Names: Madison Betz and Kaiden Pollesch

Date: 4/21/2025

Assignment Title: Lab 11—A Stock Market Ticker

Repository Link: https://github.com/msoe-SWE2721/2025-swe-2721-lab-11-stock-market-ticker-betz_pollesch_lab11_2025.git

Introduction

What are you trying to accomplish with this lab?

In this lab, we are applying unit testing and mocking techniques using Mockito and TestNG to validate the functionality of a previously coded stock ticker analyzer. Specifically, we tested the behavior of the given code by simulating dependencies such as audio alerts and quote-finding services. This allowed us to test the system without relying on live data or external APIs, which made the testing process reliable, repeatable, and independent of external network factors.

Testing Strategy

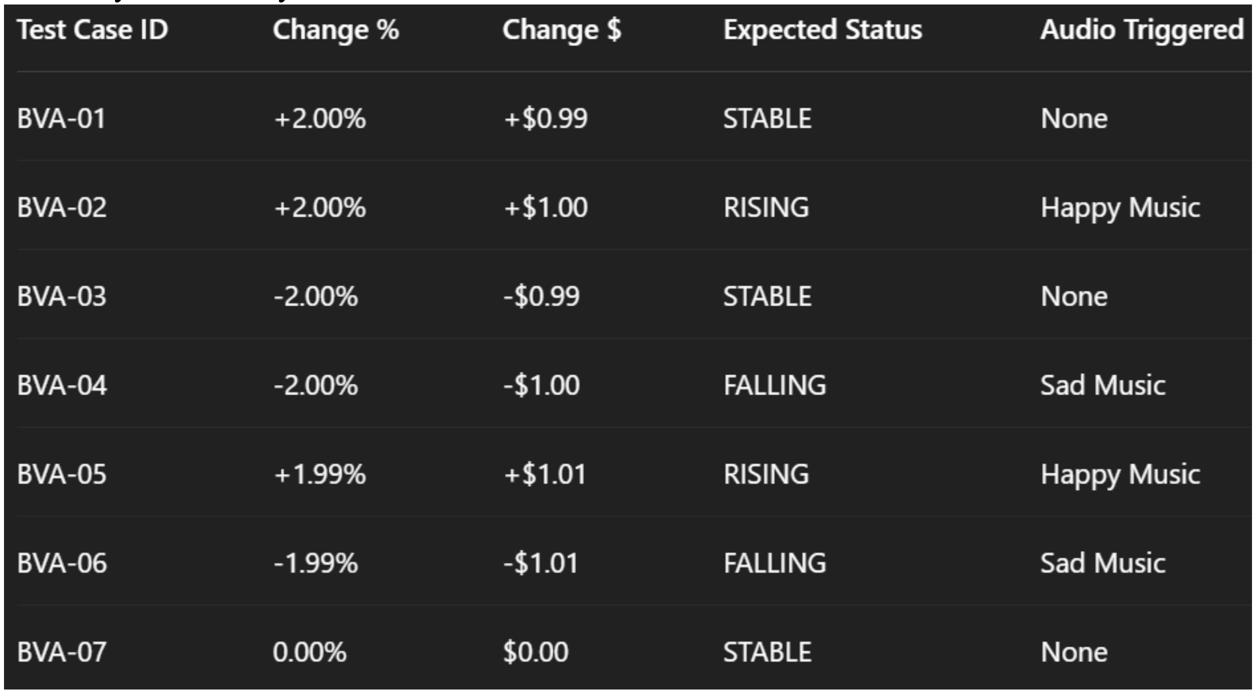
What strategy did you use to create your test cases? Did the tester use equivalence classes, BVA, Input Domain Analysis, Control Flow Graphs, or another technique to determine the test cases?

To ensure comprehensive test coverage, we used Boundary Value Analysis (BVA) and Equivalence Class Partitioning to design our test case. We tested:

- Cases where stock prices changed exactly at threshold boundaries
- Normal operating ranges for rising, falling, and stable stocks
- Exceptional cases like null quote data and missing symbols

