John Oldfield

Assessment discussion

Contents

[Drawing of Domain Model and Use-Case Texts 2](#_Toc472365575)

[Drawing of Robustness Diagrams 2](#_Toc472365576)

[Changes to Domain Model and Use-Case Texts 2](#_Toc472365577)

[Drawing of sequence diagrams 2](#_Toc472365578)

[Mismatches between code and design 2](#_Toc472365579)

[Testing rationale and plan 3](#_Toc472365580)

[Evaluation of code and design 3](#_Toc472365581)

[Code 3](#_Toc472365582)

[Design 4](#_Toc472365583)

# Drawing of Domain Model and Use-Case Texts

When drawing the domain model, I looked carefully at the scenario given to me and highlighted any nouns. I then narrowed them down to the essential entities which I used as classes. I then elaborated on these classes thinking of how many objects interacted with them and annotated this with multiplicity. I decided on JJT, Agent and Event as being the main classes. JJT would store the Agent and Event details, and Agent and Event classes would store the agent and event objects. I did think about making ticket a separate class but I decided it wasn’t necessary as the system is more focussed on the sales of the agents rather that individual tickets.

For the use case texts, I put myself into the shoes as a user of the system and decided what steps they perform using a system from their perspective. By doing this I could deduce that in this situation the Agents are actors. I could then write the use cases themselves by deciding which tasks the actors would need to perform when using the system and the method they would do it. I wrote the use cases in a method that would allow for speedier data input whilst ensuring the system was ‘idiot proof’. An example of how I did was this is incorporating the use of sliders and choice boxes.

# Drawing of Robustness Diagrams

I found that there would be a lot of checking of data required when using the system and from this the validation object was founded. This would later be used for the forms involving agents as they need to be validated when creating and modifying agents and I believed this was a necessary implementation.

# Changes to Domain Model and Use-Case Texts

After drawing up the robustness diagrams I realised I would need a validation class to perform some of the tasks related to the agent. A lot of the forms tasks required validation at certain points and figured this warranted a separate class. I added this to the domain model and took the validation class into account when writing the use case texts.

# Drawing of sequence diagrams

When drawing the sequence diagrams, I paid close attention to the use case texts and made sure to elaborate on the points I had made. I made sure that I was detailed in my steps including any parameters and functions I would use for processing a task. I made sure to think about any situations the agents may find themselves in a logical method so that I could think of the process and how it would take place. Taking into account that I have never done a sequence diagram before I feel I did an adequate job of outlining the sequence of events.

# Mismatches between code and design

Whilst my system is built around the tasks set in the scenario I have added a lot of extra features and functionality where I thought would be appropriate. Therefore, my system is not limited to the tasks set in the brief and contains extra features not taught in class such as file handling.

Whilst I was implementing the system I realised that separate dialogs for errors and confirmations were unnecessary and would only make it longer to input data as well as coding the system. Because of this I decided to use labels and texts to display errors, confirmations and alerts as it practically does this the same thing but would save the Agents time when using the system.

When I was designing the system I initially designed the system to only allow the agents to sell for a single event, however my finished system includes 3. In addition to this the functions initially written/designed are not the same to the finished ones as they had to be adapted to include 3 events, however they share the same principles For example the setSales function has been slightly changed to setSalesForGD, setSalesForCF and setSalesForB182, one for each event. Whilst I was implementing the system I decided it would be more beneficial to include a few more events to make the scenario more realistic. It also had the extra advantage when it came to testing as I had a range of event objects to test.

# Testing rationale and plan

To test my system, I decided to use a combination of black and white box testing to test both the specific code as well as the functionality of the system. I performed JUnit tests where possible to check that all my functions worked and returned expected results. This benefitted me as I could test all the functions at once and would be able to see which ones failed and why. From my JUnit tests, I could see that one of my tests failed, which given the time I could have fixed and made my application even better. I also

# Evaluation of code and design

## Code

I think what makes my program high quality is the level of detail I have gone into with many of the applications features and functionality. I have implemented all the activities asked of me but also kept in mind the usability and aesthetics of the application. As part of this I decided to style my application with CSS which wasn’t necessary but I feel it put a user-friendly touch on my system. I also added lots of little features to spruce up the application such as colour coding and providing descriptions upon hovering over a button.

The first thing that stands out is the log in screen and subsequent list of active agents on the menu that I implemented as a way of telling if a user is active or not. A login screen is a more realistic and sensible option for this type of application and used in many of today’s standard software for organisations like JJT Ltd. This took a lot of code to implement and used a variety of Java skills and a lot of problem solving.

The hardest thing for me to implement was the saving to text files as this required some additional learning to know how to get it to work the way I wanted. I am glad I managed to do it though as it helped me get better at coding Java.

One flaw with the way I have coded this application is the lack of functions I have used. A lot of processes were all done within one function rather than being split up into many smaller ones. This led to me being unable to perform as much JUnit testing as I would have liked meaning there may be potential errors which simple testing methods were unable to find.

Another problem I found with my code was it wasn’t completely error free as the testing proved that I hadn’t implemented validation to check that agent sales couldn’t go below 0. This was a problem when cancelling tickets for an event where the agent hadn’t sold enough ticket to remove all the sales from him as doing so would result in him having a negative number of sales. In the future I, will attempt to test whilst I am implementing so that I can have time to fix the errors on completion of the application.

## Design

I think the best aspect of my designing was my class diagram which appropriately showed the attributes and functions that each of my classes should contain. Because I had drawn it so well I found that when I made the decision to add more events it was easy to do as the functions were very similar and sometimes even just duplicated. I feel like I could have improved my sequence diagrams and subsequent use case texts more making each step more specific and by going into further detail.

For the most part I felt my design was justifiable and well thought out however a problem I found when coding my designs came when creating an agent. I had designed it so that agent objects would be uniquely identified with an ID like most software system however the way I processed giving them an ID was the problem. Whilst counting how many agent objects there were and adding one did give a newly created Agent a unique identifier if the Agent is deleted this resulted in agents having the same ID. Unfortunately, I didn’t realise this problem until it came to testing and therefore didn’t have the time to implement a solution. Should I do this again in the future I will make sure to look out for this problem and solve it at the design stage with the use of HashMaps or by adding more validation to ensure duplicate IDs were not possible.