Bootcamp Rationale

Goals

- Ability to add and use different modes of payment easily.

Rationale

I tried to make it so consoleMenu shows only AddBalanceAction, then when user inputs “a”, only then AddBalanceAction would list out all the available payment methods. Example output:

```
-   a: Add balance.
-   b: View Balance.
-   c: View bookings.
-   d: Confirm bookings.
-   e: Exit the system.
-   f: Add the following vehicle to the Booking System: Booking detail: Brand:
    Lexus | Year: 2023 | Price: $200.00 | Range: 500 km | Fuel Type: ELECTRIC
-   g: Add the following vehicle to the Booking System: Booking detail: Brand:
    BMW | Year: 2022 | Price: $150.00 | Range: 600 km | Fuel Type: PETROL
-   h: Add the following vehicle to the Booking System: Booking detail: Brand:
    Tesla | Year: 2023 | Price: $300.00 | Number of Seats: 5 | Fuel Type:
    ELECTRIC
-   i: Add the following vehicle to the Booking System: Booking detail: Brand:
    Mercedes | Year: 2020 | Price: $500.00 | Number of Seats: 7 | Fuel Type:
    Diesel
-   j: Add the following vehicle to the Booking System: Booking detail: Brand:
    Ford | Year: 2019 | Price: $400.00 | Loading: 1.0
-   k: Add the following vehicle to the Booking System: Booking detail: Brand:
    Volkswagen | Year: 2020 | Price: $450.00 | Loading: 1.2
-   a
-   a: GooglePay
-   b: ApplePay
-   c: etc.
```

But that might have required a whole new PaymentConsoleMenu within BookingSystem, needing a lot of refactoring(time constraints). [depending on what the client wants in terms of the appearance of the console menu].

Instead, AddBalanceAction was changed into an interface, with GooglePay and ApplePay implementing AddBalanceAction. No abstract classes were used in this case, because the execute() method of both payment methods would have been rewritten to fit output, defeating the purpose of using abstract classes.

Interface would ensure that the different payment classes would contain the methods in whilst having their own implementations.

What about Payment?

Payment as an Interface:

- All methods in Payment will be implemented in subclasses
- Multiple inheritance for multiple modes of payment
- On the other hand, interfaces do not allow for default methods. So if any methods are coded the same way, could think about making Payment an abstract class
- So Payment could also be implemented as an abstract class, if say a getPaymentMode() method was introduced so the subsequent modes of payment can be passed to the parent class. I.e. getPaymentMode() in GooglePay would return “Google”, etc.

Payment should be an interface because processPayment for different modes of payment will have different output (“processed by GooglePay / ApplePay), so it would have had to be overridden had it been a concrete class. I chose interface as it would be easier to extend Payment by just adding new ModeOfPayment classes, and implementing them(OCP) according to their specific requirements without changing the initial code. Additionally, using interfaces follows the Dependency Inversion Principle by utilising abstractions instead of concrete implementation.

The provided implementation in bootcamp:

- Fails SRP by having multiple responsibilities within the same class of code, i.e. reading user input at the same time as instantiating new payment objects
- As for OCP, the switch statement has to be modified every time a new mode of payment is added, so original code is modified, disqualifying it.
- LSP: there are no subclasses so cannot be assessed
- ISP states interfaces are to be separated efficiently, but the switch statement does not even use an interface, so it also fails ISP
- The concrete implementation of the types of payment methods is also directly implemented, violating DIP