Design Principle and Concept:

Single Responsibility Principle

We have almost 60 over classes for our project. The reason for that is because each class has very specific responsibility. Classes in boundary package deals with purely input and formatting of output, classes in controller package deals with the logic to govern the input and output of boundary classes, classes in dataController purely deals with the create/retrieve/update/delete of the flat file and misc package has other classes not in any of such classification such as the class to run the whole program.

Interface Segregation Principle:

In our classes, we do not implement interfaces that are not essential. For example, we only implement the Comparator interface to classes that needs their data to be sortable while other classes with other responsibility implements something else or nothing at all.

Don’t Repeat Yourself

We have lots of outputs that requires user to input a whole numerical number to select what they want to do next, we also have lots of outputs asking yes no questions. So we create a class specifically to validate required input like these instead of writing the same validations codes hundred over times. We also have a main dataController class where all other data controllers extend from so they can use its method to read and write text files to commit to persistent data. The data controllers themselves are created so we don’t repeat ourselves. For example, a movie show time has a movie tied into it, so instead of rewriting codes to retrieve movie details of that particular show time every time, we have a method in one of our data controller class to do it.

High level of decoupling with high level of cohesion:

We have 60 over classes for 10 or so required functions. If one of the method is not completed, it will not affect the progress of other functions. For example, we have a class (custSeatsControl) that deals with the display of a selection of seats and another class (custBuyTicketControl) that deals with the purchase of a seat. We can still test and progress with the logic (eg. Cannot leave a gap between seats) of how to purchase a seat without caring if how the display of the seat works. However, when the function for the display of seat is completed, the whole system can be fully integrated with just a few simple method call.

Encapsulation and information hiding

We have entity classes such as Cinema, Cineplex, Movies, Show time, movie schedules, seats, seat information, transactions and such. All the variables in these classes are private and can only be mutable with getters and setters or constructor so we fulfilled information hiding.

Inheritance

A movie schedule has a movie, a cinema screen in a Cineplex to run, a bunch of show times with a list of seats that has general information and purchase transaction tied to it for each individual show time. We are only able to make this complex ticket booking system by inheriting and reference multiple objects.

Usage of MvC model:

We have clear boundary, control and data (txt) classes and boundary will only interact with other boundary while data can only be accessed through a control class.