Matthew Thomas
CSCI e97 — Assignment 2 — Results

Did creating the design help make the implementation easier?
The design helped a lot in making the implementation easier. Although there were many oversights I caught after writing the design review and some code, creating the design upfront really streamlined the entire process. In particular, the class diagram I modeled in Astah helped me to get a good visualization of how all the pieces of the Store Model Service connected and interacted.

Because elements of the design changed over the course of implementation, having the design and class diagram to reference was helpful. Instead of guessing and checking when I hit a roadblock in coding, I went back to the design and redrew associations, moved operations to different classes, and even just moved the class blocks around to see the picture differently.

How could the design have been better, clearer, or made the implementation easier?
As I mentioned, I did hit several roadblocks along the way. Many of these roadblocks were learning curves as I struggled to think conceptually about how to translate the system architecture and requirements documents into a design. Creating the classes and assigning properties was easy and just a manner of following the requirements. One step past that, however, and it became more work for me to grasp. I found myself asking, “Is this a property? An association? Should this method go with this class or that one?”

My original design had a lot of extraneous associations, duplicate methods, and a jumble of arrows in the class diagram which made for somewhat of a mess. I think the implementation would have been more straightforward had I spent more time on writing the class dictionary and developing the reasons and methodologies behind why I designed a particular class a particular way.

For this assignment, the design document and the implementation both required revisions throughout the process. In particular, I found myself making the most changes to associations, modifying them to fit new challenges. Now that I’ve identified some of my hangups, this knowledge will allow for a cleaner design on future assignments and projects.

Describe implementation changes that you made to your design and how they continue to support the requirements.
Besides the changes I made to my design throughout implementation that I’ve already described here is a list of a few others, most of which are also noted throughout the design document.

- To more closely match the other Store objects, I changed the aisle “number” into an aisle “id”. This made more conceptual sense as “number” does not lend itself to

thinking about a String. Keeping all ids stored as Strings reduces complexity in the implementation by not having to parse arguments and check the conversion. This continues to support the requirements as it is only a name change, and as depicted on the requirements document, page 3, the aisles shown have identifiers of “aisle_1”, “aisle_2”, etc.

- Another name change I made to the Aisle object was to require the location options to be either floor or stock_room (instead of store_room). As this is only a name change, this keeps in the spirit of the requirements. This change helped me conceptually as originally I could not understand why there was a differentiation between a store room and a store floor, only to realize that the “room” was where excess Product supply was stored. stock_room is a much clearer term and reduces the ambiguity for others reviewing requirements or a design.
- The requirements specify on page 4 that the Inventory has an “inventory_id” attribute and a location specified by “store:aisle:shelf”. In my implementation, an Inventory’s location attribute is not stored, but can be imputed by following the “path” from Store -> Inventory. inventory_id’s are unique within shelves, and therefore, given a location string of the form store:aisle:shelf:inventory, the Store Model Service API can find the requested Inventory should it exist. This continues to support the requirements as the location data is maintained through associations and can be assembled by following a given path.
- A final implementation change I made was to remove a separate basket id and just use the Basket’s associated Customer id instead. The Basket maintains an id property, which could prove useful in the future if say, multiple baskets were allowed. For this assignment, this change continues to support the requirements since creation and lookup function the same way — an id is provided and a basket is returned — with the only difference being what the contents of the id are.

Command syntax changes:

- For define aisle, the location portion of the command is changed to
  `location (floor | stock_room)`

- For show inventory, the lookup id is changed from `<inventory_id>` to
  `<store_id>:<aisle_id>:<shelf_id>:<inventory_id>`
- Basket commands have changed the first underscore to a space, such that commands read add… remove… clear… to match more closely other commands in the syntax. Also, the `<basket_id>` should be changed to `<customer_id>`, though as both ids are the same, this change is trivial. Examples:

```
get customer_basket <customer_id>
add basket_item <customer_id> . . .
remove basket_item <customer_id> . . .
```

Did the design review help improve your design?

There were a few really key comments that Stephen and Steven provided in their design reviews that helped improve my design.

My original concept for the Appliance and Sensors was similar to an idea from a class section, which was to have an Appliance be a child to a Sensor. I thought this made sense, since they share some similarities including the fact that both create events. But Steven raised some good points about “inheritance vs. composition” which led me to rethink the design to feature a Device class with Sensor and Appliance as children. This change makes the design much more flexible and would allow new Device objects that would not necessarily be related to a Sensor.

