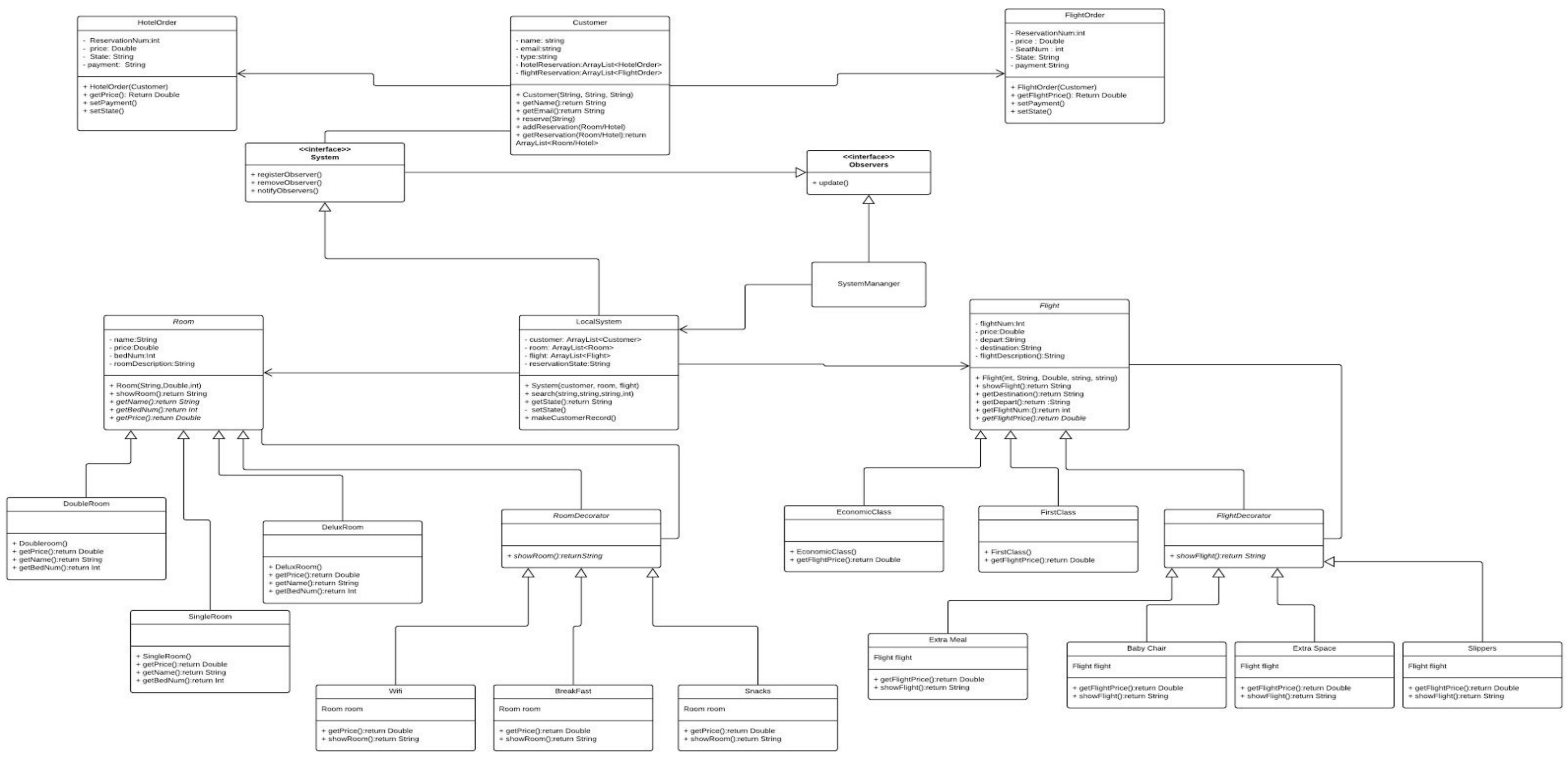
