Names: Madison Betz and Kaiden Pollesch

Date: 4/15/2025

Assignment Title: Lab 10 Logic Cover and Refactoring

Repository Link: https://github.com/msoe-SWE2721/2025-swe2721-lab-10-gilded-rose-b-121_betz_pollesch_lab10_2025.git

Introduction

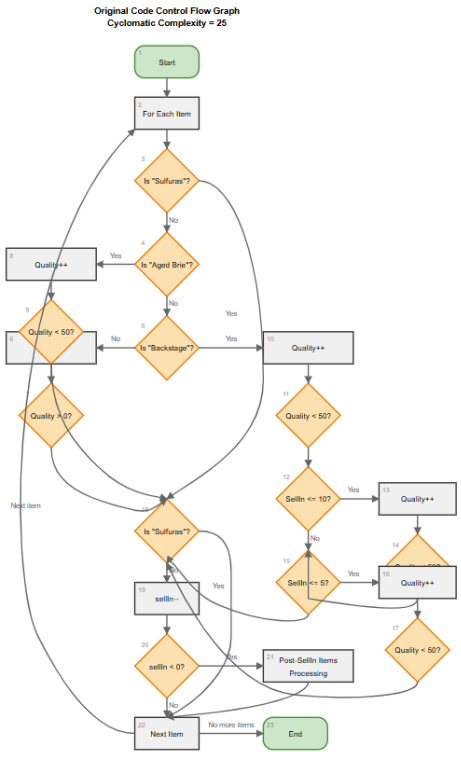
What are you trying to accomplish with this lab?

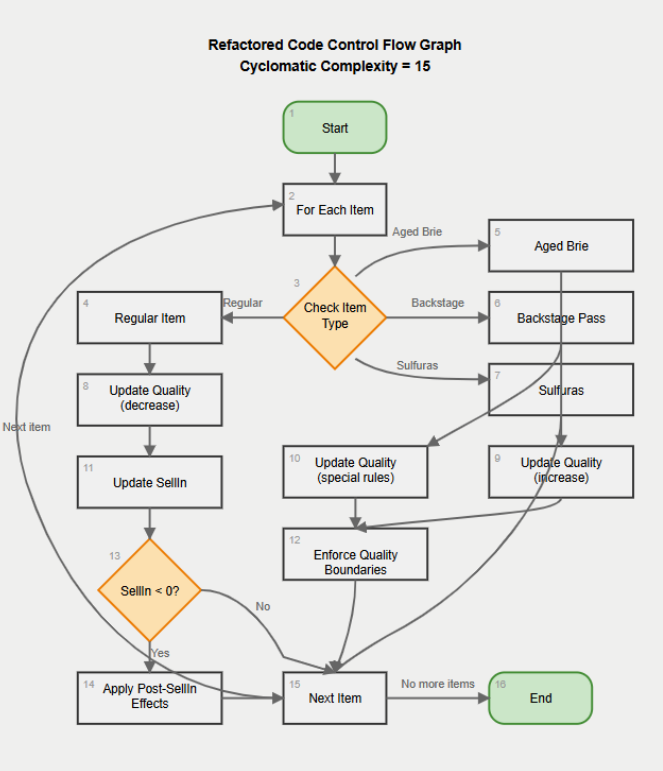
In this lab, we are practicing code refactoring techniques while maintaining program functionality through regression testing. Working with the Gilded Rose code base, we are developing test cases using logic coverage criteria, specifically MCDC (Modified Condition/Decision Coverage), and then using these tests to verify that our refactored code maintains the same behavior as the original implementation. This lab demonstrates the importance of having effective test coverage when performing refactoring activities, as well as the challenges of working with poorly documented legacy code.

Code Coverage Snapshots

Element	Class, %	Method, %	Line, %	Branch, %
edu.msoe.swe2721.lab10	50% (2/4)	61% (8/13)	75% (59/78)	79% (49/62)
GildedRose	0% (0/1)	0% (0/2)	0% (0/4)	0% (0/2)
Item	100% (1/1)	66% (4/6)	94% (32/34)	85% (29/34)
RefactoredItem	100% (1/1)	100% (4/4)	100% (27/27)	100% (20/20)
SampleMain	0% (0/1)	0% (0/1)	0% (0/13)	0% (0/6)

Control Flow Graph





Bug Listing

There were no bugs found in the initial code.
Due to careful coding and refactoring, we did not encounter bugs in the refactored code that we worked on.

