Names: Madison Betz and Kaiden Pollesch

Date: 3/24/2025

Assignment Title: Lab 7 Code Coverage with the Word Changer

Repository Link: https://github.com/msoe-SWE2721/2025-lab7-wordchangerb-121_betz_pollesch_lab7_2025.git

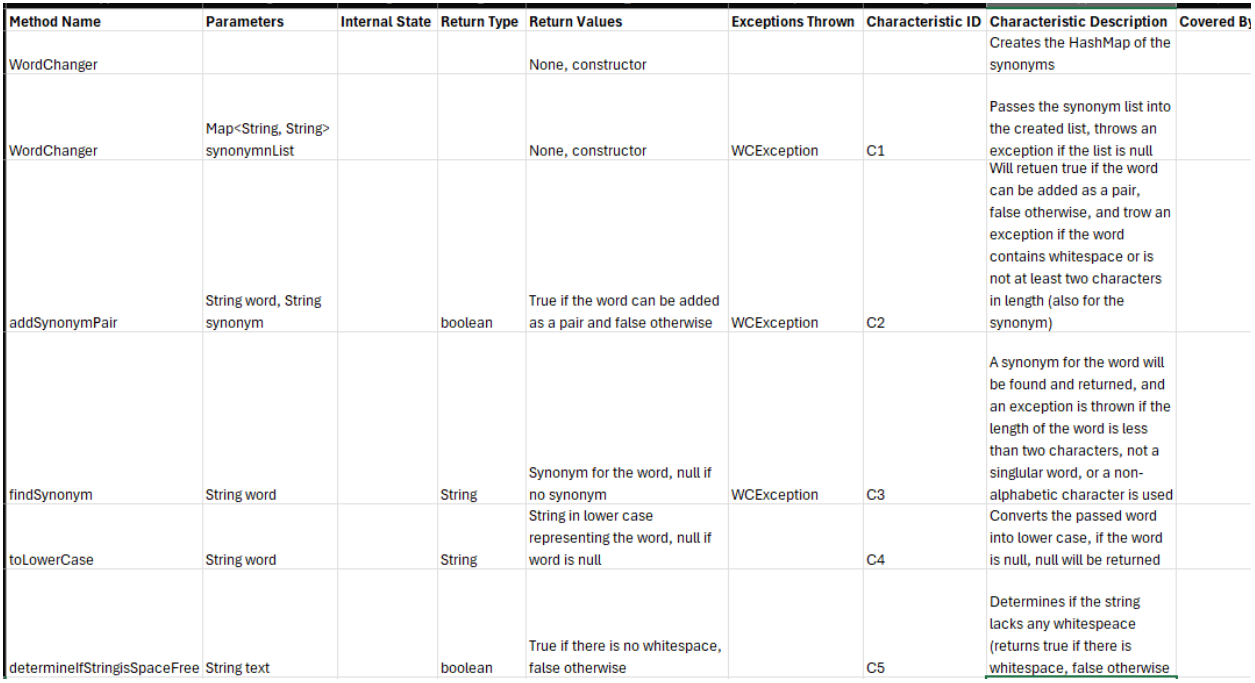
Introduction

What are you trying to accomplish with this lab?

In this lab, we are trying to accomplish thorough testing using different techniques and methods we have learned about in lecture. We are given code that performs a certain function of replacing words with synonyms. Before creating our tests, we will create an input domain table and control flow graph to better understand how the code functions and how we should generate tests for all possible paths and edge cases.

