{" + entry.getKey() + "\\}";
        result = result.replaceAll(regex, entry.getValue());
    }
    checkForMissingValues(result);
    return result;
}

private void checkForMissingValues(String result)
{
    if (result.matches(".*\\$\\{.+.\\}.*"))
        throws new MissingValueException();
}
```
More Refactoring

• `evaluate()` is still doing two things:
  • Replacing variables with values, checking for missing values

• Extracting method refactoring
  • To create simple, single, clear purpose methods

```java
public String evaluate()
{
    String result = replaceVariables();
    checkForMissingValues(result);
    return result;
}
```

```java
private String replaceVariables()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$\{" + entry.getKey() + "\\}\";
        result = result.replaceAll(regex, entry.getValue());
    }
    return result;
}
```

```java
private void checkForMissingValues(String result)
{
    if (result.matches(".*\\$\{.+\\\}.\*"))
        throws new MissingValueException();
}
```
@Test
public void missingValueRaisesException() throws Exception 
{
    try {
        new Template("${foo}").evaluate();
        fail("evaluate() should throw an exception if " +
            "a variable was left without a value!"
    } catch (MissingValueException expected) {
        assertEquals("No value for ${foo}", expected.getMessage());
    }
}

import java.util.regex.Pattern;
import java.util.regex.Matcher;
private void checkForMissingValues(String result)
{
    Matcher m = Pattern.compile("\\$\\{.+\\}\`).matcher(result);
    if (m.find())
        throw new MissingValueException("No value for " + m.group());
}
6. Loose Ends On The Test List

• Testing for performance

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

public class TestTemplatePerformance {
    // Omitted the setUp() for creating a 100-word template
    // with 20 variables and populating it with approximately
    // 15-character values

    @Test
    public void templateWith100WordsAnd20Variables() throws Exception {
        long expected = 200L;
        long time = System.currentTimeMillis();
        template.evaluate();
        time = System.currentTimeMillis() - time;
        assertTrue("Rendering the template took " + time + "ms while the target was " + expected + "ms",
                    time <= expected);
    }
}
```
Test That Dooms Current Implementation

- Write test that provides whether the code’s current behavior are correct

```java
@Test
public void variablesGetProcessedJustOnce() throws Exception {
    template.set("one", "${one}");
    template.set("two", "${three}");
    template.set("three", "${two}");
    assertTemplateEvaluatesTo("${one}, ${three}, ${two}");
}
```
Summary

• TDD
  • Test: write a test
  • Code: write code to make the test pass
  • Refactor: find the best possible design for what we have, relying on the existing tests to keep us from breaking things while we’re at it

• Encourages good design, produces testable code, and keeps us away from over-engineering our system because of flawed assumptions