DESIGN DOCUMENT

Airline Ticket reservation system

SOLID

**How does your program follow the SOLID principles?**

The **meal** class of our program follows the **single responsibility principle** of SOLID. This is evident because it is isolated from other irrelevant classes and most importantly, has only one reason to change. For example, if we were to ever need to add a parameter to our meals such as the sodium content of meals, we would not need to make a change anywhere else in the code except our meal class due to its adherence to SOLID design principles.

The **user abstract class** and its **subclasses, Passenger and Agent**  are a good example of a class following **single responsibility and Open-closed principles.** This is because it abides by the rule that its entities are open for extension but also closed for modification. We are able to add many more types of users without worry of any more modifications to code than just the additional user type. Once the subclass is added, no higher level of code needs to be touched for functionality to remain, making development and extension much easier on us. Looking at the **Single Responsibility Principle**, this abstract class would only have one single responsibility of storing user information no matter the type. The adherence to this principle makes the class incredibly easy to maintain.

The **baggage abstract class** follows the **single responsibility** since the baggage class has the sole objective of creating a bag with the appropriate attributes and does not employ any other methods to fulfill unrelated tasks. The baggage class along with its subclasses **CheckinBaggage and CabinBaggage** follow the **Liskov substitution principle.** We can see this because the subclasses only extend the usage of the abstract baggage class by filtering 2 different types of bags and calculating their independent prices. The subclasses dont override nor modify any original methods, which also satisfies the **Open-closed principles.**

Since the **Seat superclass** and its **subclasses (EconomySeat, BusinessClassSeat, FirstClassSeat)** are formatted based upon the template pattern, this directly adheres to the **Open/Closed principle** since we can add different types of seats without making any significant changes to the structure of this part of the code. For example, if we wanted to have a private jet seat, we would simply need to create a PrivateJetSeat subclass of the Seat superclass, and we are done. Additionally, there wouldn’t be any need to make changes to the SeatManager either since we are merely extending our program to add a new type of Seat. This also adheres to the **Single Responsibility Principle** since each subclass is responsible for one task, a specific type of seat. We are also following the **Dependency Inversion Principle** here since our SeatManager, a use case class, does not depend on these subclasses directly, it merely depends on an abstraction, the Seat abstract class, hence inverting the source code dependency.

Since the different **membership subclasses (Standard, Silver, Gold, Platinum)** implement the **MembershipStatus interface**, we adhere to the **Dependency Inversion Principle**. This is true since our MembershipFactory class depends on the interface MembershipStatus instead of directly depending on the membership subclasses, hence inverting the source code dependency.

The **MembershipStatus interface** and the classes that implement this interface follow the **Interface Segregation Principle** since all of the functions from MembershipStatus are implemented by all of these classes, without having any unnecessary functions. Hence by following the Interface Segregation principle here, we have also avoided the code smell of **refused bequest**, which refers to having unused methods.

We must also note that the **MembershipStatus interface** and its **subclasses** adhere to the **Open/Closed principle**. This is because we can add different types of member subclasses without making any major changes to our existing code. Neither are we required to make any changes to our existing interface. The new classes can simply implement the interface and extend the functionality of our existing features. Our above implementation also follows the **Single Responsibility Principle** since each membership subclass is only responsible for one specific member (Standard, Silver, Gold or Platinum).

**Some improvements which we can make in our design to follow SOLID principles**

Although our program follows all of the SOLID principles in one way or another, there are instances where we believe some principles are not applied significantly. For example, since our program only has one interface, MembershipStatus, we feel that our program lacks having interfaces, implying that we also lack the application of the interface segregation principle. If our program contained more interfaces, we feel that it would better show how our program follows the interface segregation principle in multiple sectors. We have not found many places to incorporate interfaces in our program so we are hoping to get some feedback from the TA on where they might fit in.