Steven’s comment about my customerList association between Store and Customer and masterCustomerList association between the StoreModelService and Customer also made me rethink my design. His comment made me realize I could just maintain one list via the StoreModelService and that the “customerList” association in the Store object was extraneous. As I mention earlier in this document, this also led me to rethink several other associations I had and how the objects linked together.

Comments about your design from your peer design review partners.

Stephen Thompson's peer design review comments:

> Hi Matthew,
>
> First off, wow. Your documentation is incredible and well detailed. I like how you broke out the actors, and the level of detail that was used to define each of the use cases. The class dictionary is well done as well. There should be no difficulties implementing this design. I also like how the image for the Class Diagram is higher quality and can actually be read when zoomed in.
>
> One suggestion I have would be to see if you can pull of some your descriptions for reuse within the implementation section, since it is so detailed. I know the implementation section talks about not referencing things that are already in the class diagram dictionary so you exclude things like the enums. You might want to ask the instructor/TAs about that as well, because you might only need to write a few lines for it.
>
> The only other suggestion that I have is to make sure that you correct the descriptions that appear to be filler, for Customer and Basket. I appreciate the level of detail you went into when giving suggestions for mine and feel a bit bad that I don't have much to say about yours, but it's clear you put a lot of time into the document.

Steven Hines' peer design review comments:

> I like your use case diagram. I'm unsure if the store controller is automatically supposed to control the appliances as well (I think it is), which is good that you recognized that. I initially separated out the two in my design doc because I thought the store controller was separate.
>
> I'm not sure if I like your appliance / sensor relationship in your class diagram. I thought about this for a while in terms of inheritance vs composition. It doesn't make sense to me that an appliance is a type of sensor. This results in having a sensorenum field for an appliance that will most likely go unused. If I were to go this way, I'd probably have appliances have a sensor field instead. (I went another way that you saw in my implementation but I'm not sure that's correct either). I think the important thing is that we design this in a way that its easy to change in the future (once we know how the controller will work). It seems largely that you've done that, so maybe its not worth debating / thinking about.
>
> I think explicitly the customerList and masterCustomerList should always refer to the same list. All registered users in one automatically register in the other. At the very least, I'd make sure to clarify that these pointers refer to the same exact objects in memory.

I explicitly labeled the StoreModelService as a singleton as well (based on piazza). But I'm not sure that either of our structures really qualifies as that. A singleton is supposed to be a factory that returns the same instance every time of the service. What we have here instead, is a service class that implements an interface with no real restriction on the amount of instances we create. I think (if I understand it correctly) we should be having a singleton factory class that creates a storemodelservice class instance and then always returns that instance. I'm going to raise a piazza question about this once I finish catching up with yesterdays section to make sure that this isn't addressed / clarified. (And yes I know the instructor specifically indicated to decorate the storemodelservice class as a singleton, which is why I'm confused – I'm mainly bringing this up so you can think about it and look for the feedback on piazza later).

I'd also want to bring up that you should have an auth_token parameter in all your API calls in the interface class.

I really like how you included associations that weren't immediately obvious to me (Customer → aisle), I'm going to be playing around with mine to make sure I include things that are technically associations if not specified.

As far as your class dictionary goes, I think you did a good job. But I do want to clarify I think according to the one piazza answer, that you should be including the storemodelserviceinterface class as well as the store model service. Essentially the interface class should have descriptions of the methods. The implementation should not have the methods because they are inherited (I think this is important as well because it makes sure that the method descriptions and requirements from the design document are an API not a class specific implementation of them).

I think your testing section needs a bit of work. You didn't really address performance or regression testing.

Your risks section was also left blank (I assume you were pressed for time like us all), so I just want to clarify that persistence of data, security issues related to user authentication, etc. should be addressed here.

Overall: I wanted to say you did a really good job here. I feel a lot more comfortable with my design as well since we ended up with something vaguely similar.

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

For my peer review of Steven Hines' design, I provided comments via in-line comments in their Word document.

My comments on Steven Hines's design:

Overall I think your diagram looks really good. A few comments though.

First, I was a bit unclear myself, especially with discussion for assignment 1 revolving around getters/setters not needed in class diagrams. For my design I went more inclusive, to play it safe. You may be thinking of these, but just to call out methods I didn't see on the diagram:

getAisle
getShelf
getInventory
updateCustomer
get/createBasket
add/remove/clearBasket
getSensor
getAppliance

The arguments to those methods closely resembled how I approached them (i.e. creating an object first, and passing that in to createX() – similar to processTransaction() from asn1). I think this makes sense, but I also see the logic in the approach Stephen took with his design (i.e. passing all the arguments through createX(arg1, arg2, arg3) to instantiate the object within the ModelService. Just wanted to put that out there as an alternate idea. Also, good job including the auth_token as a parameter. I forgot to do that myself and given Piazza post @129 (https://piazza.com/class/k03v504888cgs?cid=129) it seems like your approach is the correct method.

