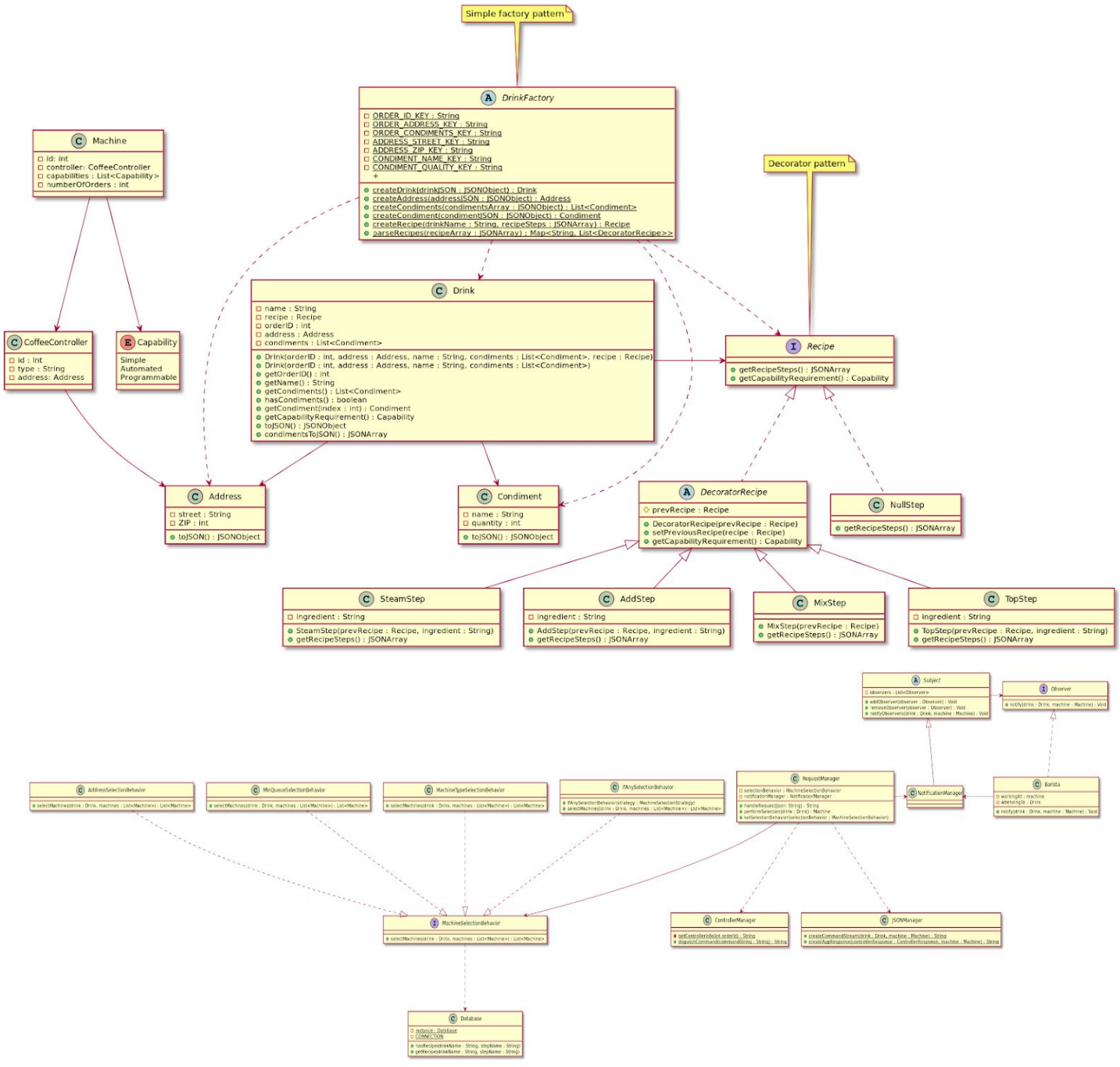
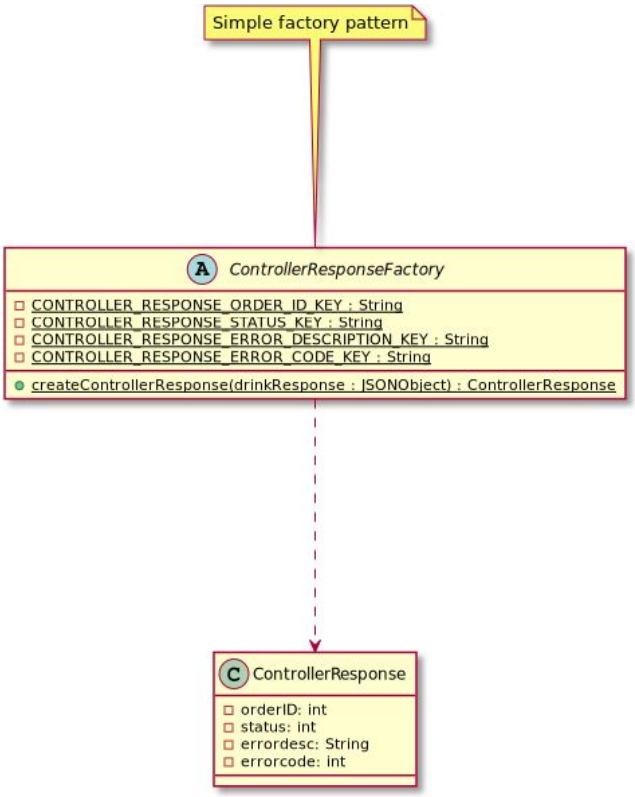
Class Diagram