Another place that requires refactoring is MembershipFactory. In terms of our Seat, we decided to create a SeatFactory that has the responsibility of getting specific types of Seat Objects and a SeatManager which contains certain manipulation functions for Seats. In MembershipFactory, we have a get Membership function and manipulation functions, hence we feel that it would be best to create a separate MembershipManager class that deals with these manipulation functions. It would go in line with the single responsibility to split this class into two, as well as make MembershipFactory and the new MembershipManager use case classes to follow clean architecture.

Another place where we acknowledge we could have made a better design choice would be in the RefundAndRescheduleSeatPriceCalculator class. Although this class, by definition, is tasked to calculate the refund price for a ticket and the cost of changing a departure date, it can be argued that this violates the single responsibility principle. In retrospect, we could have broken this class into 2 different classes each responsible for calculating the respective numbers.

CLEAN ARCHITECTURE

**CRC MODEL**

We have create our CRC model on a site called Whimsical, which helps make flow charts. [Here](https://whimsical.com/the-beginning-Ba4cHyRqbFPhUDJqxvaGpR) is the link to our model. You don’t need to be signed in to view the model.

Alternatively, you can click on this link:

<https://whimsical.com/crc-phase-1-3DhAuRbQ2arnYXawAfBN9j>

**SCENARIO WALKTHROUGH**

* When the program is run, the CmdUI class asks the user (in the command line) if they are a User or an Agent.

* If the user selects that it is a USER, the CmdUI class would ask if the user wants to sign-up or sign-in. For the sake of our scenario-walkthrough, we select the sign-up option.

* The CmdUI class then prompts the user to input their name, email and phone number. The system asks the user if it is satisfied with its answers- to which the user replies – yes to continue. This triggers the CmdUI class to use its passangerSessionHandler.BookingSystem.passangeManager class to create the user object  and assigns a unique id to the user with which the user can sign-in with later.

* When the user signs-in with this unique id using CmdUI, a menu form opens up where the user has the option to book new flights or view their old flights, and edit them. For the sake of this walkthrough the user  selects the option to book a flight. The user enters the arrival/departure destination and travel date.

* The CmdUI then uses its passangerSessionHandler.BookingSystem.AirlineManager to filter out flight according to the given specifications.

* The CmdUI class then displays the list of filtered flights to the user in the command line. The user selects the desired flight. The system then prompts the user to enter which seat standard they want – Economy/Business/First.

* Then the system displays a flight seating chart using the seatPresenter class in the command line for the user to choose their seat. After the user has made their seat selection, the system asks the user if it wants to provide any baggage information, or proceed directly to meal preferences.

* For this walkthrough, The user chooses to proceed directly to meal selection at this stage. Where the class passangerSessionHandler.BookingSystem.MealManager is used to get all the meals and is presented to the user.

* After the meal is selected, the CmdUI uses passangerSessionHandler.BookingSystem.TransactionManager to generate a transaction and then the TransactionPresenter is used to present it to the command line.
* At this point the CmdUI class asks if the user wants to proceed with the transaction and book the ticket.

* As the user clicks yes, CmdUI uses passangerSessionHandler.BookingSystem.TicketManager.addTicket function to book the ticket and add it to the app.

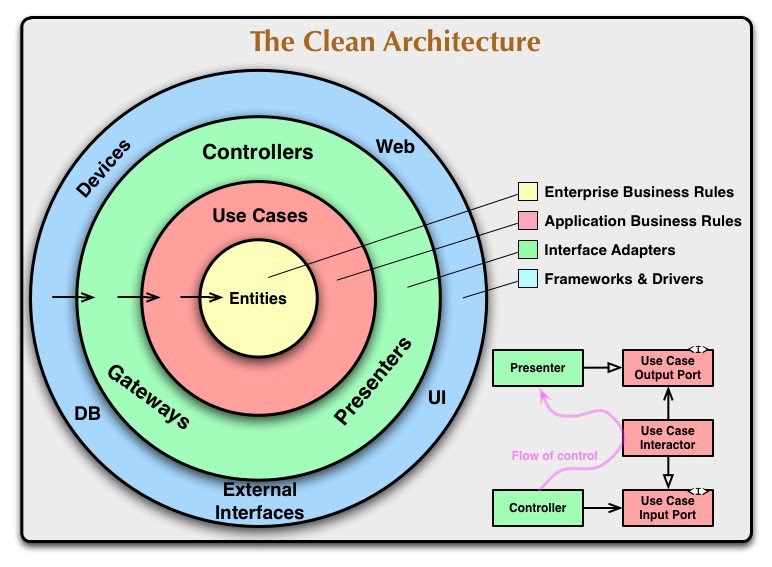
* Then using the TicketPresenter the ticket is shown onto the commandline, and the user is taken back to the menu.

