Behavioral Design Patterns

Contents

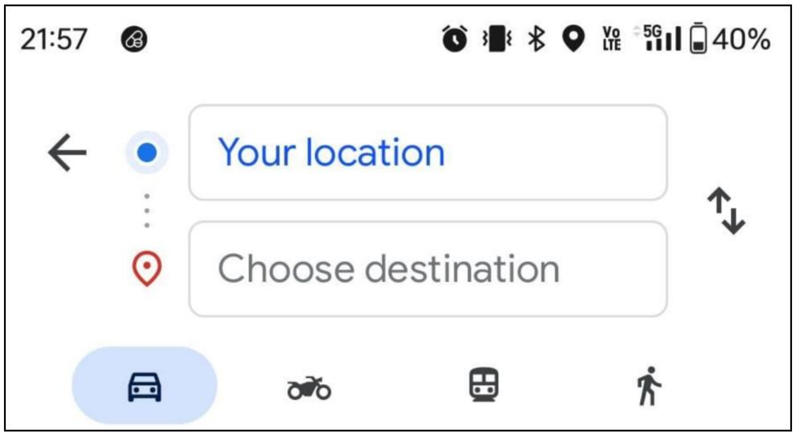
Code and Notes are @ https://github.com/nishithjain/Behavioral_Design_Patternss

1. Introduction

- Based on behaviors/actions/methods.

Strategy Design Pattern

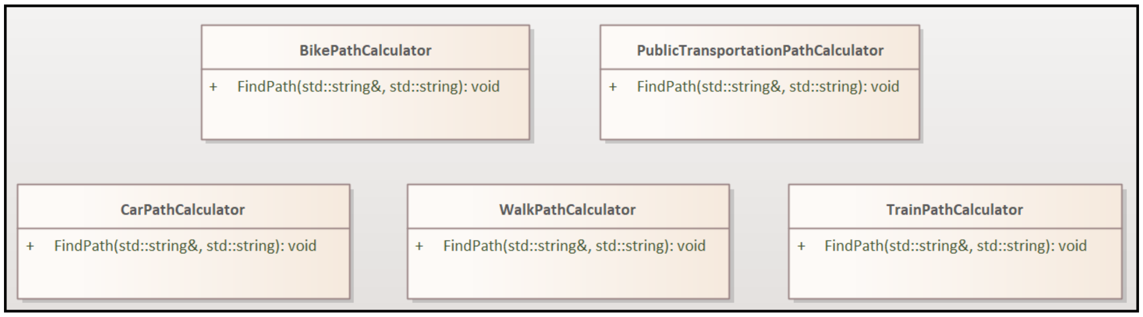
- All of us has used Google maps.
- We can search for a path from point A to point B.
- Also, it will show different transportation mode, such as bike, car, by walk, by cycle and by public transportation.
- Multiple paths are possible depending on mode of transportation.



- If we don't use SOLID principle or Design Patterns, this is how `GoogleMaps class` would look like…

```
class GoogleMaps
{
public:
	void FindPath(const std::string& src, const std::string& dest,
		const TRANSPORTATION_MODE mode) {
		if (mode == TRANSPORTATION_MODE::CAR) {
			// Logic for finding the path by car.
		}
		else if (mode == TRANSPORTATION_MODE::BIKE) {
			// Logic for finding the path by bike.
		}
		else if (mode == TRANSPORTATION_MODE::TRAIN) {
			// Logic for finding the path by train.
		}
		else if(...) {
			...
		}
	}
};
```

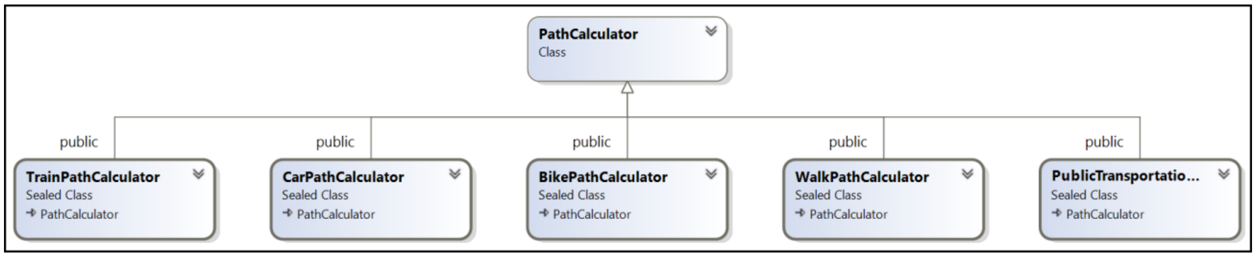
- Above code violates…
  - OCP – If new mode is added, we need to modify the existing code.
  - SRP – The `FindPath()` method is responsible for finding path for all the modes.
- Every way of finding the path is independent of each other.
- Rather than implementing in 1 method, we can implement in separate `class`es.