Input Domain Analysis

Input Domain Table



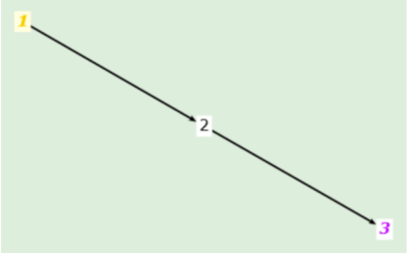
Method Name	Parameters	Internal State	Return Type	Return Values	Exceptions Thrown	Characteristic ID	Characteristic Description	Covered By
WordChanger				None, constructor			Creates the HashMap of the synonyms	
WordChanger	Map<String, String> synonymnList			None, constructor	WCException	C1	Passes the synonym list into the created list, throws an exception if the list is null	
addSynonymPair	String word, String synonym		boolean	True if the word can be added as a pair and false otherwise	WCException	C2	Will retuen true if the word can be added as a pair, false otherwise, and trow an exception if the word contains whitespace or is not at least two characters in length (also for the synonym)	
findSynonym	String word		String	Synonym for the word, null if no synonym	WCException	C3	A synonym for the word will be found and returned, and an exception is thrown if the length of the word is less than two characters, not a singlular word, or a non-alphabetic character is used	
toLowerCase	String word		String	String in lower case representing the word, null if word is null		C4	Converts the passed word into lower case, if the word is null, null will be returned	
determineIfStringisSpaceFree	String text		boolean	True if there is no whitespace, false otherwise		C5	Determines if the string lacks any whitespace (returns true if there is whitespace, false otherwise	

Control Flow Graphs

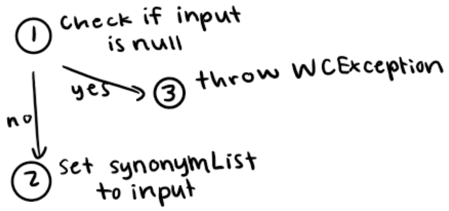
Control Flow Graph (Drawn then Generated)

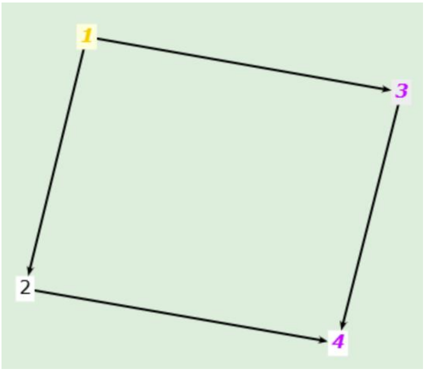
Default Constructor (CC = 1)

1
Create empty HashMap

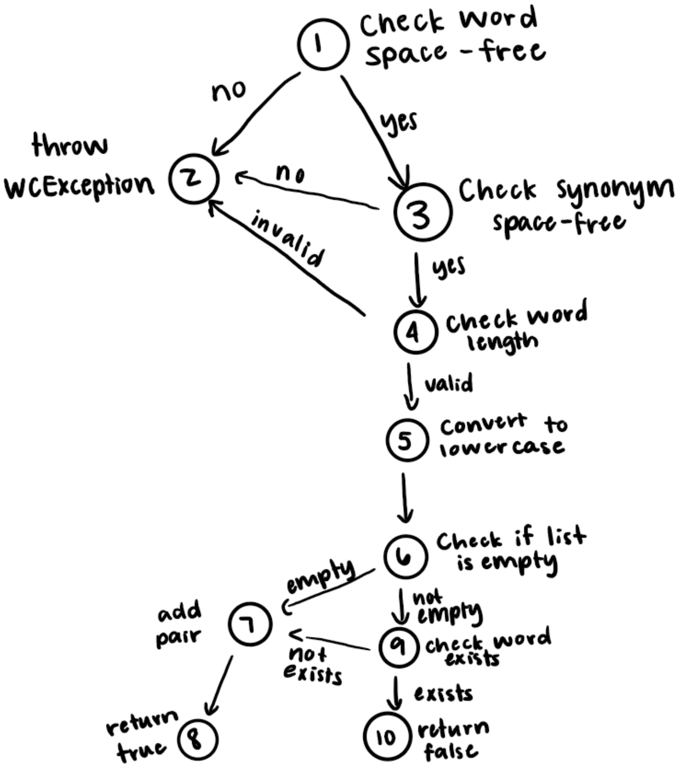


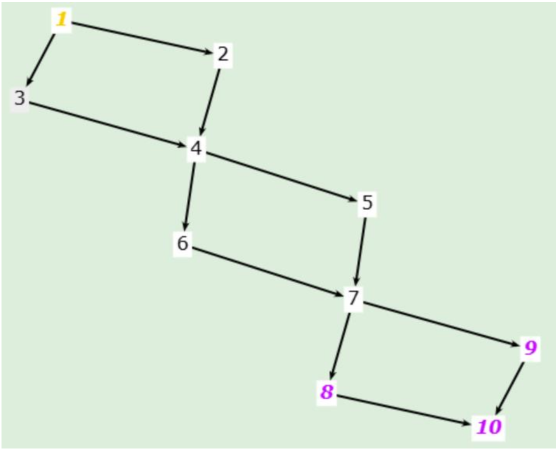
Parameterized Constructor (CC = 2)