* The user now chooses to end app use and exits the program.

**Are there any clear violations if we were to randomly look at the imports in a few of your files?**

No. We don’t think there are any such violations in our program

**Is the Dependency Rule consistently followed when interacting with details in the outer layer?**



The dependency rule is clearly followed throughout our entire project and this can be seen via a walkthrough of how we load data from our database. It could be helpful to look at the CRC model along with this walkthrough.

As soon as the app is launched, An instance of PassengerSessionHandler is made in the CmdUI (User interface class). This PassengerSessionHandler then in turn initializes a PassengerDataHandler, TicketDataHandler and BookingSystem class.

 The PassengerDataHandler, and TicketDataHandlerser  take in the BookingSystem and update it with the data in the database.

Note that here CmdUI, which is in the outermost layer, is dependent on PassengerSessionHandler, which is in the interface adapter layer, and the PassengerSessionHandler is dependent on only classes of its own layers which are PassengerDataHandler, TicketDataHandler and BookingSystem.

Let us walk through how the users’ data is loaded into the app. The PassengerDataHandler class loads the data from the server using a function called fetchPassangersIntoApp that uses DataConnector to get all the users’ data.

It then uses the BookingSystem.passengerManager.addPasenger function to add the passengers to the app.

Note that here the PassengerDataHandler, a class in the interface adapter,  is dependent on BookingSystem which is a class also in interface adapter, and on ticketManager which is a class in Use Cases.

Now let's look at how the PassengerManager class is adding the passenger. The  PassengerManager class has a function called addPasenger that creates a Passenger object and adds it to the list of passengers in the app.

Note that here PassengerManager, which is a UseCase class, is dependent on only the Passenger class which is an entity class.

**DESIGN PATTERNS**

**Places where our group has used design patterns!!**

We have implemented two design patterns upon the Seat superclass and its subclasses. Firstly, since Seat is an abstract parent class, and it has different types of seats as its subclasses, this is a clear implementation of the **Template Design Pattern**.In general, the template pattern helps the Seat abstract class be seen as a way to provide us with different implementations of its methods within its subclasses.

Additionally, we have also applied the **Simple Factory Design Pattern** by creating a SeatFactory class that allows us to create different types of Seat objects (Economy, BusinessClass, FirstClass) without showing how they are formed to the client. These two implementations can be seen in the pull request called “Main dinkar”, as well the implementation of the MembershipFactory class.

Another similar use of the **Template method design pattern**, in our project, is when we create an abstract Baggage class which contains information about baggages. We then create two subclasses called Cabin Baggage and CheckedIn Baggage which inherit the Baggage abstract class. The sub-classes can then override the abstract methods to account for the variation in baggage type while re-using most of the similar code.

We have also implemented the **Observer Design Pattern** in the TicketManager and PassengerManager Class. We have made the TicketManager class the Observable Class and the PassengerManager the Observer Class. By implementing this pattern, the PassengerManager class gets notified whenever a passenger has booked a ticket. This is helpful as it notifies the PassangerManager to take the necessary action, such as updating the points of the passenger who has booked the ticket and their membership status. This implementation of the Observer Design Pattern can be found in the pull request called “Phase1-Avnish”.

Also a part of the implementation for the Simple Factory Design Pattern for the MembershipFactory class can also be found in “Phase1-Avnish” as well as the “Main dinkar” pull requests.

