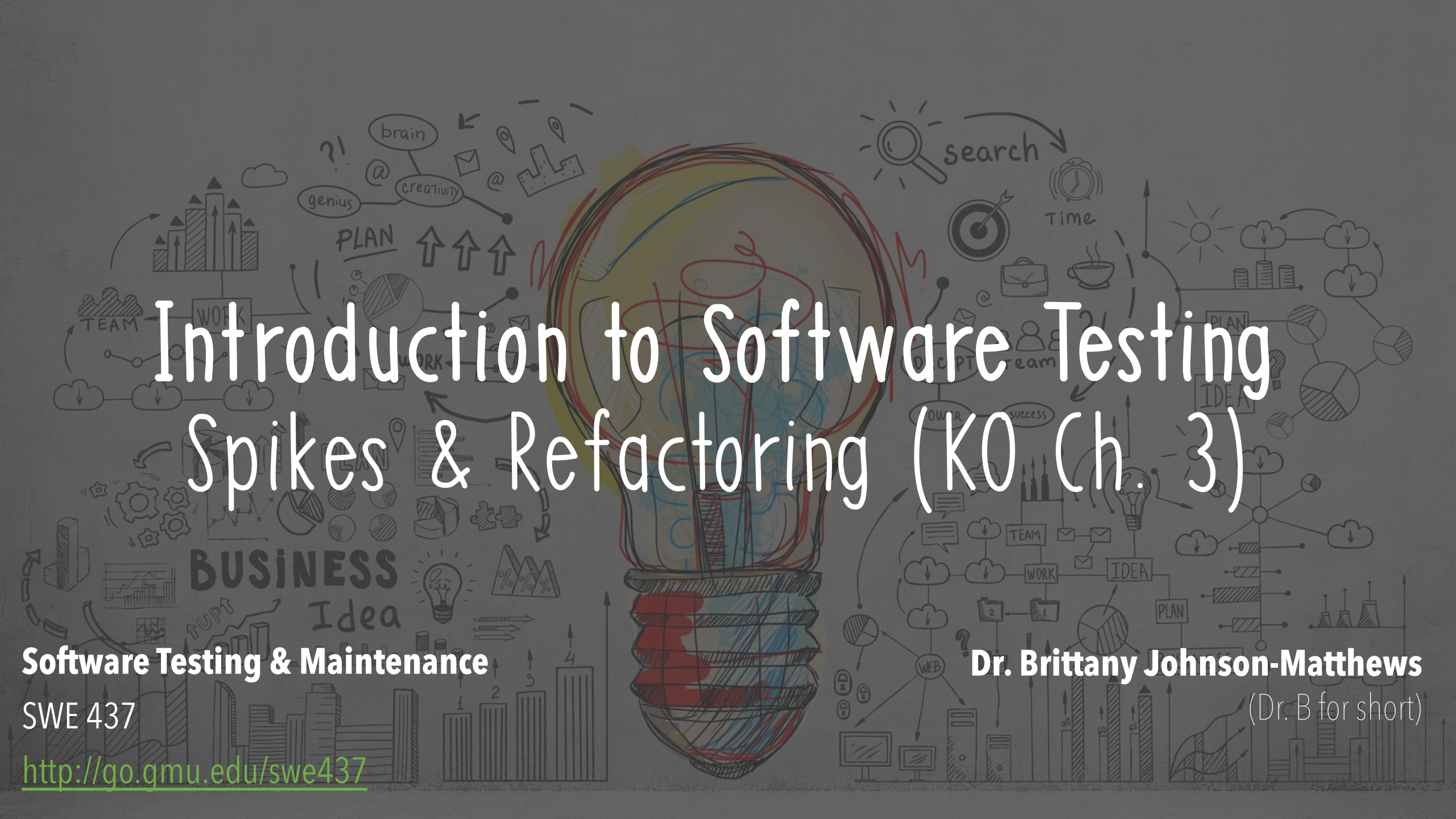


Introduction to Software Testing
Spikes & Refactoring (KO Ch. 3)

Software Testing & Maintenance

SWE 437

http://go.gmu.edu/swe437

Dr. Brittany Johnson-Matthews

(Dr. B for short)

Overview

Exploring a potential solution

Changing design in a **controlled** manner

Taking the **new design** further

Most excellent designs are the result of a continuous process of simplification and refinement

The problem from Ch. 2

Existing design replaced variables via simple matching

-For all variables `v`, replace `${v}` with its value:

```
result = result.replaceAll (regex, entry.getValue())
```

Failing test from chapter 2: Sets the value to `"${one}, ${two}, ${three}"`

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

Tweaking the current design won't make this test pass

What is a spike?