In terms of associations, you included the customerList in the StoreModelService, which I thought was quite nifty. There were several I didn't see though, including an association between Inventory and Product (in place of the 'productId' property you included, and between Customer and Aisle (in place of the 'location' String you included). I included both of these as composite aggregations (black diamond arrow), but Eric's Piazza post this morning seems to indicate the Inventory class at least should be an association classe, i.e. double arrows (post @136 — https://piazza.com/class/k03v504888cgs?cid=136).

Obviously, designs can differ. I may have gone a bit overboard with associations in my design so take my suggestion with a grain of salt.

I noticed you separated Size out into its own class. I remember there being Piazza discussion about the 'size' issue in terms of shelf storage — which may be your thinking about this. The requirements state size is weight and/or

volume, which sounds wishy-washy, so a Size class might indeed be the way to go.

Final comment is on appliances/sensors. I was torn myself on how to go about this. I ended up with a parent/child relationship to reduce duplicate code between the two — e.g. same respondToEvent() methods. My approach is probably lacking some specificity — I like your keeping two separate lists, one for Appliances and another for Sensors to easily differentiate between the two. I was confused myself in terms of how these devices will function with the further detail coming in assignment 3 so I know this was the most "eh, let's see" part of my design until I started coding.

— —

I would call out the 'store.script' file we're required to submit and what the bash script will do. Will it be interactive and prompt the user for names of test scripts? Will the input be a directory and be the bash script's job to loop through all files located within, passing through as arguments to the TestDriver?

I'm not sure if the additional detail is necessary, but I think it wouldn't hurt to include for the graders reading it.

For my peer review of Stephen Thompson's design, I provided comments via in-line comments in their Google document as well as more general/summary comments via email. The comments I provided via email were:

Hi Stephen,

I made some comments and suggestion edits in the doc itself. Overall, I think the design looks solid. I really liked some of the other sections — Overview, Exception Handling, etc. — in additional to the structure of the class diagram. A few points here and there, which as I noted throughout, I'm still thinking through ideas as well so mainly my suggestions are things to think about and framing things differently.

The one area I didn't comment on in the doc itself was the class diagram; there wasn't a good place I saw to do so — so I'll do that here.

First, some general comments. I'm no Astah or UML expert, but I would look closer at some of the arrows and multiplicity numbers you included. I mainly copied examples (which albeit could be incorrect examples) but for a specific example, the StoreModelService to <<interface>> arrow I have as a dashed line (see slide 27 in the week 2 lecture handout). Re: multiplicity, I forget what the assumed default is, but I would look at all the "1"s you have at the shaded-diamond part of the arrow. The one doesn't hurt, but I think it might be

extraneous? On the arrow-head portions, I would also look at the number you selected. One example: Store -> Aisle (sensor, appliance) has a “1..*” listed. For the application to be useful, there does need to be at least one Aisle (sensor, appliance), but on initial creation, there could be zero. (e.g. I just bought the building, but haven’t bought any of the equipment to put inside yet).

This might be a TBD part of the diagram, but a lot of the StoreModelService methods lack arguments to them (with the exception of AddAisle()). If it was an oversight, what was your idea on how to handle these methods? I also didn’t notice where the following methods (which appear in the command syntax) might exist:

- Add/show product
- Create event/command (your Update**State() methods?)
- Add/remove/clear basket

A final note, about associations, your method of trying sensors/appliances to both the Store and an Aisle is interesting. My design might need to elaborate on this a bit (currently I just have the store keep track of all ‘devices’). One bit I would be concerned about is update all of the associations anytime one changes. For example, when a robot assistance moves from one aisle to another, both the store and the aisle need to update their references. I would also think of the arrows going the opposite way, where an Appliance or Sensor can belong to an Aisle rather than the Aisle containing 0..* devices. This could possibly vary depending on the sensor/appliance where a microphone would stay in the same aisle, never moving, compared to a robot assistant which could roam around as it performs tasks.

I hope these comments are helpful. If anything here or in the Google Doc needs clarification, please let me know and I’ll try to explain further.