We have also implemented the **Iterator Design Pattern** in the TicketManager, TransactionManager, PassangerManager, AirlineManager, Airline, Flight and MealManager Classes. Since we were implementing the Iterator Design Pattern in so many classes, we created a generic GeneralIterator.java class which we called in the above mentioned classes to implement our iterator design pattern. By implementing this design pattern, we are not having to expose the underlying representation of how the elements are stored in a particular class. This can be found in the “Phase1 eeshan” pull request.

**Identify and describe any patterns that could be applied in future with more time?**

If we have more time to add more functionality to our program, we can consider implementing the **Decorator Design Pattern** upon the Membership portion of our program. We could add a new property of having Flight Attendants as an additional constraint upon being incorporated into the Membership programs. We would create a FlightAttendantDecorator abstract class that could “decorate” the Membership program by providing the additional feature of special Flight Attendant perks dynamically. This leaves room for further extension since we could form subclasses of FlightAttendantDecorator that can represent additional perks of a Flight Attendant alongside of what Membership status they have.

To facilitate booking of a ticket, we have partially implemented the **Memento Design Pattern.** This design pattern functionally allows us to carry out a rollback operation which restores the state of a class to a previous version. We have used this pattern to store mementos of the state of the available seats in a plane, and if a user decides to change/cancel seats, the map of the seats is restored to what it was before the booking had been made. With more time, we are looking to extend the functionality of this design pattern by allowing for this rollback operation to be carried out on the meals and baggage of the user too so that if a user decides to change meal preference mid booking, the can restore the state of the ticket to just before the meal was chosen, instead of starting the entire procedure again. This can be seen in the pull request memento design pattern in branch phase1dix

**USE OF GITHUB FEATURES**

Our group made extensive use of GitHub features by creating commits for each activity that was being completed along the way. This can be seen from the commit history in the main branch on GitHub. Also, all group members made extensive use of creating issues. The issues mentioned which design patterns could be implemented (while pointing to specific classes) to optimize our code. Issues were also posted regarding which Code Smells were found and needed to be corrected. Also, issues were created to refactor poor code and improve it. Regular pull requests were made to the current branch for all group members to get an updated version of the code and resolve any merge conflicts which may have arisen.

**Code Style and Documentation**

We have ensured to maintain a good coding style and documentation. All our files have appropriate documentation explaining what the code is doing. All our coding styles use simple variable and function names which can be understood by the average user and programmer. The names are descriptive in nature, giving us an idea of the primary task of a variable or function. Furthermore, all IntelliJ warnings have been fixed. Also, all pull requests have been reviewed before merging to the main branch and all code conflicts have been resolved. At all stages of writing our code, we have ensured to follow good coding practices.

**TESTING**

**What did your group test?**

**TestAirlineManager:** This class is testing the **AirlineManager** in its functionality of adding and getting airlines from its storage space. The method testAddAirline tests if the airlines are being added to the arraylist properly by checking the size of the array before and after the insertion of airline and also asserting whether a given airline is in the correct index of the list which depends on when the airline was added. The method testGetAirline is asserting if the airline at index 0 is the same airline that is returned by the GetAirline function when the index 0 is passed.

**TestFlightFilter:** This tests the **FlightFilter** class for its functionality of returning the available flights of a particular airline. The available flights returned must match the arguments passed into its constructor and must be of the specified airline.

**TestRefundAndRescheduleSeatPriceCalculator:**  This tests the methods of **RefundAndRescheduleSeatPriceCalculator** class. It tests whether the calculateRefundByDaysLeft method and calculateDateChangeChargeByDateLeft return the correct numerical depending on the formula used to calculate these values.

**TestPassengerManager:** This class is testing whether the **Passenger manager** is properly adding to the list of passengers when a new one is created by asserting the size has increased by 1 and the proper ID number is assigned.

**TestTicketManager:** This class is testing the **Ticket manager** by creating the required objects for the manager to work, running the method and checking to see if an instance of ticket was created.