A detour to **learn something** new

-Package, details on API, etc.

-Whether proposed design will work

Spikes are **experimental** in nature

Self education – increase knowledge, skills, or abilities

Exploring a potential solution

Break the templates into "**segments**"

Prototyping with spikes

-A spike is a detour to learn

-In the template example, we learn more about using **regex**

Learn by writing tests (***learning tests***)

-Need to figure out an API?

-Write some tests that use the API

-**RegexLearningTest** on Ammann's website, from section 3.3
https://cs.gmu.edu/~pammann/Koskela/code/RegexLearningTest.java

Example spike for learning an API

-Note that Koskela thought **find()** would count occurrences

-He learned it breaks strings into pieces

Learn on a short detour, then apply

Controlled changes to design

Creating an **alternative implementation**

Start with the **“low hanging fruit”**

-TDD Development of Template parser

Remove duplication from tests

-Refactoring is *always* important

Controlled changes to design

Apply learning from the spike

-Final code version (not Segment class, originally a String)

```
  private void append(String segment, StringBuilder result) {
     if (isVariable(segment) { evaluateVariable(segment, result); } // dispatching ☹
        else { result.append(segment);}
}
```

-Koskela refactors substantially

-TemplateParse.java

Controlled changes to design

Switching over **safely**

Adopting the **new implementation**

-Recoding the `evaluate()` method

Cleaning up by extracting methods (more refactoring)

-Pull out the old stuff that's no longer relevant

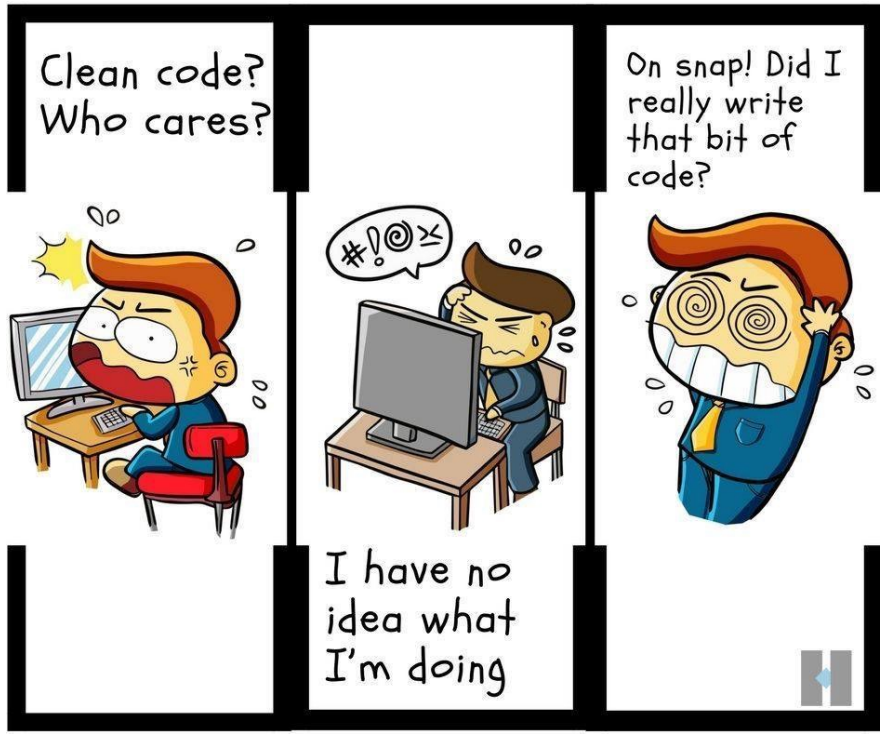
Result is new `Template` class (Template.java)

No new functionality, but definitely improved!

Improving the new design

Keeping things compatible

- Build on existing functionality
- Refactor logic into objects
  - Motivation for segment class
- Make the switchover
- Getting caught by safety nets
  - Don't forget your exceptional behavior!
- Delete dead code + further clean up



Test sets make requirements concrete.

Core Idea

Use regexp to break the following string:

“${greeting} ${fname},

Thank you for your interest in ${product}.”

Into the following 5 pieces:

“${greeting}” “${fname}”

”,

Thank you for your interest in “ “${product}” “.”

Now the variables can easily be identified and **replaced**

regex will not explode if values have ‘$’, or ‘{‘, or ‘}’

Practice, practice, practice!

Chapter 3 has a lot of details that you should explore on your own.

I suggest going through the exercise with the code and JUnit

A spike for you!

Code location:

https://cs.gmu.edu/~pammann/Koskela/code/

Template.java, Segment, PlainText, Variable