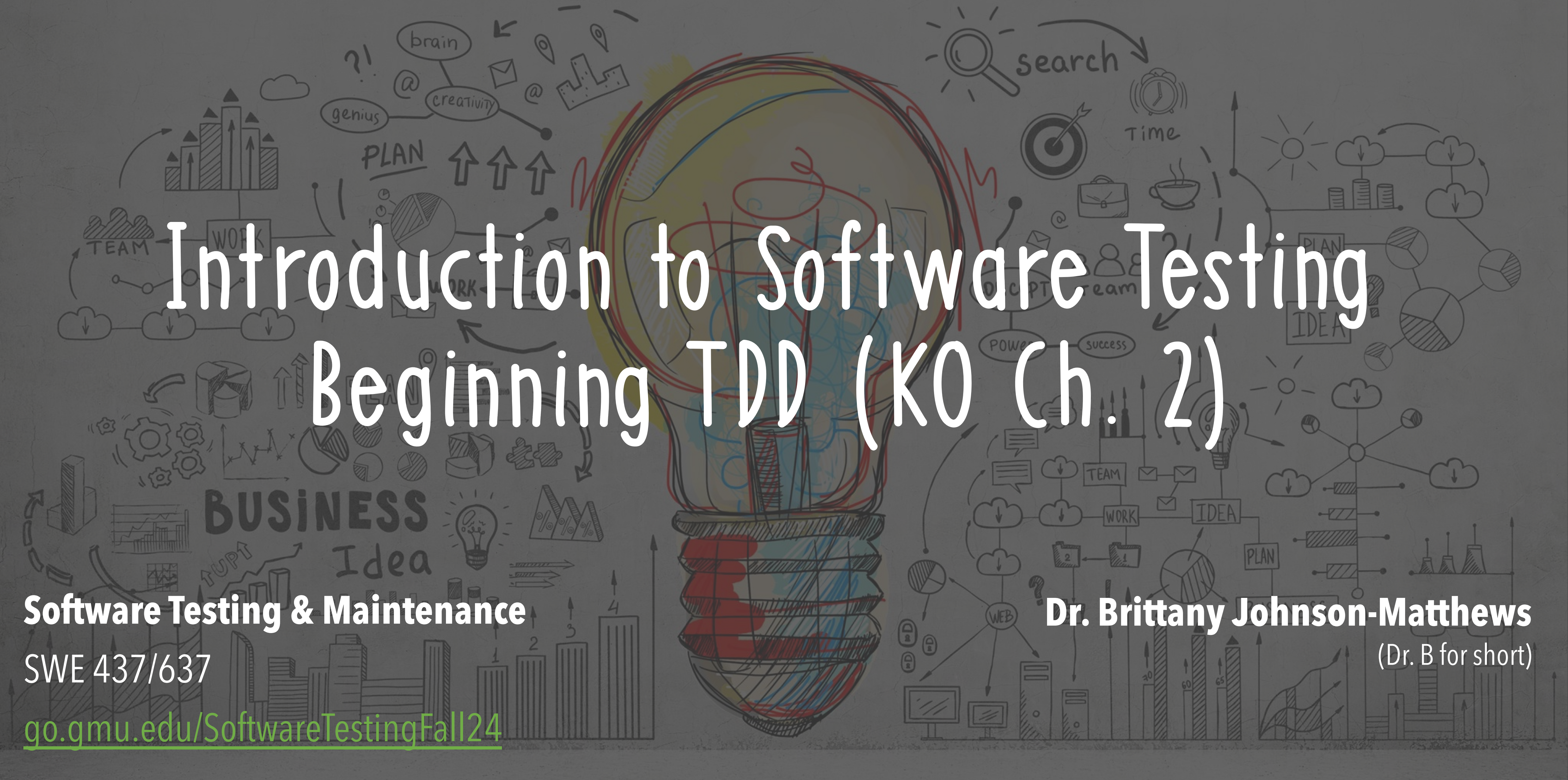
Introduction to Software Testing
Beginning TDD (KO Ch. 2)



Software Testing & Maintenance

SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Today's In-class Exercise

Today's exercise will be **immersed** and **individual**.