**TestMembershipFactory:** This file tests the functions contained in the MembershipFactory class. The testGetMembership function tests whether getMembership returns the correct types of objects of Membership (Standard, silver, gold and platinum). The testSetMembership function tests whether setMembership is able to switch the membership status of a passenger based on how many points they have and the given points threshold. The testCalculateLoungeAccess and the testCalculatePointsForLoungeHours function test whether the corresponding functions return the correct amount of hours and points .

**TestSeatFactory:** This file tests the functions contained in the SeatFactory class. The testGetSeat function tests whether getSeat returns the correct types of objects of Seat (EconomySeat, BusinessClassSeat, FirstClassSeat).

**TestSeatManager:** This file tests some of the functions contained in the SeatManager class. We made an EconomySeat object with a price of $200 as our basis for testing the functions, calculateRefundByDaysLeft and calculateDateChangeChargeByDateLeft in the SeatManager. The tests check whether the refund and the charge are changing correctly based on how many days are left until the departure date. We didn’t get time to test the other two functions in SeatManager but they can be tested as well.

**MealManager\_test:** This file tests if there are any meals in the system. This file also tests the functionality that meals are getting added correctly to the menu. This test file also checks if the system correctly returns a meal requested by the user, if that meal is present in the system.

**BaggageManager\_Test:** This file tests that after all the discounts and rebates relating to a users membership status, discounts and addition of any extra penalties, the cost remains a positive number so that the airline can turn a profit. If the cost is a negative number, it would mean that we must increase the price per baggage so that at the end of the day the airline does not make any loss.

**Components that were a bit difficult to test**

The **Reschedulemanager class** incorporates usage from another manager called **TicketManager**. This implementation would be more of a challenge to test since if a particular test results in a fail, it could mean either class failed and we would not know which part to fix exactly.

The setMembership function in membershipFactory is hard to test since it requires us to use a private instance variable, points, that is not part of the constructor of Passenger. We have to call functions like setPoints and getPoints of this private variable while instead it would be easier if we could just have a passenger object that includes points as a parameter. We are still able to test this function but it may be to configure for a better design and easier testing.

**REFACTORING**

**Code Smells and other minor refactorings!!**

Initially in phase 0, we had **redundant interfaces** being implemented by the parent class Seat and the interface MembershipStatus, but since they already contain these methods, where their subclasses implement these methods, there was no need to keep the redundant interfaces.​​ This refers to the code smell of **Dead Code**, and the quickest way for us to fix this issue was to apply the **Collapse Hierarchy** treatment by merging the redundant interfaces with either Seat or MembershipStatus, and deleting the redundant interfaces

To avoid having the **Duplicate Code**, code smell, we shifted certain functions from the Seat parent class and its subclasses, as well as the MembershipStatus interface and its subclasses, into “Manager” classes. This also allowed us to follow Clean architecture since these entities contained functions that should not be there, but after this refactoring, the use case Managers now contain these functions instead. The change to fix duplicate code can be found in the pull request called “Main dinkar”.

When looking at places to refactor our code, one code smell we came across was **Change Preventers** within theUser class. We noticed that everytime we tried to make changes in any of the subclasses, we had to change the superclass to accommodate for it. We noticed this flaw relatively early when adding the agent subclass. To fix this, we did the divergent change and split up the behaviours in the User abstract superclass. This is because we managed to divide functionality into subclasses so it not only fixed our initial problem, it also made our program better abide by SOLID design principles. These adjustments to fix the **change preventers** code smell can be found in a pull request called “Seat.”

When reviewing our code, we encountered the **Bloater** code smell in one of our methods in the CUI called findFlight. This method was initially programmed to execute 2 big tasks. First, it was prompting the user to enter their travel details so that the program can display all the flights available that match the users criteria. Second, the same method was responsible for asking the user to choose their flights. Since this method was turning out to be very long, we decided to split the method into 2 different methods which would each carry out their own separate tasks. We implemented the flightSearchFilter method that searches for the flight from the database depending on the user's input. If there is a flight in the database, this method calls on the whatToDoWithFlights method to allow the user to choose which flight to book. Hence, by breaking a big method into smaller methods, we were able to prevent the Bloater code smell.

