Ticket Machine Web App Report

Tutor: Craig Gallen

Unit: COM 528

Jacob Dale

# Contents

[Contents 2](#_Toc60823448)

[Decisions Made During Creation of Diagrams 3](#_Toc60823449)

[Where the Code doesn’t Match the Design 3](#_Toc60823450)

[Rationale for Test Strategy 3](#_Toc60823451)

[Critical Evaluation of Code Implementation and Design 3](#_Toc60823452)

# Decisions Made During Creation of Diagrams

I decided to create my diagrams after I had implemented the requirements into the project and written a test document for the project. I felt this was the most efficient use of my time as it meant I had a better understanding of the functionality when creating the diagrams.

The use cases were simple to design as I just had to put myself in the shoes of the customer or station manager and write a brief set of instructions on how they would go about performing their tasks. I could also the test documents as a reference for what each user does.

I unfortunately run out of time to create the robustness diagram.

# Where the Code doesn’t Match the Design

The Class diagram I inherited form this project is almost completely unaltered. I did consider adding in all the methods for each of classes, but as time was short in this sprint, I decided my efforts were better spent developing the code and use case diagrams. I did add the id field to the Ticket class however, as that was a requirement that all tickets have a unique ID.

I haven’t altered the backend of the code that would alter the class diagram therefore I couldn’t add to the diagram. There is a new class I created called VariableStorage.java which I implemented as a work around for a null exception I encountered developing TicketMachineRestService.java. The null exception was being generated because serviceFacade.getPriceCalculatorDAO() was returning null, the variable storage was designed to simply store a PriceCalcualtorDAO object so the time schedules could be edited. It is a work around and not a fix that needs to be replaced with a proper fix. Unfortunately, time was not on my side therefore the fix will need to be pushed to a future sprint.

# Rationale for Test Strategy

The test strategy I created for this piece of work has a simplified format of the one I use for testing at work. It has a basic design because the tester would usually be someone who doesn’t know what the code behind the app looks like. The tester might not even have a working knowledge of programming in Java or any other language so, I tried to make it as easy to understand as possible. I am concerned that the language I used may be too informal for a real-world test. That would be addressed in the next version of the document.

Instead of writing the test document against the use cases I have designed, I wrote the test as if I was a user using the system and checked I had tested against all the software requirements once the first version was complete.

I tried to work in an agile manner, which allowed me to write my test document before creating the use cases. I feel that this helped me when creating my use cases as it gave me the experience with what the app needed to do.

# Critical Evaluation of Code Implementation and Design

Overall, I am content with the implementation of the code. I am aware there are many improvements that can be made to improve the efficiency and user friendliness of the app. One of the big improvements I would make is to abstract the embedded Java code away from the JSPs. This would decouple the app a little which - in theory - should improve the scalability, help reduce development time in future and make the app more understandable for fresh eyes. I am seeing 100+ lines of embedded Java a little daunting.

Abstracting the Java form the JSPs would ease unit testing because if I wanted to unit test the current embed Java I would have to copy and paste it into a unit test and every time that code gets a slight change, the unit test would or it would become invalid. If the code was abstracted all I would have to do is reference the corresponding classes and test their function. Therefore, I have not implemented many unit tests; I received the project with unit tests already written to test the backend of the app. In future versions I would add to these unit tests as I got a better understanding of the backend. The test procedure should cover for the lack of frontend unit tests making it an unworthy investment of time to write said front end unit tests. In a future sprint abstracting the Java and creating frontend unit tests would be the top priority, I would use Test Driven Development to ensure it.

I try to follow the SOLID principles (Single responsibility, Open–closed, Liskov substitution Interface segregation, Dependency inversion (injection)). Unfortunately, in this project I broke just about all those principles. Creating a solution using SOLID should make software designs more understandable, flexible, and maintainable.

One of the requirements has been not implemented properly and that is the scheduling for the ticket machines price and peak times. I put it in the client side of the app, which was my first mistake. My second was leaving it there and not trying to implement it in the server side. I tried to change it after my demonstration as suggested, however I couldn’t get it to work. I tried to implement it by creating of PriceCalcDAOJpaImpl and creating queries for the inherited methods from the interface. For fear of not being able to implement it in time before the hand in deadline, I reverted my code to the demo stage and started adding validity checks for openGate.jsp.

As for the design, there wasn’t much for me to add to the class diagram as I only added JSPs and edited pre-existing functions. The Use cases were simple to create as I could use my test document as a guide for what a user would do in the app.