Test Driven
Lasse Koskela
Chapter 2: Beginning TDD

Paul Ammann

&

Jeff Offutt

2018

Overview

- From requirements to tests
- Choosing the first test
- Breadth-first, depth-first
- Let's not forget to refactor
- Adding a bit of error handling
- Loose ends on the test list

Experience is a hard teacher because she gives the test first, the lesson afterward

General Problem Statement

Build a subsystem for an email application

Allow users to use email templates to create personalized responses to people in the company

From Requirements To Tests: Template System Example

Template System as Tasks

- Write a regular expression to identify variables from the template
- Implement a template parser that uses the regex
- Implement a template engine that provides a public application programmer interface (API)
- …

Template System as Tests

- Template without any variables renders as is
- Template with one variable is rendered with variables replaces by value
- Template with multiple variables is rendered with each variable replace by an appropriate value
- …

Which approach do you find more natural?

What Are Good Tests Made Of?

- A good test is atomic
  - Does one and only one thing
  - Keeps things focused
- A good test is isolated
  - Does not depend on other tests
  - Does not affect other tests

This is not a complete list, but a start

Programming By Intention

- Given an initial set of tests
  - Pick one
  - Goal: Most progress with least effort
- Next, write test code
  - Wait! Code won't compile!
  - Imagine code exists
  - Use most natural expression for call (design the API)
- Benefit of programming by intention
  - Focus on what we COULD have
  - Not what we DO Have

Evolutionary API design from client perspective

Choosing The First Test

- Some detailed requirements:
  - System replaces variable placeholders like ${firstname} in template with values provided at runtime
  - Sending template with undefined variables raises error
  - System ignores variables that aren't in the template
- Some corresponding tests:
  - Evaluating template "Hello, ${name}" with value name = Reader results in "Hello, Reader"
  - Evaluating "${greeting}, ${name}" with "Hi" and "Reader" results in "Hi, Reader"
  - Evaluating "Hello, ${name}" with "name" undefined raises MissingValueError

Writing the First Failing Test

- Evaluating template "Hello, ${name}" with value Reader results in "Hello, Reader"
- Listings 2.1, 2.2, 2.3

We just made the following decisions about the implementation

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

DO try this at home

Code To Make Compiler Happy

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

- This allows the test to compile
- The test fails, of course (listing 2.4)
- Running it should result in a RED bar
- We're at the RED part of RED-GREEN-REFACTOR

Making The First Test Pass

```
public class Template
{
   public Template (String templateText) {
   }
   public void set (String variable, String value) {
   }
   public String evaluate() {
      return “Hello, Reader”;     // Couldn’t get simpler than this!
   }
}
```

- We’re looking for the green bar – listing 2.6
- We know this code will change later – That’s fine
- Three dimensions to push out code : variable, value, template

Another Test

- Purpose of 2nd test (listing 2.7) is to "drive out" hard coding of variable's value
- Koskela calls this triangulation

Triangulate with a different value

```
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());
  }
}
```

Another Test

- Revised code (listing 2.8) on page 57

```
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;
      }
  }
```

Another Test

- Note revisions to JUnit in listing 2.9

```
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 coded values

Breadth-First, Depth-First

- What to do with a "hard" red bar?
- Issue is what to fake vs. what to build
- "Faking" is an accepted part of TDD that means "deferring a design decision"
- Depth first means supplying detailed functionality
- Breadth first means covering end-to-end functionality (even if some is faked)

Handling Variables as Variables: Listing 2.10

```
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);
    }
}
```

Multiple Variables

- Test (page 60)

```
@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());
}
```

Multiple Variables: Listing 2.11

```
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

Replace each variable with its value

Special Test Case

- Special case test page 62
- This test passes for free!

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

Let's Not Forget To Refactor

- Refactoring applies to both the functional code and to the test code
- Compare listing 2.12 (pg 63) with refactored listing 2.13 (pg 64)
- Note use of fixtures

Listing 2.12: Problems

```
@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 ("name", "Reader");
    template.set ("doesnotexist", "Hi");
    assertEquals ("Hello, Reader", template.evaluate());
}
```

Listing 2.12: Problems

```
@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 ("name", "Reader");
    template.set ("doesnotexist", "Hi");
    assertEquals ("Hello, Reader", template.evaluate());
}
```

Redundancy creates risk

Listing 2.12: Problems

```
@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 ("name", "Reader");
    template.set ("doesnotexist", "Hi");
    assertEquals ("Hello, Reader", template.evaluate());
}
```

More redundancy

Listing 2.12: Problems

```
@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 ("name", "Reader");
    template.set ("doesnotexist", "Hi");
    assertEquals ("Hello, Reader", template.evaluate());
}
```

Same test twice

Listing 2.12: Problems

```
@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 ("name", "Reader");
    template.set ("doesnotexist", "Hi");
    assertEquals ("Hello, Reader", template.evaluate());
}
```

Same test values twice

Listing 2.13: Refactor Test Code

```
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 {
    assertTemplateEvaluatesTo ("1, 2, 3");
  }

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

  private void assertTemplateEvaluatesTo (String expected) {
    assertEquals (expected, template.evaluate());
  }
}
```

Common fixtures for all tests

Simple, focused tests

Shared method

Adding A Bit Of Error Handling

- A variable does not have a value
- Adding exception test listing 2.14
  - Note different approaches to testing exceptions
    - try-catch block with fail() vs. @Test (expected= …)

```
@Test
public void missingValueRaisesException() throws Exception {
    try {
        new Template ("${foo}").evaluate();
        fail ("evaluate() should throw an exception if "
                + "a variable does not have a value!");
    } catch (MissingValueException expected) {
    }
}
```

Listings 2.15, 2.16

```
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 (".*\\$\\{.+\\}.*")) {
    throw new MissingValueException();
  }
  return result;
}
```

Does **result** still have a variable with no value?

Refactor to move an if-block out of **evaluate**()

```
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 (".*\\$\\{.+\\}.*")) {
    throw new MissingValueException();
  }
}
```

Extract Method Refactoring

```
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;
}
```

- evaluate() does two things:
  1. Replacing variables with values
  2. Checking for missing values
- These are different, and should be separate
  - This is called "*extract method*" refactoring

More Refactoring: Listing 2.17

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

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;
}

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

evaluate() method's internals better balanced

New method is simple and has a single, clear purpose

Must re-run all the tests to ensure nothing broke

A Truly Difficult Special Case

- What happens in the special case that a value has a special character such as `$', `{', or `}'?
  - These are the kinds of non-happy path tests TDD often skips
- Implementing this test breaks the current implementation:

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

Values have the special characters `$', `{', and `}'

- regexp throws an IllegalArgumentException
  - Requiring a major design change

Chapter 3 addresses major rewrites