When reviewing our code, we also came across the **Bloaters Code smell** for **Large Class**. This was present in the TicketManager class which was too large. We resolved this by placing the presenter/display methods in another class called TicketPresenter and the transactions related to the ticket in another class called TransactionManager. This can be seen in the “Phase1 avnish” pull request.

We also detected the code smell **Dispensables - Duplicate Code** in the AirlinesManager class. Over here the methods getAirline and setAirline were making use of a lot of similar code. This issue was resolved during refactoring for our Phase 1 code. This can be seen in the “Phase1 eeshan” pull request.

We also detected the same code smell **Dispensables - Duplicate Code** in PassangerManager class. We had similar code in getPassangerWithId and removePassangerWithId. This code smell got automatically resolved when we made changes to the functionality of our program which resulted in changes to the return types and internal code of these methods. This can be seen in the pull request “Phase1 avnish”.

We also detected the **Bloater Code Smell for Large Class** once again in the AirlineManager class. This class was too large. To resolve this issue, we extracted the presenter/display methods of this class to two separate classes called FlightPresenter and SeatMapPresenter. Also, we detected another bloater code smell for a **long parameter list** in the constructors of the Ticket Class and resolved that code smell. This can be seen in the pull request named “Phase1 eeshan”.

***Above are a few examples of refactoring which our team encountered. We encountered several other code smells as well which our team resolved along the way.***

***Our team also refactored the code by implementing several design patterns on the code from Phase0 as highlighted in the Design pattern section of our document.***

**Are there any obvious code smells still in your code that you missed fixing?**

So far as we have reviewed our code, at this point of time, we cannot seem to find any more code smells or refactoring errors.

Our team plans to go back to our code for Phase 2, given time, and once again conduct an extensive review to find any potential code smells which might not have been clearly visible or apparent during the two review sessions (during Phase 1) our team conducted to find code smells in our code.

**CODE ORGANIZATION**

**Is your code organized in a meaningful way? Is it easy to find things in your package structure?**

Yes! We believe that our code is organised in a meaningful way. We have used the **Package-by-Layer** packaging structure. This essentially means that we are packaging our code by each layer of Clean Architecture.

In this packaging structure, all our Entities would be in a single package/folder, all our Use Cases would be in another package/folder, all our Controllers/Presenters/GateWays would in in another package/folder and our Devices/User Interface would be in a separate package/folder.

This would make it a lot easier for someone to find the code to a particular function/class they need by just checking if the code they need to edit is in an entity, use case, presenter/gateway or in devices/user interface layer.

One look at our CRC model would explain how the Package-by-Layer packaging structure made the most sense. Our CRC model already shows our classes, interfaces and abstract classes divided up into layers of clean architecture. It would only make sense for us to leverage the work we already did (by dividing up our classes by layers of clean architecture) and implement it in our packaging structure.

**FUNCTIONALITY**

**Project Specification**

We seek to create a program on the Domain of an Airline Ticket Reservation system.

* Our program should have a UI which would be able to accept sign-in/sign-up data from the user. The system would be able to accommodate two different types of users – passengers and agents. We have not implemented the agent feature but we plan to implement it in future.

* The passengers would be able to search for flights based on their departure date and their arrival and departure destinations

* They would also be able to book a ticket, cancel a ticket, reschedule a ticket, enter their meal preferences and their baggage data (details of both checked-in and cabin luggage).

* The user would also be able to choose their seat numbers on the flight and the standard of the ticket – Economy/Business/First.

* The users would also have a membership – Standard/Sliver/Gold/Platinum member – to avail various discounts and privileges (like lounge access) on their ticket, which would get upgraded with their points/miles.

* If the user does not have any membership, the system automatically assigns them a Standard membership. The system automatically keeps adding miles (travel points) to the user’s account based on the number of tickets booked.