We will stop periodically during today's lecture for **discussion and coding**.

You should also follow along with me using the book or an IDE.

This is part of your participation grade and beneficial to your success!

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

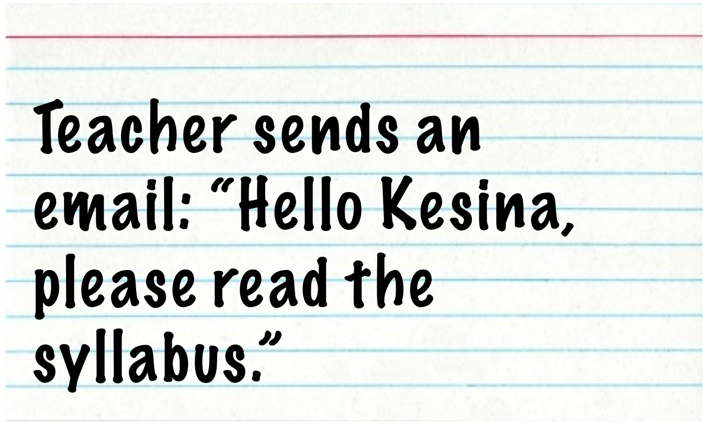
Example from book

General Problem Statement

Build a subsystem for an email application

Allow users to use **email templates** to create personalized responses for repeated email messages

Example:



Teacher sends an email: "Hello Kesina, please read the syllabus."

From requirements to tests: template system

Template system as **tasks**

- Write 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 render 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 makes a good test?

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**

Write the first (failing) test

Evaluating template "**Hello, ${name}**" with value **Reader** results in "**Hello, Reader**"

Listings 2.1, 2.2, 2.3

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

We just made the following decisions about the implementation

Try this on your computer

Now, let's make it compile

This allows the test to **compile**

The test **fails**, of course

Running it should result in a **RED** bar

We're at the RED part of RED-GREEN-REFACTOR

Listing 2.4

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

From RED to GREEN

Listing 2.6

```
public class Template
{
   public Template(String templateText) {
   }
   public void set(String variable, String value) {
   }
   public String evaluate() {
      return "Hello, Reader";     // Minimal code to make test pass
   }
}
```

We're looking for the **GREEN** bar

We know this code **will change later** – that's fine

3 dimensions to push out code: variable, value, template

Test #2: triangulation

Purpose of 2nd test is to "**drive out**" hard coding of variable's value

Koskela calls this **triangulation**

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

Triangulate with a different value

Listing 2.7

Making the 2nd test pass

Revised code

Listing 2.8

```
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;
        }
  }
```

3rd test

Note revisions to JUnit test

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 term in TDD that means "**deferring a design decision**"

Depth first means supplying detailed functionality

Breadth first means covering end-to-end functionality (even if part is *faked*)

Making the 3rd test pass

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);
    }
}
```

Change “Hi, ${name}” to “Hi, someone else”

4th test: multiple variables

A new test with more than one variable

Section 2.3.2

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

4th test: multiple variables

```
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

Listing 2.11

Breadth-first, depth-first

Special case of a variable that does not exist

 -Variable should simply be ignored

This test passes for free!

Section 2.3.2

```
@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 remember to refactor!

Refactoring applies to both the **functional code** and to the **test code**

Compare listing 2.12 with refactored listing 2.13 (Section 2.4)

Listing 2.12

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

Can you spot any problems?

Issues with redundancy

```
@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

Issues with redundancy

```
@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

Issues with redundancy

```
@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

Issues with redundancy

```
@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

Refactor to reduce redundancy

Listing 2.13

```
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

Add some error handling

A variable without a value?

Adding exception test

Note different approaches to testing exceptions

- -Try-catch block with fail() vs. @Test(expected=…)

Listing 2.14

```
@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) {
    }
}
```

Extract method refactoring

Listing 2.15

```
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;
}
```

Check if **result** still has a variable with no value

Listing 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());
  }
  checkForMissingValues(result);
  return result;
}

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

Refactor so evaluate() does only one thing

More refactoring: Listing 2.17

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

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

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

regexp throws an IllegalArgumentException

- Requiring a major design change

Chapter 3 addresses this...