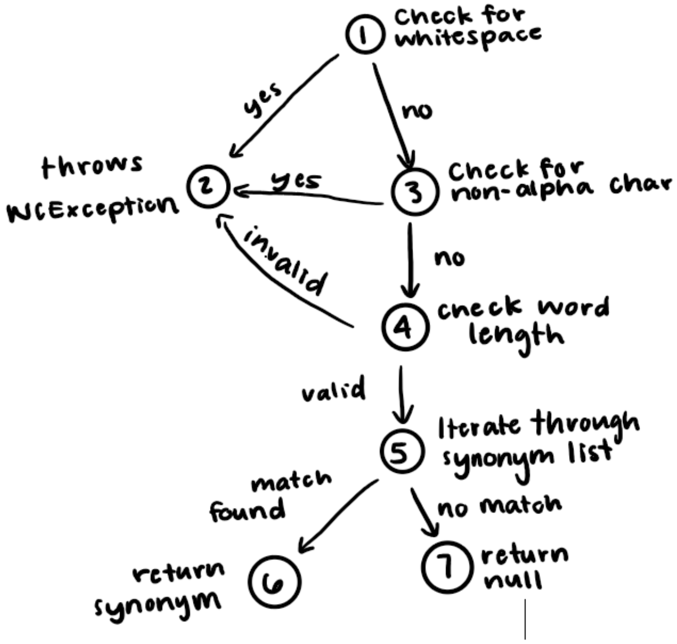


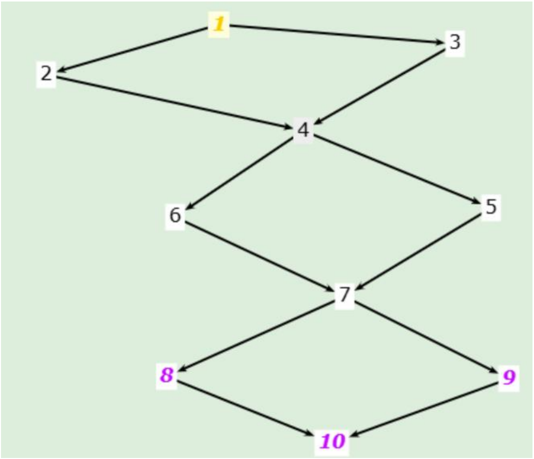
AddSynonymPair(String word, String synonym) (CC = 7)





FindSynonym(String word) (CC = 6)





