Name: Steven Kaing

ID: 33155666

Last Modified: 03/04/2024

Design Rationale

Design Philosophy:

Currently, the vehicle booking system allows users to book vehicles, with each booking incurring a specific cost. However, the design lacks any implementation of payment methods for users to add their funds to their balance account in the system.

The aim of this project requirement is to implement a form of payment functionality for the purpose of adding balance to the user's balance in the system.

1. There are 2 payment methods that need to be implemented: Google Pay and Apple Pay and they all need to process a payment.
2. The payment function should incorporate the SOLID Principles and other core principles to increase quality of design, maintainability, and extensibility of the code.
3. For future extensibility, the design should be able to use any payment service regardless of if they have different implementation.

This rationale is directed towards future developers that may need to extend the payment functionality and implement new features for increased functionality or readability.

Design Reason & Logic:

ApplePay and GooglePay both extended the Payment Interface class.

- Since all different types of payment services process payments and have the same structure (may or may not have same functionality), it is logical to abstract these entities to avoid repetition (DRY).
    - We want to minimize multiple levels of inheritance as it will be difficult to manage if methods are overwritten.
    - If Payment were to be a concrete class, this would imply that there would be some default method of processingPayment. However, we know that payment services have different payment methods, thus different implementations.
    - Let's say a future developer extends the project and wants to make a class called PayPal implement 2 interfaces: Payment and Refundable. This would not be possible if Payment was a class. Thus, an interface here would encourage extensibility in our system, new operations can be easily added by adding more interfaces since each interface has its own responsibility (ISP).
    - Additionally, Payment represents a "behaviour" that is implemented by different payment services. The Payment is not establishing a "is a" relationship, but rather a capability that another class can implement.
        - However, Payment may be implemented as an abstract class, as we can see so far that processPayment implementation have similar implementations and thus can be done in one common parent abstract class.

        - But from a real-world perspective, processPayment in any payment service specifically implements their own SDK. This point aligns more with our goals for easier extensibility when future developers add more complex payment services, that may not have similar implementations to Google or Apply Pay.

- This approach makes the system more extensible and easier to maintain, without modifying existing code.
    - When adding new payment services to our BookingSystem, we do not need to fix the allowableActions method in User class and only code line added are the declaring instance and creating interface (OCP).
    - We want future developers to extend our design and may add additional payment services, i.e. PayPal, that might have different processPayment implementations, which can easily be accommodated with an interface.

- All methods in interfaces are public and methods in abstract classes can have any visibility. In the real-world context, a private processPayment method would make sense as this prevents any private data (payment amount) from potentially leaking. However, in the design, we require AddBalanceAction to have a dependency on the payment services as it needs to access the processPayment method. Thus, logically it would be easier to make Payment methods, public, thus more suitable to implement a Payment interface.

In the User Class, we create a list of payments and add payment services. These are the payment services the user has access to. Thus, it is logical to add this into the User class.

- Creating a payment list inside the User class allows us to pass in an object of type Payment into the AddBalanceAction (Payment payment) in allowableActions.
    - Passing a Payment Object into AddBalanceAction allows any payment service to processPayment and works regardless of what payment you use (LSP)

- We create the instances of the Payment services in the BookingSystem class and added them into payment list in User class.
    - However, doing this makes the BookingSystem take on multiple responsibilities (SRP).
    - It would not make sense to create new instances of Payment services in the User class as it is not the RESPONSIBILITY of the user to create payment methods (BookingSystem instantiates a User).
    - Since BookingSystem had an ASSOCIATION with User class already present, thus instantiating payment instances in the BookingSystem made more sense compared to instantiating in User class.

Why the Alternative Solutions were not optimal?

- Create instance directly in allowableAction and pass it as a parameter in AddBalanceAction.
    - If we were to create multiple payments, then we would have multiple instances in allowableAction, which makes our design hard to interpret for other developers.

- Hard to manage.

Possible Provided Implementation:

```
public class AddBalanceAction implements Action{
    private Payment payment;

    @Override
    public String execute(User user, BookingSystem bookingSystem){
       Scanner scanner = new Scanner(System.in);
       System.out.println("Enter a number to add to your current balance: ");
       float balance = Float.parseFloat(scanner.nextLine());

       System.out.println("Select a payment method: ");
       System.out.println("1) Apple Pay");
       System.out.println("2) Google Pay");
       int selection = Integer.parseInt(scanner.nextLine());
       switch (selection) {
           case 1 -> this.payment = new ApplePay();
           case 2 -> this.payment = new GooglePay();
       }

       if (payment instanceof ApplePay){
           const limit = ((ApplePay) payment).LIMIT
           if (balance <= limit){
               user.addBalance(balance);
               return String.format("%.2f is added to the user balance!", balance);
           } else {
               return "Failed to add to the user balance.";
           }
       }
       else if (payment instanceof GooglePay){
           user.addBalance(balance)
           return String.format("%.2f is added to the user balance!", balance);
       }

       return "";
    }

    @Override
    public String menuDescription() {
        return "Add balance.";
    }
}
```

The provided implementation above is not ideal. Why?

- We can see that the AddBalanceAction class is a “GOD” class as it is taking on too many responsibilities. Here, the class is creating instances of payment services but is also

processing the payment at the same time. This violates the Single Responsibility Principle (SRP), as each module should have responsibility for a single part of the program's functionality.
    - Instead, we should separate the responsibilities into different classes or interfaces.

- The switch statement is an issue here. If we wanted to add a new payment service such as PayPal, we would need to add more lines of code (print statement for new payment method, add a new case in switch statement). Thus, adding a new payment method would result in modifying the existing code, violating the Open/Closed Principle (OCP).
    - Instead, we could pass in the new instance of Payment into AddBalanceAction as a parameter.

- The AddBalanceAction class directly depends on the ApplePay and GooglePay concrete classes. This is an instance of a higher-level module (AddBalanceAction) depending on the lower-level modules (ApplePay and GooglePay) and is not depending on abstractions.
    - This should depend on abstractions, so an interface or abstract class should be utilized, and new instances of the payment services should be passed as parameters into classes via dependency injection.