Results Document

This assignment demonstrated the real power of “Design first agile”. Most of the heavy lifting was done during the initial brainstorming and peer review sessions that the implementation went smoothly with no surprises. Integrating the three services didn’t pose much problems as the services had clearly defined boundaries formally known as the bounded context in domain driven design. I initially faced some issues hooking up the Store Controller Service and the Store Model Service like the chicken and egg problem. SCS had to register itself to be able to listen to events but needed to listen to the register-controller event to be able to register itself. The workaround was to have some bootstrap mechanism to initialize the ledger service and SCS before further commands that involve those services are executed.

I have a command factory class that parses events and initializes the appropriate command setting whatever state is needed to execute actions. I also implemented a queuing mechanism for storing commands that are generated for later execution. There is a specific event that forces commands in the queue to execute and to clear the queue. This assignment was command pattern, observer pattern, factory pattern and multithreading in action.

The code is heavily commented with java doc. Tiny bits of implementation details added for a greater customer experience are defined in the class dictionary as well as in comments. The logs generated by the test script turned out to be enormous. IntelliJ was truncating the logs, so I output the logs to a file. I also removed the extra clutter from the logs using a text editor to make them more legible. The tests were also extensive and were testing every possible scenario I could think of without getting too caught up in minute details. I have learnt from this assignment that there is a fine line between too little and too much details.

I can only imagine how difficult this would have been if it was not well thought out and I immediately started coding. There is also no such thing as a correct design which led to arguments with my peer review partners. The peer review has helped me consider other approaches but I decided to stick with my initial design in some of the details.

To execute the program

javac com/cscie97/ledger/*.java com/cscie97/ledger/test/*.java

javac com/cscie97/store/controller/*.java

javac com/cscie97/store/model/*.java com/cscie97/store/model/test/*.java

java -cp . com.cscie97.store.model.test.TestDriver store.script

Peer Review

My reviewers were Derek Kinzo and Taylor Meyer. There was some constructive feedback given to me during the peer review that I incorporated in my design. There were also some suggestions I didn't agree with and kept my initial design.

Derek's suggestion I agreed to

1. Try avoiding crossing lines in the diagram. This was a good suggestion and added clarity
2. It is not clear how the factory generates commands in the diagram. This suggestion I partially incorporated (My understanding) The factory does not have an association relationship with any of the commands but a dependency. I wrote notes about this in the class dictionary but kept the dependency relationship in the class diagram.

Derek's suggestion I didn't agree to

1. Since the patterns are obvious, you don't need to explicitly state them. (My understanding) Even though the patterns are obvious, stating them explicitly removes some communication barriers. Developers familiar with the pattern can get into an agreement quickly and can decide if the pattern fits nicely to the problem at hand.
2. AbstractCommand class is unnecessary and you should favor composition over inheritance. (My understanding) There was some default behavior common to all the commands. For example, most of the commands operate on Store Model Service and Ledger instances. Defining that common behavior with inheritance made sense. There was no undesired behavior the commands were inheriting. AbstractCommand is like a lazy adapter that defines associations to the external API. There maybe circumstances where the ledger service is not involved like when a robot assists customer, but the benefit outweighs the drawbacks. I didn't want to tightly couple each concrete implementation of the command with SMS and ledger as well.

Taylor's suggestion

"Thank you for your feedback, and that is correct! In return I think you did a great job with your class diagram. One piece of feedback revolves around the device Command (AbstractCommand) and its relationship with ModelService (IStoreModelService). It may have been beneficial if the Event Handler dealt with the ModelService (IStoreModelService), and then a 'device command' created to provide instruction to the device."

This was a good feedback, but I tried to keep it simple for the purposes of this implementation. It will be something I will look into when I don't have time constraints.

My suggestion to Taylor

I like your approach of relying on interfaces but it is not clear how the Store Controller Service gets notified when an event is triggered. Also my approach was to have a generic event that gets interpreted to a command. Are you having custom events and a corresponding listener that is interested only in this particular event ignoring the others?

Taylor's design had specific events sent out from the Store Model Service that each command from the Store Controller Service will act up on. My suggestion was to have a generic event and delegate the parsing to the Store Controller Service. My understanding was he would need to change the Store Model Service significantly to tailor his design.

My suggestion to Derek

"That is actually a factory pattern used to recognize the events and generate the commands. I have since renamed it from "ControllerEvent" to "CommandFactory" so it'd make more sense for the reader... I also noticed I had the dependency lines wrong. I fixed that as well to be in between the ICommands and Controller classes."

Derek's suggestion closely resembled mine besides what we disagreed on. He cleared up some of the confusion with naming according to his response above.

Final Notes

It looks like I am benefitting a lot from these assignments. As we are progressing, I am witnessing my designing skills greatly improve. I had many surprises when I started coding in assignment 2 but the surprises were much less this time in assignment 3. One suggestion I have is to throw in as many design patterns as possible into the assignments so that we can have some hands-on experience with most of gang of four design patterns.