Differences

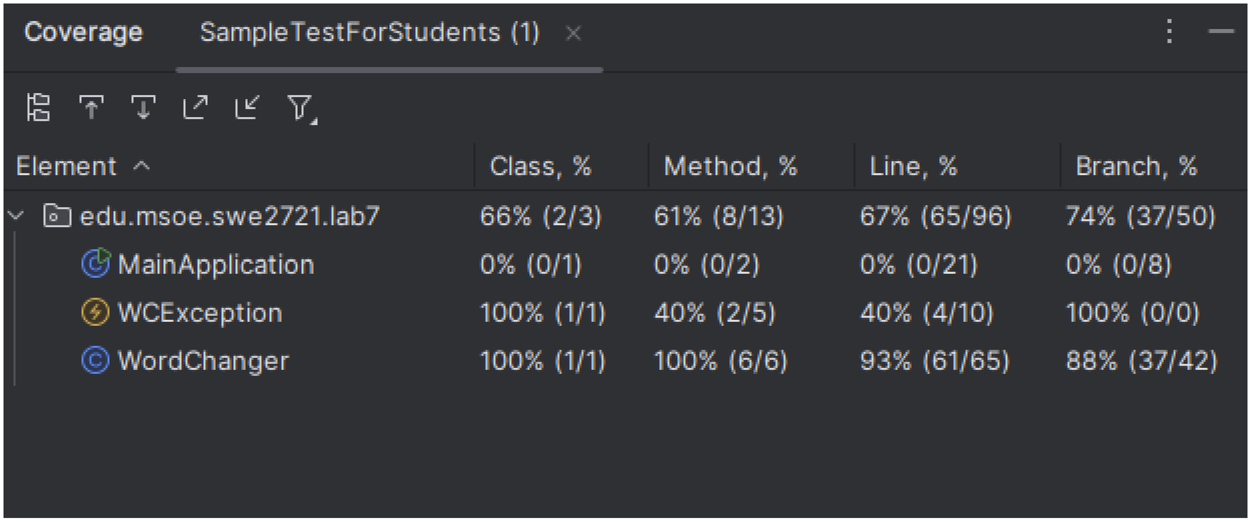
The main differences between the drawn and generated control flow graphs are that the drawn graphs are able to have more detail and explanation of what is happening at each node. This allows for the paths to be more easily recognized between nodes. The generated coverage graphs are simplistic, but they do make it easier to reorder where the nodes go.

Table

Method	Cyclomatic Complexity Paths	Prime Path Coverage (Algorithm 2)
`WordChanger()` (default constructor)	**Path 1**: Instantiating an empty `HashMap`.	**Path A**: Verifies creation of a new (empty) synonym list.
`WordChanger(Map<String, String>)`	**Path 1**: `synonymnList == null` -> throws exception. **Path 2**: `synonymnList != null` -> proceeds and sets internal map.	**Path B**: Checks exception thrown when `synonymnList` is `null`. **Path C**: Confirms successful initialization with a non-null map.
`addSynonymPair(String, String)`	**Path 1**: `word` or `synonym` has whitespace -> throws exception. **Path 2**: `word.length() <= 1` -> throws exception. **Path 3**: `synonym.length() <= 1` -> throws exception. **Path 4**: Map is empty -> adds pair.	**Path D**: Covers exceptions for whitespace in parameters. **Path E**: Covers exceptions for length checks. **Path F**: Covers adding to an empty map. **Path G**: Covers adding when map has entries.

	Path 5: Map not empty and no existing pair -> adds pair. **Path 6**: Pair exists -> returns false.	
`findSynonym(String)`	**Path 1**: Input has whitespace -> throws exception. **Path 2**: Input contains a non-alphabetic character -> throws exception. **Path 3**: Input length < 2 -> throws exception. **Path 4**: Key not found -> returns `null`. **Path 5**: Key found -> returns associated value.	**Path H**: Tests exception on whitespace or non-alphabetic. **Path I**: Tests word length criteria. **Path J**: Covers found vs. not found in the map.

Code Coverage Snapshots



Element	Class, %	Method, %	Line, %	Branch, %
edu.msoe.swe2721.lab7	66% (2/3)	61% (8/13)	67% (65/96)	74% (37/50)
MainApplication	0% (0/1)	0% (0/2)	0% (0/21)	0% (0/8)
WCException	100% (1/1)	40% (2/5)	40% (4/10)	100% (0/0)
WordChanger	100% (1/1)	100% (6/6)	93% (61/65)	88% (37/42)

Discussion

Can you identify any segments of code you might have missed had you used input domain analysis to construct your tests?

Segments of code we might have missed if we only used input domain analysis are different branching that happens in add synonym pair that would not be fully captured by input domain analysis. Edge cases are also harder to spot when using input domain analysis and are more visible in control flow graphs.

Were there any paths you listed based upon your cyclomatic complexity analysis which you could not write a test for?

Some of the paths that involved nested conditions within add synonym pair that create multiple paths that require tests to reach were difficult to write a test for. In addition to this, some of the paths involving exception throwing were challenging to test.

Were there any prime paths that were generated, you could not write a test for or revise the path to make a test for?
Most tests were able to be generated, but it was challenging to make tests involving exceptions being thrown. It was also difficult with paths that required specific input combinations and scenarios that would require intentionally wrong inputs to trigger certain code branches.