(A full sized version of the entire class diagram can be found within the code zip file)

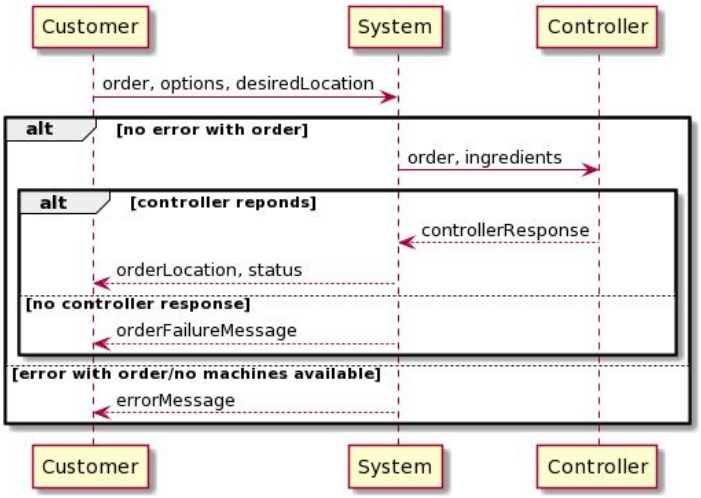



Simple factory pattern
ControllerResponseFactory
CONTROLLER_RESPONSE_ORDER_ID_KEY : String
CONTROLLER_RESPONSE_STATUS_KEY : String
CONTROLLER_RESPONSE_ERROR_DESCRIPTION_KEY : String
CONTROLLER_RESPONSE_ERROR_CODE_KEY : String
createControllerResponse(drinkResponse : JSONObject) : ControllerResponse
ControllerResponse
orderID: int
status: int
errordesc: String
errorcode: int

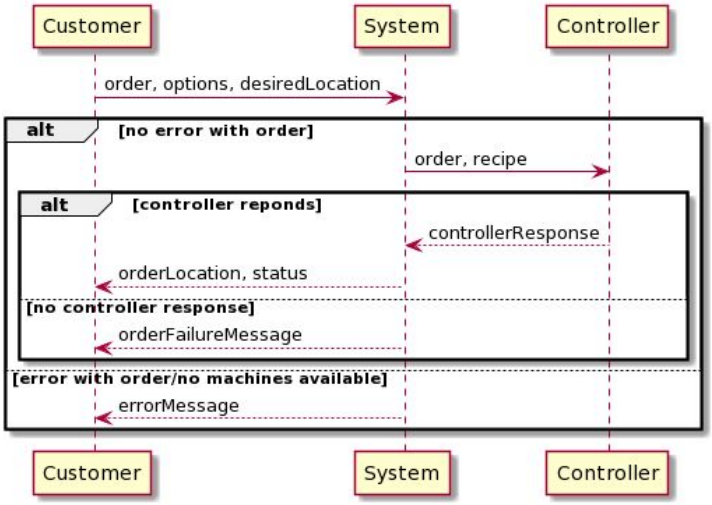
Sequence Diagrams

The sequence of steps for UC-3 and UC-4 are virtually identical; thus, the sequence diagrams for both use cases are virtually identical.

UC-3: *Order a coffee with ingredients that can be specified for a programmable coffee machine*



UC-4: *Order a coffee with specific recipes that can be sent to the programmable coffee machine, where the recipes include ingredients as well as preparation instructions.*



Design Rationale

We used the Decorator pattern to implement Recipes. Since a Recipe is a list of sequential steps that all have an action and (almost all have) an ingredient, we decided the steps could compose themselves to create a Recipe. Then, to get all of the steps needed for Recipe, the outermost step simply calls a method which returns a JSONObject of all the necessary recipes that have been decorated on top of each other. Drinks keep a pointer to their recipe as an instance variable.

Factory Pattern was used in generating drinks and controller responses using the DrinkFactory and ControllerResponseFactory respectively. Use of Factory pattern in these areas decluttered the creation code for controller responses and drinks; said creation code for these classes included JSON parsing that we ultimately deemed to be irrelevant to the rest of the classes' functions.Removing the creation codes from drinks and controller responses allowed those classes to function as the real world entities they were made to represent as.

Assumptions & Trade-offs

- **Trade-Off:** *Use of Factory Method over Abstract Factory*
    - Since our implementation of factories only moved the creation of one class to a factory, we figured Abstract Factory seemed to be overkill. If in the future, there were different types of ControllerResponses, drinks, etc, we could easily expand the factory methods to support creation of these objects as well. One concern that could arise is the difficulty of franchising Drinks, i.e. making drinks specific to a particular store since the factory method is not necessarily built for this functionality; however, we made the assumption that each store has their own database with their own recipes, drinks, etc. Therefore, franchising did not need to be handled in our implementation.
- **Assumption:** *Each store using the service has their own database*
    - See Trade-off above regarding Factory Method
- **Trade-Off:** *Removal of Order class*
    - Initially, we used an order class which stored information about: the drink requested, condiments for the drink, address for the order, and an order ID. We decided to remove this class to improve cohesion and intuitiveness in our design. In particular, we found that orders having a list of condiments fails to make much sense as condiments are specific to the drink. While our classes are now more cohesive, we do think this complicates receiving and unpacking orders for a

particular barista/cafe as instead of receiving an order of sorts which can then be finished by the barista, baristas will now receive notification of a drink being made. At that point, they will have to know whether the drink needs condiments added by the barista or whether the machine can do this for them, whereas in the Order implementation Baristas could tell their assistance was needed if the order contained any condiments. That said, we figured modifications could be made within the construction of the response to the barista to solve this issue.