* The system would be able to calculate the final price of the ticket based on the user’s seat selection, meal preferences, baggage details and after applying any discounts the user may be eligible for based on their membership status.

* The system would also be able to combine together all the above-mentioned data to create a ticket for the user.

* In the future we plan to implement the agent feature, and The agent would also be able to do all the same tasks as the user. The only difference would be that the agent would have different sign-in credentials and would be managing the ticket on behalf of their customers. The agent would be able to add and remove their respective customers.

* The system would also be able to keep track of the details of all the flights for a particular airline, including each flight's departure date and arrival/departure destinations.
* The system should also have a data persistence component so as to store the state of different users and the state of different tickets created even after the program has been closed.

**DATA PERSISTANCE**

To Persist data in our app we are using SQL. We are using a postgresql database storage server to store data of our users and the tickets they have booked. We have created an Airline database on the server with two tables namely users and tickets. We then have a class called DataConnector that connects to the server and helps run queries. The DataHandler classes (PassengerDataHandler, and TicketDataHandler) then use this DataConnector class to load all the users and tickets before the app starts, and update the tables with the user actions, which include booking, rescheduling and refunding a ticket.

Thus, our program is able to store state and load state. Also, this state can persist across runs of our program.

**Reflection of the specification!**

We believe that the program does what the specification intends that it would do.

**Any changed functionality and is the functionality sufficiently ambitious given the size of your group?**

Initially we had planned to also implement the Agent interface for our program which would also allow agents to use our system. However, due to the recent decrease in our group size from 6 to 5 members and upon discussing with the TA we decided to leave the Agent implementation part of our program for the future. We did this to reduce part of the complexity given the amount of time we had to implement our project.

**Any major design decisions?**

All major design decisions were generally centred around which design patterns we had to implement and which area do we need to implement them in. This is described in the Design Pattern section of this document. All such decisions were taken collectively as a group with everyone contributing equally.

**INDIVIDUAL CONTRIBUTIONS & FINAL THOUGHTS**

**DINKAR**

Dinkar has helped with the application of design patterns within our program. More specifically he helped refactor our code to help us implement the template design pattern and the factory design pattern in certain areas of our code. These designs help us better structure our program to follow certain SOLID principles that, for example, can leave our program open for extension for phase 2, as well as help remove any unwanted dependencies. He also assisted in finding code smells within our code and helped to fix them. Some examples of such refactoring include removing code smells like duplicate code, dead code, etc. He helped write tests for our program and helped with styling and documenting our code via comments and javadoc. Throughout this process, he stuck to making sure his work followed the SOLID principles, avoided code smells, and stuck to our packaging strategy that directly correlates with packaging by layers of Clean Architecture. Finally, he helped to configure the CRC model to satisfy our changes to phase 0 and finish up the design document. I plan to contribute in making sure we make changes to our program based on the TA’s feedback as well as helping write more tests to cover more possibilities of specific methods and our program as a whole. I also plan to help find and implement ways to enhance our program’s capabilities for phase 2.

**AVNISH**

Avnish helped in coming up with design patterns which could be implemented in the project. He actively engaged in discussions to explain why the suggested design patterns would be a good fit for the project. He worked and implemented the Observer Design Pattern and also helped implement the Simple Factory Design Pattern. Throughout the coding process, Avnish followed good coding principles. He was able to detect and resolve several code smells such as duplicate code, large class and long parameter list amongst others. Avnish also was able to find places in the code to implement the SOLID design principles. He also detected and removed traces of the Agent Class (along with its function calls and usages) functionality from the program to reduce the project complexity. He wrote tests to test the functionality of BaggageManager and MealsManager amongst other tests. He helped extensively in adding documentation and comments in classes. He contributed significantly in group discussions and in making changes to the CRC model and making sure that the code followed the Clean Architecture Principles at all times. He helped decide on the overall code packaging structure and presented his views on why it was beneficial to use the package-by-layer packaging structure. He also contributed significantly in writing up and merging the entire design document into a report. In phase 2, I plan on refactoring the code better and finding places where potential design patterns could be implemented. I also plan on working on a feature which would allow users to use their points to make gift purchases of items such as smart-phones, laptops and headphones amongst others. I also plan on writing more tests to test the functionality of the program.

