Matthew Thomas
CSCI e97 — Assignment 3 — Results

Did the design document make the implementation easier?
The design document helped make the implementation easier. I found, going from assignment 2 to 3, that the diagrams in particular have been the most helpful in terms of implementation. Seeing how the system as a whole is structured helps me grasp the fundamentals of what the software is supposed to do, which, in turn, helps me to figure out the use cases and requirements. In understanding that, I am better able to add more detail to the diagrams.

Given my experience on assignment 2, I found creating the design document to be easier overall this time around. In my initial design, I overlooked several details, but I was able to correct that in my submitted version. Because the design was more complete, and I felt more confident in the design for this assignment, I feel like the implementation was also a lot easier. It felt similar to assignment 1 where we implemented the design provided by the course staff. When questions came up during implementation, I was able to find more answers by looking back to my design document and referencing what I had written, rather than having to think about an answer to include in the design and the implementation.

How could the design have been better, clearer, or made the implementation easier?
I think the design could have made the implementation easier if it included more of the "setup" configuration. I feel I got a handle on a single-system design from assignment 2, but this assignment introduced more inter-connectivity and therefore more complexity. While the component diagram (from the System Architecture document) helped me work through the problem, it still involved a bit of conceptual work for me to understand how the pieces all fit together — "Where do I instantiate these objects and how do I reference them?".

A specific example of where my design could have been clearer is the individual Command classes. My original draft was less detailed in that area because I did not fully grasp how a Command would interact with the Store Model. The design just passed in the source Device to most Commands (some passed a Customer object too) and my thinking was the Command could glean most of the information from that Device. Working through the sequence diagrams, and later the implementation, I realized I needed to pass more information because of how the Command interacts with the Store Model Service. For future designs, I can see creating more diagrams as well as different types of diagrams (like sequence diagrams) to make sure I'm thinking about all parts of the system — both an individual module and how the modules interact.

Did the design review help improve your design?
I found one comment from Antony to be particularly helpful. Antony touched on a key interaction in the system that I had skimmed over in the design, and that I was poorly

implementing in practice. Antony correctly noticed that my design skimmed over how access between the Store Controller, Model, and Ledger was handled. That comment, along with Piazza post @226, helped to clarify my thinking and more clearly demonstrate how the components maintained the core class idea of separation of concerns. It was nice to catch this early on the implementation phase thanks to the peer review.

How did you find the integration of the components?
This was the first assignment and experience I have had where I integrated several components and I found the experience quite fun. The class diagrams and patterns were incredibly helpful in the process in that they clearly defined "where" the pieces should go. Also, having the public API for both the Store Model Service and Ledger Service ready to go from assignments 1 and 2 was incredibly helpful.

Changes to your design based on the peer design review or implementation.
I did not make any major changes to my design based on the peer design review. As mentioned above, the peer design review pointed out one area of my design that was missing explicit associations or class properties, which I added, but the general structure remained the same.

A big change that I made after I began implementation was to include the Factory Pattern from the *Head First Design* book. Creating all the individual command classes I realized there was a lot of overlap between them, and passing a Device and StoreModelService into each one got redundant. Through implementing the Factory Pattern, I included an AbstractCommand class which dealt with a lot of common properties and methods and left the individual Command classes to implement the execute() method to satisfy each of their requirements.

Another change was adding the RestockCommand. During implementation I realized that the steps needed to command a Robot to restock the shelf could be abstracted into its own command, such that it could be invoked separate from a BasketEventCommand. While this was the only additional command I created, this concept follows the Command Pattern closely and lends itself to easy expansion via abstraction.

A final change, as mentioned in Piazza post @233, is that the Ledger retrieves account balances based on the last committed block. Each Block contained 10 transactions in assignment 1, which could lead to inaccurate readings if a Customer requested their balance and was told they could cover their basket, but at checkout was told they could not leave because their "actual" balance included transactions not yet committed to the blockchain. To solve this problem, I went with one of Eric's suggestions and changed the block size to store only 1 transaction, and therefore always reflect the most up-to-date Customer account balance.

Comments about your design from your peer design review partners.
Antony Gavidia's peer design review comments:

Hi Matthew,
thank you for our feedback! it was really good. I will replay here to foment some discussion (if necessary). Just wanted to point out something about my design:

I followed the blue print given by the professor for the design of the Ledger, but I completely deviated for the design of the StoreModelService. I am a functional programmer, so I dislike side effects (basically functions/methods with "void"). I ended up doing a functional implementation of the StoreModelService that doesn't mutate or hold the state anywhere. Now, given that my Ledger is stateful, my StoreModelService is functional, and the observer pattern forces you to implement something stateful, I have ended up with something of a chimera hahaha. Anyway, I have finished with the implementation and somehow it all runs well (although not the most elegant implementation under any schema, not the most horrible by any principle either). My implementation of Controller is also functional, it doesn't mutate/modify anything (except for the logging function).

As for your main question, what is List<String> (returned by the update method), it is a list of commands (store model DSL commands) to be processed by the store model service. The signature for the update method is:

public List<String> update(String event, StoreStateSuperType state)

So the controller receives the event, a copy of the store state at that time, and returns back a list of DS commands. If I had "void" instead the controller wouldn't be able to do absolutely anything given that the store model doesn't have any state to modify (not even a way to grab any state since it is not stored anywhere). Sorry if this was a bit confusing. Again, I had no choice but to make the best I could with what I already had.