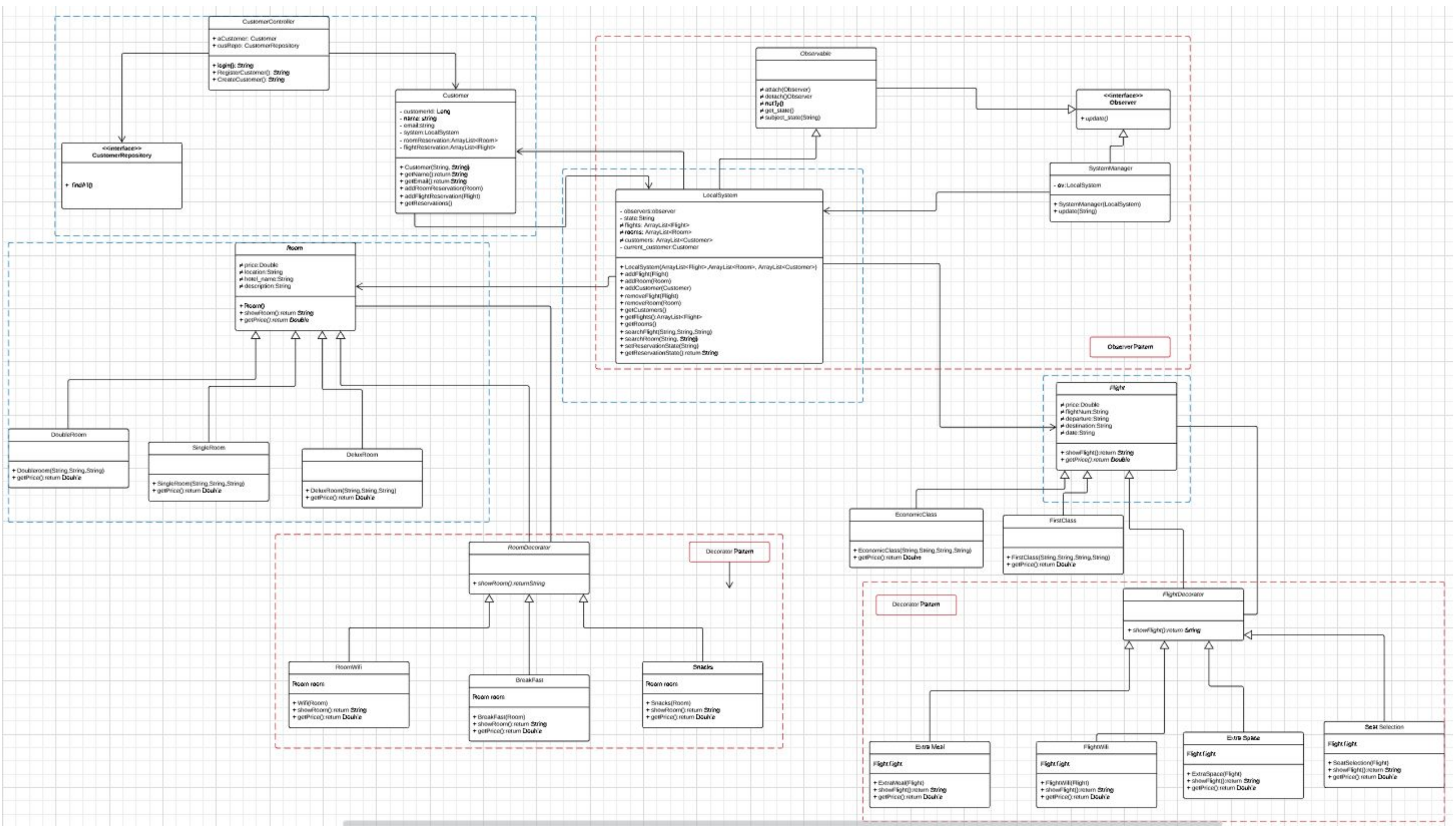
Final Project Report