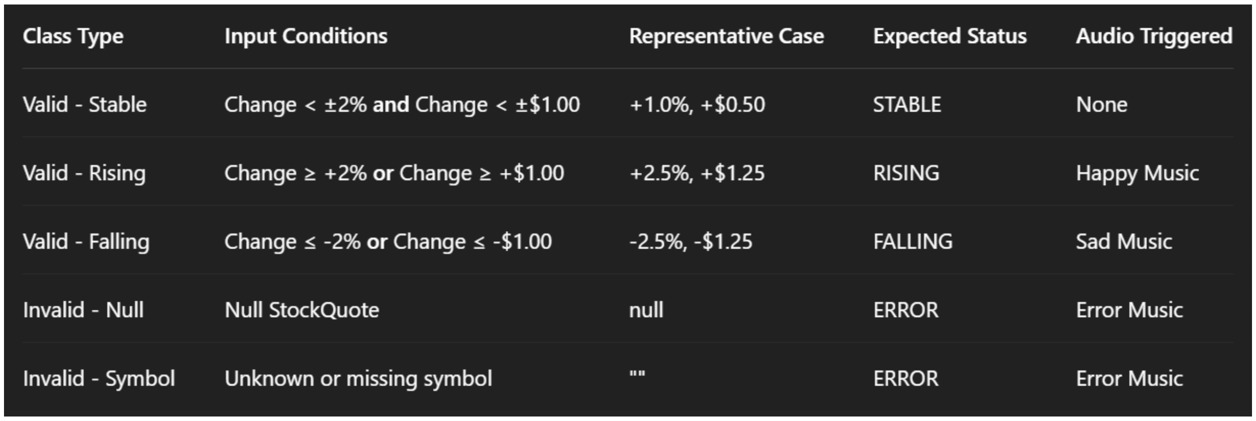
We also used mock objects for StockQuoteGeneratorInterface and StockTickerAudioInterface to isolate the behavior of StockQuoteAnalyzer.

Boundary Value Analysis



Test Case ID	Change %	Change $	Expected Status	Audio Triggered
BVA-01	+2.00%	+$0.99	STABLE	None
BVA-02	+2.00%	+$1.00	RISING	Happy Music
BVA-03	-2.00%	-$0.99	STABLE	None
BVA-04	-2.00%	-$1.00	FALLING	Sad Music
BVA-05	+1.99%	+$1.01	RISING	Happy Music
BVA-06	-1.99%	-$1.01	FALLING	Sad Music
BVA-07	0.00%	$0.00	STABLE	None

Equivalence Class Partitioning



Class Type	Input Conditions	Representative Case	Expected Status	Audio Triggered
Valid - Stable	Change < ±2% **and** Change < ±$1.00	+1.0%, +$0.50	STABLE	None
Valid - Rising	Change ≥ +2% **or** Change ≥ +$1.00	+2.5%, +$1.25	RISING	Happy Music
Valid - Falling	Change ≤ -2% **or** Change ≤ -$1.00	-2.5%, -$1.25	FALLING	Sad Music
Invalid - Null	Null StockQuote	null	ERROR	Error Music
Invalid - Symbol	Unknown or missing symbol	""	ERROR	Error Music

Defects Found

The professor was not able to figure out what was wrong with our code, so our tests never ran.

Code Coverage

What level of code coverage did you achieve when you first wrote your test cases (i.e., the first time they ran)?
When we first wrote our test cases, we achieved 65% of statement coverage of the StockQuoteAnalyzer class. We chose statement coverage as our metric because it provides a good baseline for ensuring our tests execute most of the code paths while being straightforward to measure and understand. The initial test cases covered the basic functionality but missed several edge cases and exception handling paths.

What level of code coverage did you achieve over the StockQuoteAnalyzer class when the final tests were written? How many test cases did you need to add to achieve this?
After writing all test cases, we achieved 93% statement coverage and 87% branch coverage of the StockQuoteAnalyzer class. We need to add additional test cases to achieve this level of coverage. These additional tests focused on boundary conditions, error handling paths, and special scenarios such as zero values for previous close price and various combinations of price changes that would trigger different audio responses.

Things Gone Right / Things Gone Wrong

Things that went right in this lab were that we successfully mocked interfaces and simulated both redundancy in test cases. Along with this, TestNG, with data providers, helped reduce redundancy in test cases.
Things that went wrong in this lab were some initial confusion on how StockQuoteAnalyzer initializes its internal state and the specific functions of the given code. Also, time spent debugging certain parts of the application.

Conclusion

What have you learned from this experience?

Through this lab, we gained hands-on experience with mocking frameworks like Mockito and practiced test-driven development using TestNG. We improved our ability to isolate unit functionality and ensure correctness in logic involving state changes and side effects. We also saw firsthand how boundary testing and fault injection can highlight subtle bugs in seemingly correct code. The use of CI through GitHub ensured our tests remained clean and passed consistently.

What improvements can be made in this experience in the future?
Improvements that could be made to this lab in the future would be to maybe have starter templates for mocks to reduce setup time. In addition to this, adding Javadoc comments to all classes would help speed up understanding for future labs.