Team Deliverable #5

Overview

The primary goal for this deliverable was fault injection. To do this we injected 5 different errors into the sakai source code that would cause test cases that would normally pass to fail. The primary purpose of this was to test that our script was in fact working and would report errors in the code should they be present, as opposed to just always stating that a test case passed. This also was useful for helping us better understand the code as it would not be possible to make the code fail in a specific way without a solid understanding of it. Another purpose of the fault injection would be to show that test cases were written correctly. Overall this was one of the easier deliverables of the project.

Important Concepts

The process of injecting the source code with faults is a pretty simple process, assuming one is familiar with how the code and drivers work. There are a few things that we found were important to consider. The first of these is that what ever fault one is trying to inject it should not cause the code to fail to compile. The test cases are not meant to test for syntax errors they are written to test functionality. If the code cannot compile then it not execute at all which wont help with testing. Syntax errors are best left to an IDE to keep track of. The next important aspect of this process is that one needs to have a firm grasp of how the code they are injecting errors into works. This will help in avoiding the previously mentioned concern.

Injection Process

The injection process started after we determined a few key points in the code that would cause the code to fail. We chose five places to place changes to the source code that would cause a failure. These were located in the StringUtil.java file, specifically to the contains and trimToLowerNull methods. Once these locations were chosen small changes were made to the code that would cause certain test cases to fail.

The Injected Faults

The first fault injected was located in the contains method, specifically around line 442.

Here we replaced:

```
if (stringCollection == null || value == null) return true;
```

with

```
if (stringCollection == null || value == null) return false;
```

this would cause the test cases involving null arrays of strings to return true when the contains method was called. The array would of course not contain anything as it is null.

The Second fault was located within the same method but a few lines lower, near line 445

Here we replaced:

```
if (value.length() == 0) return true;
```

with

```
if (value.length() == 0) return false;
```

This would cause an error when one searched an empty array for any string, which should return false as an empty string cannot contain an element by the very definition of what an empty string is.

The Third fault was injected a few lines lower, at line 453

Here we replaced

```
if (value.equals((String) o)) return ;
```

with

```
if (value.equals((String) o)) return false;
```

Doing this would result in the method returning false when an array actually does contain the string it is being searched for.

We placed the Fourth fault was placed in the trimToNullLower method at line 231:

Here we commented out

```
value = value.trim();
```

Which would cause the method to NOT remove any spaces before or after the string. This caused Test Cases 7, 9, and 11 to fail as they all have leading spaces.

The final Injection was near line 239.

At this spot we changed

```
return value.toLowerCase();
```

with

```
return value;
```

This would stop the method from turning uppercase letters into lowercase letter.

Conclusion

As one can see the changes themselves were not too intensive, the results however, were quite dramatic. Out of our 25 test cases these injections caused 9 to fail. It was quite clear from these results that our testing automation framework is successful in its goals and that any test cases using these methods were constructed correctly, and more importantly tested correctly.