Design rationale

Design goal

The purpose is to implement a flexible payment system to allow users to add balance to their account. The design of the system must adhere to SOLID principles for maintainability and scalability.

Overall Concept of Design

The design introduces an abstract class Payment, as well as two concrete classes ApplePay and GooglePay that extends from the Payment abstract class. The system currently allows users to add balance to their account using either ApplePay or GooglePay, where ApplePay has a limit of a thousand dollars, while GooglePay has no limit.

Audience:

The audience for this design rationale are the developers working on the booking system, as well as any other stakeholders that will contribute to the system.

Reasoning for Design Decisions

Payment abstract class vs interface

Principle applied: Abstraction

Explanation: Payment is defined as an abstract class as it currently consists of the processPayment method, which is the default behaviour of any payment options, as every payment will need to be processed one way or another. This provides a common method implementation that all subclasses will need to override, ensuring that all subclasses have the processPayment functionality. This allows for abstraction as the implementations of the subclasses that extend the Payment abstract class will be abstracted away.

Alternative

If Payment was defined as an interface, every payment option class will need to implement the payment interface, this is inherently inefficient, as we know that all possible payments will require processing. It also will not utilise the advantage of interfaces, which is being a more flexible way of defining contracts, as it allows classes to implement from multiple interfaces.

AddBalanceAction and Payment abstract class

Principle applied: Open/Closed principle, Dependency Inversion Principle

Explanation: The AddBalanceAction takes in an ArrayList of Payment objects, and iterates through all of the payment objects to display their names for the user to choose their payment options. This utilises the Open/Closed principle as if other types of payment options are added, the AddBalanceAction class will not need to be modified. This makes it much easier to scale the application, as if 1000 more types of payment options are added, we will not need to dramatically change the code. This also follows the dependency inversion

principle, as the AddBalanceAction only has associations with abstractions, not concrete classes.

Alternative
The default code for AddBalanceAction had dependencies on ApplePay and GooglePay, as it was using the instanceof method to check if the payment objects were either GooglePay or ApplePay, this violates the dependency inversion principle. Also, this implementation would also make it extremely difficult to scale the system, as if more payment options are added, many if statements would need to be added for the system to work.

Payment limit for ApplePay:
Principle applied: Encapsulation
Explanation: ApplePay's payment limit is a private attribute, and the class has no getter or setter methods. Hence, it prevents unauthorised parties from having direct access to the attribute, ensuring data integrity and reducing risks of unintended changes.

AddBalanceAction
Principle applied: Single Responsibility Principle
Explanation: The AddBalanceAction class only has a single responsibility, which is to add balance to a user's account.

Pros and Cons of Design:

Pros:

- Clear separations of concerns by utilising packages to store classes relating to that group, makes the code organised and easy to maintain and understand by new developers.
- Design that follows open/closed principles ensures that classes are open for extension but closed for modification, as well as good usage of abstractions, allowing the system to be easily scalable.
- Usage of encapsulation by making attributes private ensures data integrity and security.
- Usage of the single responsibility principle by creating separate classes for each functionality ensures that there are separate classes for each functionality, this ensures that the code will not be well separated and organised.

Cons:

- The introduction of multiple interfaces as well abstract classes may increase complexity of the system.

Conclusion

The design is a good way of implementing that payment functionality within the booking system. By following principles such as the SOLID principles and fundamental object oriented programming concepts, the design is easily maintainable and scalable, as well as have good readability for future development.