BikePathCalculator
+ FindPath(std::string&, std::string): void
PublicTransportationPathCalculator
+ FindPath(std::string&, std::string): void
CarPathCalculator
+ FindPath(std::string&, std::string): void
WalkPathCalculator
+ FindPath(std::string&, std::string): void
TrainPathCalculator
+ FindPath(std::string&, std::string): void

- Now inside `GoogleMaps` class, we can create objects or respective classes based on mode.

```
class GoogleMaps
{
public:
	void FindPath(const std::string& src, const std::string& dest,
		const TRANSPORTATION_MODE mode) {
		if (mode == TRANSPORTATION_MODE::CAR) {
			CarPathCalculator ...
		}
		else if (mode == TRANSPORTATION_MODE::BIKE) {
			BikePathCalculator ...
		}
		else if (mode == TRANSPORTATION_MODE::TRAIN) {
			TrainPathCalculator ...
		}
		else if(...) {
			...
		}
	}
};
```

- But this is bad design. We can move this object creation logic to simple/practical factory.
- In order to do that, first we need to create an `PathCalculator` interface/abstract `class`.
- These `CarPathCalculator`, `BikePathCalculator`, `TrainPathCalculator`, etc. will inherit from this interface/abstract `class`.



- Now, we can implement simple factory as shown below…

```
class PathCalculatorFactory
{
public:
	std::unique_ptr<PathCalculator>
		static GetPathCalculatorForMode(TRANSPORTATION_MODE mode);
};


std::unique_ptr<PathCalculator>
PathCalculatorFactory::GetPathCalculatorForMode(const TRANSPORTATION_MODE mode)
{
	if (mode == TRANSPORTATION_MODE::CAR)
		return std::make_unique<CarPathCalculator>();
	if (mode == TRANSPORTATION_MODE::BIKE)
		return std::make_unique<BikePathCalculator>();
	if (mode == TRANSPORTATION_MODE::TRAIN)
		return std::make_unique<TrainPathCalculator>();
	if (mode == TRANSPORTATION_MODE::WALK)
		return std::make_unique<WalkPathCalculator>();
	if (mode == TRANSPORTATION_MODE::PUBLIC_TRANSPORTATION)
		return std::make_unique<PublicTransportationPathCalculator>();
	return nullptr;
}
```

- Now in the `GoogleMaps class`, we can do…

```
class GoogleMaps
{
public:
	void FindPath(std::string& src, const std::string& dst,
		      const TRANSPORTATION_MODE mode)
	{

		const auto pc = PathCalculatorFactory::GetPathCalculatorForMode(mode);
		pc->FindPath(src, dst);

	}
};
```

- This is the Strategy Design Pattern.
- If there are multiple ways of doing something, think of Strategy design pattern.

Observer Design Pattern

Problem Statement:
When a product goes out of stock on an e-commerce website, users can opt to be notified when it becomes available again. The system should allow users to subscribe for notifications and notify all subscribed users when the product is back in stock. User can opt out if they don't want the notification.

- As per the problem statement, we can have a class which manages product availability and notifies users when the product is back in stock. Let's call it ProductNotifier.

- There is a user who want the notification when the product becomes available again.
- The user needs to register by clicking on "**Notify Me**". This means, there should be a method 'Subscribe()' which takes the User object as parameter.
- There can be different type of users. User who wants to get notification via email, via SMS, etc.
- Hence it is better to declare an interface/abstract class and these different type of users can derive from it. When the product becomes available, Notify() method of user will be called.

```
class IUser
{
public:
	virtual ~IUser() = default;
	virtual void Notify(const std::string& product_name) = 0;
};

class ProductNotifier
{
public:
	void Subscribe(std:: shared_ptr<IUser> user);
};
```