Now, regarding your implementation, I think it is perfect! It seems your are following the patterns explained in the book exactly as they are supposed to be used. I can't really criticize anything, only I have a question that I don't get, how does the controller have access to the state of ledger and the store model? the observer (the controller) registers with a Subject, not with the StoreModelService, so it doesn't have access to all its methods; or are you just passing the Ledger and Service object in the constructor of the Controller? Or perhaps are you doing some casting somewhere? Please let me know, it might help me in my own designs in the future. Thank you!

Trisha Singh's peer design review comments:
Hi Matthew,

Thanks a lot for your comments, I finally caught up with all the lecture videos today so I can see how your comments tie up with the lectures and I was pretty off track w.r.t depicting the Command Pattern in the class diagram.

Regarding your question about how Events are implemented, I was surprised to see the prof discuss a similar idea in last week's lecture - I had exactly the same idea in mind when coming up with the design.

Feedback for your design:
- Like Anthony, I really can't find any faults with your class diagram.
- I'm not sure if its necessary to include but I see your overall design document is missing the list of commands that can be sent in via the command processor. I saw that in the LedgerService design document so I like to include it too.

Thanks,
Trisha

Comments provided by you for each of your peer design review partners.

My comments on Antony Gavidia's design:

Hi Antony,

I think your class diagram is a good start. I have a few questions and areas to think about, some of which you may already have answers to, but I was unclear about without seeing a class dictionary or more detailed explanations.

The biggest point I noticed was the connections between the Controller, Model, and Ledger. I would reference the System Architecture document that was provided in assignment 2. From what I understand, the controller depends on the Model Service and Ledger, though I see that you have a dependency between the Model Service and Ledger. Similarly, you have a getLedger() method in the Subject interface which, if you modify the associations, wouldn't need to exist. On the flip side, if you maintain your current design, I'm not sure why the Subject would include that method, compared to the Store Model Service itself. Requiring a Subject to know about the Ledger seems like it would violate the separation of concerns.

Another bit I was confused about was some of the variables and return types in some of your classes. Mainly, why are your notify(), update(), and analyze() methods returning a List<String>, and what are the Strings that list contains? Is it the event string parsed, a status report of how the command handled an event, or something else? Passing the "event : String" as a parameter to notify() and update() makes sense to me, but I don't see what returning a List is meant to do. Down in the CommandBase class, it's not clear to me from the diagram where, or what, the deviceSourceId, header, or token properties come from (you only pass in a StoreStateSuperType and BlockChainLedgerType) or what their purpose is. I get the deviceSourceId, but header and tokens I'm not sure what they do.

A few quick final thoughts. Don't forget about the CommandProcessor class and how that ties in. Including the note about multiple command classes extending the parent is good, but I would Include at least one or two, if not all, of the actual classes you'll be including — and cover all of them in the full class dictionary. And remember to include Exception classes as necessary, I think a StoreControllerException would be a good one, following the pattern of assignments 1 and 2.

If you have any questions or areas you'd like further comments on, let me know.

Good luck on your implementation!
Matthew

My comments on Trisha Singh's design:

Hi Trisha,

I think your class diagram is off to a good start. I have a few questions and areas to think about, some of which you may already have answers to, but I was unclear about without seeing a class dictionary or more detailed explanations.

Your Observer pattern appears to be well structured. I see you're passing a StoreEvent object through the notify() method which is an interesting idea! Without seeing any details about how it would be implemented, I'm not sure if there's any specifics I can give, but I would keep the repeated ideas in class (separation of concerns, etc.) in mind when thinking about this. Does it make sense for the StoreModelService to parse or create this Event, or should this be handled by the Controller? If you do go with a StoreEvent, one specific piece I have is to make sure it connects to something — I don't think floating classes are a good idea, or even possible?, in UML.

It doesn't look like the Command Pattern discussed in class is fully implemented here. You have a floating ApplianceCommand class which is passed to the StoreModelService. I believe you are correct in that there is some association between the Controller/Command and the Store Model Service (it needs Store, Aisle, Customer, etc., information), but the command itself shouldn't be passed over, as far as I know. A Command should have an execute() method as described by the book (and Piazza post @194) which would then obtain/utilize Store information and methods to perform actions).

A final big picture comment is about the scope of the diagram. I was confused a bit myself about what to include and how much in the diagram and the rest of the design document. I saw your post @214 in Piazza about this and the topic was also discussed at the beginning of last night's section (Oct. 22nd). I think an approach more similar to mine or Antony's design which includes the Observer/ Observable and StoreModelService interface is close to the "correct" design. Going all the way down to a Speaker, Turnstile, etc. I think is too much detail for the scope of this assignment. It was also discussed in section grouping the classes into different "packages" in the same diagram to further differentiate between the Controller Service (assignment 3) and the rest.

A few final quick thoughts. Keep in mind the CommandProcessor and Exception classes that you might need. As you work on the Command Pattern more, I would include at least a few of the specific Command classes you'll implement (e.g. EmergencyCommand, CheckoutCommand, etc.) in the diagram, and be sure to mention all of them in the full design document.

If you have any questions or areas you'd like further comments on, let me know.

Good luck on your implementation!
Matthew