Table of Program Statement Clauses and Test Case Values

Test Case	Item Name	Initial SellIn	Initial Quality	Expected SellIn	Expected Quality	Conditions Covered
1	"Regular Item"	10	20	9	19	Standard item decrease
2	"Aged Brie"	10	20	9	21	Quality increases for Aged Brie
3	"Backstage passes to a TAFKAL80ETC concert"	15	20	14	21	Backstage passes basic increase
4	"Backstage passes to a TAFKAL80ETC concert"	10	20	9	22	Backstage passes <11 days
5	"Backstage passes to a TAFKAL80ETC concert"	5	20	4	23	Backstage passes <6 days
6	"Backstage passes to a TAFKAL80ETC concert"	0	20	-1	0	Backstage passes after concert
7	"Sulfuras, Hand of Ragnaros"	10	80	10	80	Legendary item no change
8	"Regular Item"	0	20	-1	18	Double decrease after sell-by
9	"Aged Brie"	0	20	-1	22	Brie double increase after sell-by
10	"Regular Item"	10	0	9	0	Quality never negative
11	"Aged Brie"	10	50	9	50	Quality never above 50
12	"Backstage passes to a TAFKAL80ETC concert"	10	50	9	50	Quality maxed at 50

Discussion

As a tester, how difficult was this approach to define test cases?
Creating test cases using logic coverage was somewhat simple. The main challenges included considering the different cases. However, this approach was beneficial because it tested the different pieces of the code to verify their functionality. The complex decision logic in some methods made logic coverage particularly appropriate because of the logic within the method.

If you had tried to use pure input domain analysis on this problem, how effective would your test cases have been? Did this code base, as it was written and described, lend itself to using IDA?
If we had tried to use pure input domain analysis on this problem, our test cases would have been less effective because the behavior of the system depends heavily on the specific item names and their special rules, not just on the range of possible input values. The code base did not lend itself well to using IDA because the boundaries between different behaviors weren't based on ranges of numeric values but on categorical differences (item types) and complex interactions between multiple fields.

As a tester, how difficult was it to write tests without having access to a good specification?
Writing tests without having access to a good specification was difficult because it required reverse-engineering the requirements from the code itself and the provided sample output. We had to carefully analyze the execution pattern of different items to understand the intended behavior. The sample program output provided some guidance, but it did not cover all edge cases or explicitly state all the rules.

As a developer, how difficult was it to refactor this code given the quality of the documentation?
Refactoring the code given the quality of the documentation was challenging because it required being extremely careful not to change any of the existing behavior, even when that behavior seemed illogical or inconsistent. The lack of clear specifications meant that we had to treat the existing code as the definitive source of truth, even when its intent wasn't obvious.

As a developer, how difficult was it to ensure that you were not breaking the program as you made your changes?
Ensuring that we were not breaking the program as we made changes was critically important, and the comprehensive test suite developed using MCDC coverage was essential for this task. The regression tests provided immediate feedback whenever a change altered the behavior of the system, allowing us to quickly identify and fix issues.

As a developer, could you have been effective at performing this refactoring if you did not have the tests that the tester developed for you?
We could not have been effective at performing this refactoring if we did not have the tests that the tester developed because the complex business logic and numerous special cases would have

made it nearly impossible to verify that all behaviors were preserved. Having comprehensive tests that covered the decision logic was crucial for providing quick feedback on whether changes maintained the correct behavior.

Things Gone Right / Things Gone Wrong

Things that went right in this lab were using the MCDC test coverage approach to successfully identify test cases that covered all branches in the original code, providing a solid foundation for refactoring. Another thing that went right in this lab was that the refactored code significantly reduced cyclomatic complexity while maintaining all the original functionality, making it more maintainable if there were to be future changes.
Things that went wrong in this lab were figuring out how some of the rules for how quality should change in certain scenarios, leading to test failures that required additional analysis to resolve. Another challenge was with edge cases, particularly around the quality bounds, because of test coverage needs being initially missed. Also, the refactoring process took longer than expected due to the need to carefully preserve all the existing behaviors, even when they seemed inconsistent or illogical.

Conclusion

What have you learned from this experience?
From this lab experience, we learned about the critical importance of comprehensive testing when refactoring legacy code. We also learned how much easier maintenance becomes when code is structured clearly, even if the underlying rules and functions are complex. We gained practical experience with using test coverage as a gude for refactorin, which will be valuable for future software development work.

What improvements can be made in this experience in the future?
For future iterations of this lab, we suggest providing a more comprehensive set of test cases as examples to guide in developing test suites. It would also be beneficial to have clearer instructions on how to approach the refactoring process, maybe with suggestions for different refactoring strategies that could be applied. Adding a requirement to document the refactored code would also be beneficial for making people show what they have found in the original code and the new structure of the refactored code.