- Team Members
    - Chakrya Ros
    - Xinyu Cao
    - Warren Payne
    - Christian Simons
- Final State of System
    - There are two systems. We started the project with a Spring framework, and after a few weeks of work the team ran into a wall when trying to map the Java classes (which are now completed) into the Spring framework. Christian and Warren stayed the course with Java during the last week to continue to integrate the system. Xinyu and Chakrya went about designing a backup in Django (a stack that our team has more fluency with) with the available time. The Spring project currently has completed classes (which have been tested apart from Spring successfully), a working Hibernate database, and a front end which mapped (through Thymeleaf) to the Java backend. The Django system is currently has fully functional in hotel search, user login/register, hotel booking, view user's reservation, and administration modification in database. It has completed classes and design patterns (also can be tested apart from the django framework), a Django database that contains a table with Room information, and a front-end that is connected with the database to show all the availability in hotels and all other functional features .
- Final Class Diagram and Comparison Statement
    - Original (from project 4):


HotelOrder
- ReservationNum:int
- price: Double
- State: String
- payment: String
+ HotelOrder(Customer)
+ getPrice(): Return Double
+ setPayment()
+ setState()
Customer
- name: string
- email:string
- type:string
- hotelReservation:ArrayList<HotelOrder>
- flightReservation:ArrayList<FlightOrder>
+ Customer(String, String, String)
+ getName():return String
+ getEmail():return String
+ reserve(String)
+ addReservation(Room/Hotel)
+ getReservation(Room/Hotel):return ArrayList<Room/Hotel>
FlightOrder
- ReservationNum:int
- price : Double
- SeatNum : int
- State: String
- payment:String
+ FlightOrder(Customer)
+ getFlightPrice(): Return Double
+ setPayment()
+ setState()
<<interface>> System
+ registerObserver()
+ removeObserver()
+ notifyObservers()
<<interface>> Observers
+ update()
SystemMananger
Room
- name:String
- price:Double
- bedNum:Int
- roomDescription:String
+ Room(String,Double,int)
+ showRoom():return String
+ getName():return String
+ getBedNum():return Int
+ getPrice():return Double
LocalSystem
- customer: ArrayList<Customer>
- room: ArrayList<Room>
- flight: ArrayList<Flight>
- reservationState:String
+ System(customer, room, flight)
+ search(string,string,string,int)
+ getState():return String
- setState()
+ makeCustomerRecord()
Flight
- flightNum:Int
- price:Double
- depart:String
- destination:String
- flightDescription():String
+ Flight(int, String, Double, string, string)
+ showFlight():return String
+ getDestination():return String
+ getDepart():return :String
+ getFlightNum:():return int
+ getFlightPrice():return Double
DoubleRoom
+ Doubleroom()
+ getPrice():return Double
+ getName():return String
+ getBedNum():return Int
DeluxRoom
+ DeluxRoom()
+ getPrice():return Double
+ getName():return String
+ getBedNum():return Int
SingleRoom
+ SingleRoom()
+ getPrice():return Double
+ getName():return String
+ getBedNum():return Int
RoomDecorator
+ showRoom():returnString
Wifi
Room room
+ getPrice():return Double
+ showRoom():return String
BreakFast
Room room
+ getPrice():return Double
+ showRoom():return String
Snacks
Room room
+ getPrice():return Double
+ showRoom():return String
EconomicClass
+ EconomicClass()
+ getFlightPrice():return Double
FirstClass
+ FirstClass()
+ getFlightPrice():return Double
FlightDecorator
+ showFlight():return String
Extra Meal
Flight flight
+ getFlightPrice():return Double
+ showFlight():return String
Baby Chair
Flight flight
+ getFlightPrice():return Double
+ showFlight():return String
Extra Space
Flight flight
+ getFlightPrice():return Double
+ showFlight():return String
Slippers
Flight flight
+ getFlightPrice():return Double
+ showFlight():return String

- New class diagram (project 6): **ALL CHANGES INDICATED WITH BLUE OUTLINE, PATTERNS INDICATED WITH RED OUTLINE**



- What changed between projects 4-5 and 6. In Project 6:
    - The classes CustomerRepository and CustomerController were added to allow for use of H2 database use and JPA
    - Customer was updated to include two arraylists of Flights and Rooms
    - FlightOrder was omitted
    - LocalSystem was updated to support the Observer pattern and various functions were added including addFlight, removeFlight, getCustomers(), getRooms(), setReservationState(), getReservationState(), etc.
    - In Room, several unnecessary methods were omitted, retaining only the methods showRoom() and getPrice() that are to be implemented it’s subclasses. The subclasses DoubleRoom, SingleRoom, and DeluxRoom have only their constructors and the implemented getPrice()
    - In Flight, several excessive methods were omitted, including getDestination(), getDepart(), getFlightNumber()