What is the main difference between the approach using cyclomatic complexity and the approach using prime paths in terms of how loops are handled?
The main difference between using cyclomatic complexity and using prime paths are that cyclomatic complexity considers the number of linearly independent paths through the code and prime paths look at multiple traversals of loops and emphasize different execution sequences within loops.

Were there any statements or branches that were infeasible to test based upon the code structure? Why was this the case?
Some statements that were infeasible were some extremely specific input combinations that create unlikely execution paths. Other difficult branches would be nested error handling that require unique parameters and constraints.

Without making changes to the code, could you change the code structure so that the code would be reachable and testable?
Without making changes to the code, it would be possible to break down more complex methods into smaller, more focused methods. In addition to this, separating the validation logic from the method functionality would help with making the code more testable.

Given your experience with Input Domain Analysis and this method, which approach feels easier to do and why?
The input domain analysis feels easier to do, but that might be because we have more experience with it. It is a simpler initial approach and focuses on characteristics that can be found quickly in the java docs. Input domain analysis breaks down the methods in a less complex way, however it does not capture how the methods interact. Cyclomatic complexity and prime path testing does a more thorough testing job, but it takes more to set up.

Things Gone Right / Things Gone Wrong

Things that went right were the different types of planning and organization for testing. Using the input domain analysis and control flow graphs allowed for writing tests to be much simpler. This is because it showed better the different expected behaviors and paths that the code could take.

Things that went wrong with this lab was the time to get the lab finished. This lab was scheduled around spring break so there was less time to work on it at school. This made the lab more difficult to finish on time.

Conclusion

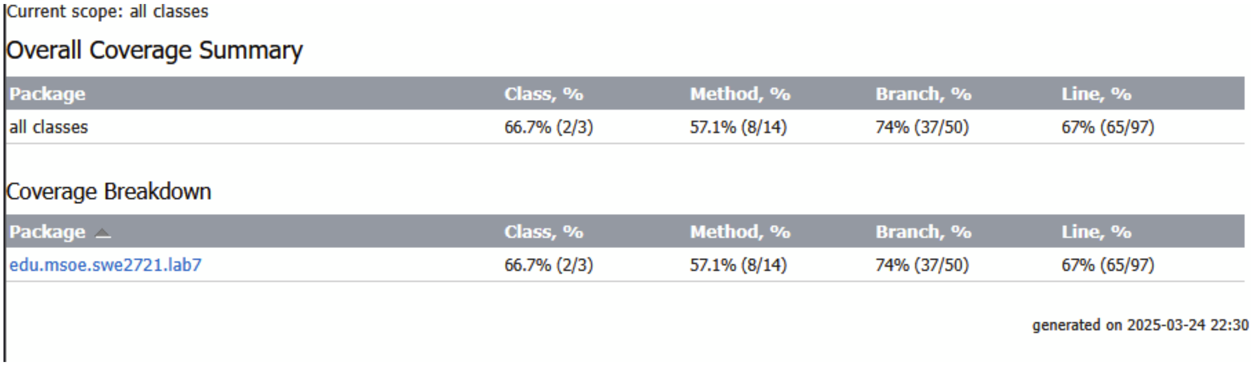
What have you learned from this experience?

From this experience we have learned how to practically use cyclomatic complexity and prime path testing with given code. We had learned about it previously in lecture, but this is the most we have put it into practice with code. This allowed us to learn about how this type of preparation for testing can show the different paths and interactions between methods to test edge cases and create test cases for several different inputs.

What improvements can be made in this experience in the future?

An improvement that could be made to this experience in the future would be to have the lab scheduled for a different week. With it being a partner lab, it was more difficult to work as partners outside of school being spring break.

Appendix



Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %
all classes	66.7% (2/3)	57.1% (8/14)	74% (37/50)	67% (65/97)

Coverage Breakdown

Package ▲	Class, %	Method, %	Branch, %	Line, %
edu.msoe.swe2721.lab7	66.7% (2/3)	57.1% (8/14)	74% (37/50)	67% (65/97)

generated on 2025-03-24 22:30