Jasmine Koh 33454876

FIT2099 Bootcamp - Requirement 3C

1) OOP Design Principles applied in my design.

I've tried my best to apply all the OOP design principles taught in class into my design, so that my code can more flexible and maintainable. Below are the design principles that I've used:

A. DRY - Don't Repeat Yourself:
The management of bookings are encapsulated within the user's class methods, such as addBookings and removeBookings methods, which are responsible for managing the bookings. This ensures that the logic behind managing the bookings are all defined in one place and can be reused whenever needed, rather than duplicated across different classes and files within my codebase.

B. SOLID –
   a. SRP – Single Responsibility Principle:

   I made sure to design each class so that all classes have only one responsibility. As an example, my Vehicle Class is only for managing vehicle-related data and actions, such as getting the price of the vehicle, and a list of allowable actions that vehicle class can do without putting those actions directly inside the vehicle class.

   b. OCP – Open Closed Principle:

   Interfaces such as Action, Payment and ActionCapable allow my system to be extended without needing to modify existing code.

   c. LSP Liskov Substitution Principle:

   This principal stat that objects of a superclass can be replaceable with objects of a subclass without affecting the correctness of the program. For example, the action interface declares methods of execute and menuDescription, and any class that wants to implement the action interface must provide these two methods. This means that when an Action is expected, an instance of class that implements action interface can be used as well.

   d. ISP Interface Segregation Principle:

   Interfaces are used throughout the system (Action, Payment etc.) which allows different classes to implement methods relevant to them, instead of a single general purpose

interface. By doing so, my code will be more flexible to changes in the future and easier to understand.

e. DIP Dependency Inversion Principle:

This principle means that high level modules shouldn't depend on low-level modules, and they should both be dependent on abstractions instead. For example, my Action interface provides an abstraction for actions that can be performed by the system, and any class that wants to implement Action interface must provide the methods from the Action interface as well. This also makes my system more flexible for future changes.

2) Pros and Cons of my Design

Pros: By using the design principles above, I make sure that my code can be more **maintainable**, where big changes can be made without modifying a big part of the code, hence reducing bugs, and making it easier to understand the code. Adding new features will also be easy with minimal changes to existing code needed as my system is very **extensible**. Interfaces and Abstractions used in my system also ensure reusability in my codebase, hence avoiding code duplication.

Cons: Having to adhere to all the design principles in the beginning can be a little **complicated** and would consume more time and effort in terms of creating a good system that can follow all the principles. There can also be a risk of **over-engineering** where the system would be too strict, this can lead to unnecessary complexity.