- Third-Party code vs. Original code
    - For the Spring framework, the code provided includes:
        - mvnw
        - mvnw.cmd
        - All directories in the submission with the exceptions that will be listed below
        - All underlying infrastructure that is not scene within the submission (the stuff that makes Spring work)
        - All files in the test directories and subdirectories (unused)
    - For the Django framework, the code provided includes:
        - Register and login form, model, views
    - Original Code:
        - All files within the controllers (2), dao (1), components (2), decorators (14), entities (1) directories; as well as LocalSystem.java, Main.java, and SystemManager.java.
        - All front end files, including all files within the templates (5), all files in css (5), all files in the images, and all files in static (6). These are the constituents of the front end.
        - All subdirectories of the “com” directory
        - Setting.py for Django framework
- Overall OOAD process of the project development cycle
    - As mentioned above, the team worked solely on the Spring implementation until the week prior to submission. We started our work by deciding on a project; eventually landing on our flight/room booking web application. Afterwards, we spent a few days looking into JHipster and Django. Django was the more obvious choice, since all of the team is relatively fluent in python with one team member having experience with python web development; but after consulting Professor Montgomery and successfully getting a local instance of JHipster running, we decided to pursue a Java application. The core reason for this is that we believed it to be the greater learning experience and more suited for heavily object oriented work. After a bit more research, we dropped from a JHipster stack to a more modest Spring stack (a subset of the JHipster stack) for simplicity sake.
    - From here, our analysis phase began as we started to look into the high-level requirements for the project and listing out some use cases. In identifying the requirements for the project, we first started by simply defining what our application is meant to do/solve (it’s meant to aid users in finding flights as well as rooms, much like Priceline); we then leveraged these very high-level requirements to hone in on specific requirements (this was where we defined our use cases). From here, we identified what our likely important objects would be, described them, and removed any redundancies. The root classes included our LocalSystem, Customer, Rooms, and Flights; where the LocalSystem functioned as the center-most class to control the Customer, Flights, and Rooms entities., Finally, we diagramed the objects’ relationships to one another (class diagram

and sequence diagram), and in doing so we completed the **conceptual model** for the project.

- After completing the analysis phase of the project, we moved into the actual **design** of the project. This is where we took our conceptual model and mapped it to an actual viable system design. To start the design process we expanded on our class diagram to include many more subclasses (i.e. the class diagram that we submitted). This is where were able to incorporate **design patterns** to aid in the system's functionality. Since our stack choice was Spring, we were inherently going to use the **MVC pattern** for dealing with data access and views. We also elected to use the **decorator pattern** to add behaviors to the **Flight** and **Rooms** objects (these can be found in the components and decorator directories). Finally we decided that perhaps **Observer pattern** could be used in letting **Customer** be notified in the case that a room or flight was booked (to avoid double booking rooms and flights). This concluded our design portion of the planning process.
- From here we moved on to implementation. We worked on the Spring code for a few weeks but to little avail, and during the last week, we divided the team so that two could go work on a backup Django system while the other two do what they can to integrate Java with the Spring framework.
- All in all, the analysis and design phases went pretty good, but two mistakes were made. We decided on our technology (Spring) before we settled Conceptual Model and System Design. Ideally, the technology should be chosen to fit the needs of the system rather than contouring the system to fit into the technology. The other mistake was not divvying up the workload efficiently, leading to overlap work and incohesive code.
- With regards to the project as a whole, we made some of the "classic software mistakes" as described by Steve McConnell.
    - **We abandoned the original plan under pressure**. We cut our workforce in half by breaking off to design a backup Django application.
    - We had a **lack of feature-creep control**. We underestimated the learning curve of the Spring framework. This lead to an **overly aggressive schedule** that wound up preventing us from completing a fully functional webapp.
- To list some design process elements we incurred:
    - We developed a **conceptual model**
    - We leveraged the conceptual model to a fluid **system design**
    - We ran into some issues with McConnell's classic software mistakes. Including **lack of feature-creep control**, an **overly aggressive** schedule, and **abandoning the development plan under pressure**.

Final Repo: https://github.com/chakryaros/Django-webapp

Reference:

- https://medium.com/@himanshuxd/how-to-create-registration-login-webapp-with-django-2-0-fd33dc7a6c67
- https://docs.djangoproject.com/en/3.0/intro/
- https://howtodoinjava.com/spring-mvc/spring-mvc-display-validate-and-submit-form-example/
- https://docs.djangoproject.com/en/2.2/intro/tutorial01/
- https://simpleisbetterthancomplex.com/series/2017/09/04/a-complete-beginners-guide-to-django-part-1.html