**KEVIN**

Kevin had helped with planning out where in the program we could implement certain  design patterns to fix occuring problems. He contributed to our group's design and implementation of the memento design pattern. This design pattern will be utilized when adding an undo function to our program to enhance the user's experience when booking tickets. There were also contributions made when identifying areas in the code with code smells such as change preventers when reaching the refactoring phase. Kevin also made sure to abide by SOLID design principles when writing his code and adjusting areas to  follow it better. Along with the rest of the team, he contributed in the final document write up and applying changes to the CRC model. In phase 2, I plan to continue to implement memento across our entire program so that any ticket booker can go back one step to undo instead of restarting every time a mistake is made. I also plan to keep on testing and helping out where necessary.

**DIXSHANT**

Dixshant has helped in brainstorming several areas in our code to implement various design patterns to make the code more efficient. He worked on the implementation of the memento design pattern which has currently only been implemented to store mementos of seat but is looking to extend the functionality in phase 2 to store mementos of various other attributes. He also refactored the code to prevent code smells like bloaters and break code down into smaller bits of code. Dixshant also helped with the documentation of the code, and caught and resolved bugs with the help of several tests in the process. Dixshant adhered to the SOLID principles while implementing his design patterns and did not violate clean architecture guidelines. As with the rest of the group, he helped finish up the write up of design document and made changes to the CRC model that reflects our code.

**EESHAN**

Eeshan contributed significantly in coming up with several design patterns which could be implemented in not only Phase1 but also later in Phase2. He implemented the Facade ,Iterator and Observer design patterns to improve the overall efficiency of the code. He was also helped revise and update the project specification and scenario walkthrough document. He shared his ideas on places where SOLID principles could be applied and Clean Architecture could be followed. He contributed significantly in identifying and solving code smells such as long parameter list, large class, duplicate and dead code amongst others. He helped in extensively documenting and commenting the entire project code to allow for user readability. He also added tests to extensively test the functionality of the PassangerHandler and TicketFilter classes. Throughout the coding process, he followed the SOLID and Clean Architecture principles. He was also responsible for merging all the code together. He also implemented the data persistence component of our program besides significantly and extensively contributing to the overall development of the User Interface. He also assisted in implementing the package-by-layer packaging structure to the entire code. Like all others, he made contributions to making changes to the old CRC model and writing up this project's Design Document.

**We are proud to say that each member of our group has contributed EQUALLY and extensively in making this Phase 1 project submission a success. Each group member actively engaged in discussions and contributed their perspectives while making design and functionality decisions. Each group member was actively involved in writing and fixing the code, as well as in creating the Design Document. We, as a group, strongly believe that Phase1 submission is our collective collaborative effort which has seen EQUAL participation and contribution by each member of the group.**

Open questions your group is struggling with?

Extensions: For our current implementation, our flight data is from a predefined set we created without API’s or real time data. In order to extend our program, we thought it would be a good idea to generate new flights but we would require more time to look into ways to create it and also to reliably test it. Doing that part could also further expand into getting and loading real world data of flights into our program which we believe would require elements like web scraping to do, which we would have to learn.

What has worked well so far?

So far splitting up our program into entities and use cases (by using the SOLID design principles) has worked out well so far. This has helped reduce inter-class dependencies and allowed everyone to work independently while implementing several of the required use-cases and entity classes. (This has increased the overall efficiency of our group to write code at a faster pace.) The low coupling of our design has enabled us to carry out several modifications to our code with little effort and almost no significant impact on other classes. Also, splitting our entities and use cases into different layers and classes has made it much easier for us to implement our Interface Adapters. Thus, making changes and adding new features (by different group members) is something which can be done well and seamlessly with our current design.

**THANK YOU!!**