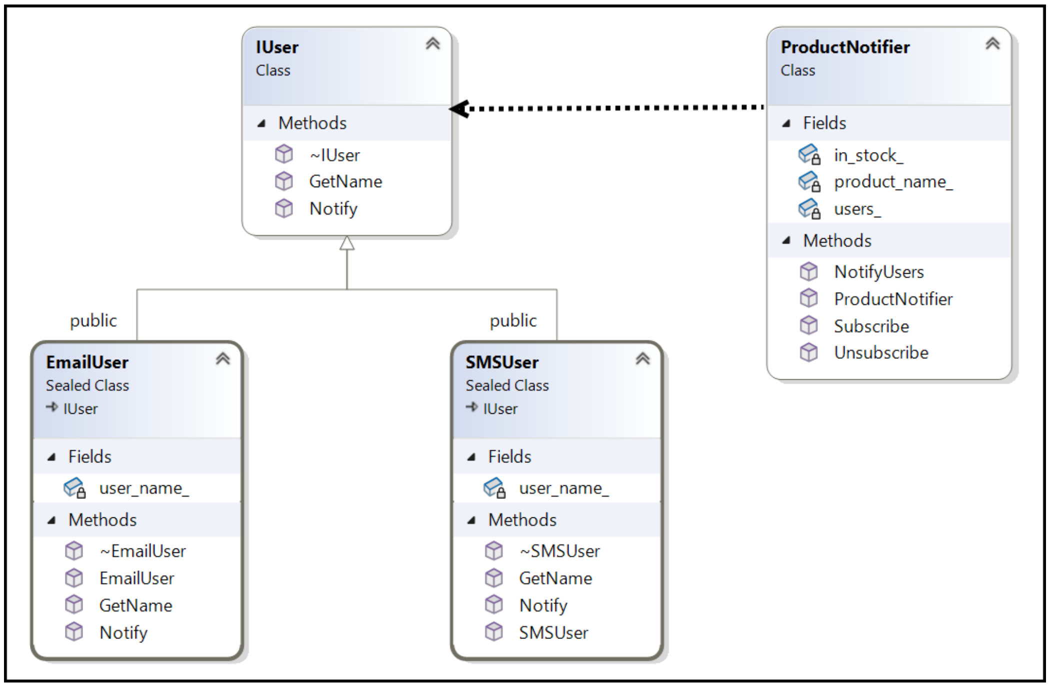
- Since ProductNotifier should notify the User who has registered, it should remember the user. Hence, we need a list to store IUser objects.

```
class ProductNotifier
{
	std::vector<std::shared_ptr<IUser>> users_;
public:
	void Subscribe(std:: shared_ptr <IUser> user);
};
```

- These users will be informed when the product becomes available. Hence, we need to declare a method, which notifies all the users who has registered for notification by going through the list.
- The users can unsubscribe from notification. So, we need to declare a method `Unsubscribe()` which takes the user object who wants to unsubscribe.

```
class ProductNotifier
{
        std::vector<std::shared_ptr<IUser>> users_;
        std::string product_name_;
        bool in_stock_;
public:
        explicit ProductNotifier(std::string product_name);
        void Subscribe(const std::shared_ptr<IUser>& user);
        void NotifyUsers() const;
        void Unsubscribe(const std::shared_ptr<IUser>& user);
        void SetInStock(bool is_in_stock);
};
```

- In summary,
  - `ProductNotifier` (Publisher): Manages product availability and notifies users when the product is back in stock.
  - User (Subscriber) `EmailUser`, `SMSUser`: Represents users who want to be notified when the product becomes available.
  - `IUser` (Observer Interface): Defines a method to be called when users are notified.



- Summary of Issues Without the Observer Pattern:
    - **Tight Coupling**: The `ProductNotifier` would be tightly coupled to specific notification mechanisms, making it rigid and hard to extend.
    - **Poor Scalability**: The class would become bloated as more notification types are added, making the system less scalable.
    - **Violation of OCP**: Constant modifications to core logic would violate the Open/Closed Principle.
    - **Difficulty in Adding New Notification Mechanisms**: New mechanisms would require modifying existing code instead of extending the system.
    - **Hard to Test**: A tightly coupled system is more challenging to test, especially in terms of unit tests.
    - **Lack of Flexibility**: The system becomes less flexible and harder to adapt to new requirements.
    - **Code Duplication**: Without centralization, you might end up duplicating notification logic across the system, leading to maintenance challenges.