Behavioral Design Patterns

Contents

[1. Introduction 2](#_Toc177478915)

[Strategy Design Pattern 2](#_Toc177478916)

[Observer Design Pattern 5](#_Toc177478917)

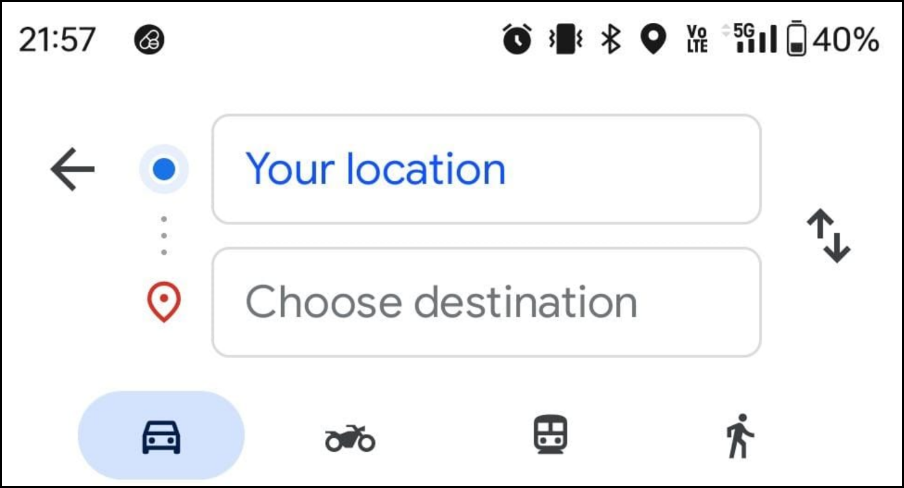
Code and Notes are @ <https://github.com/nishithjain/Behavioral_Design_Patternss>

# Introduction

* Based on behaviors/actions/methods.

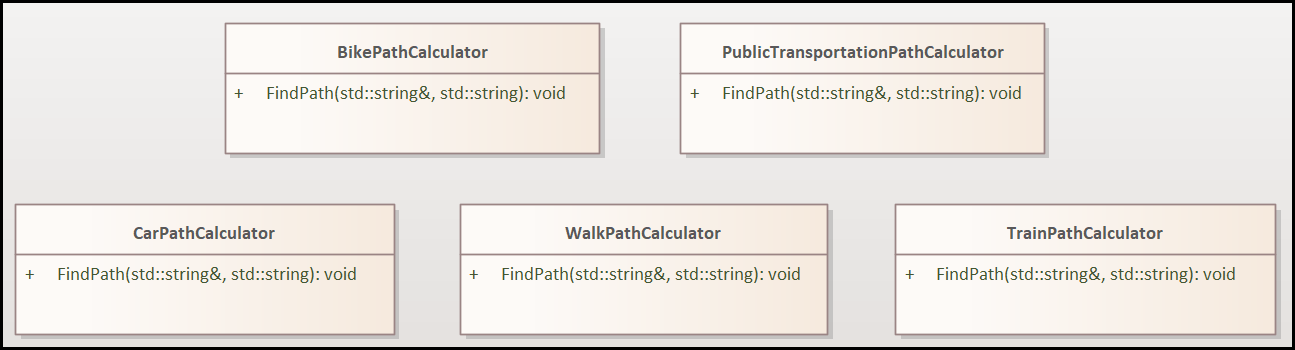
## Strategy Design Pattern

* 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.



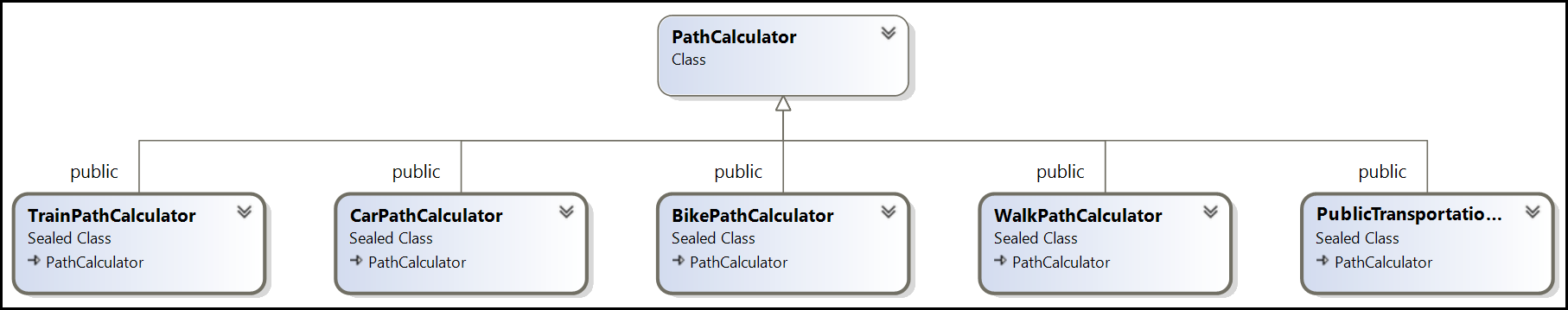
* If we don’t use SOLID principle or Design Patterns, this is how GoogleMaps class would look like…

|  |
| --- |
| 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 classes.
* 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);  }; |

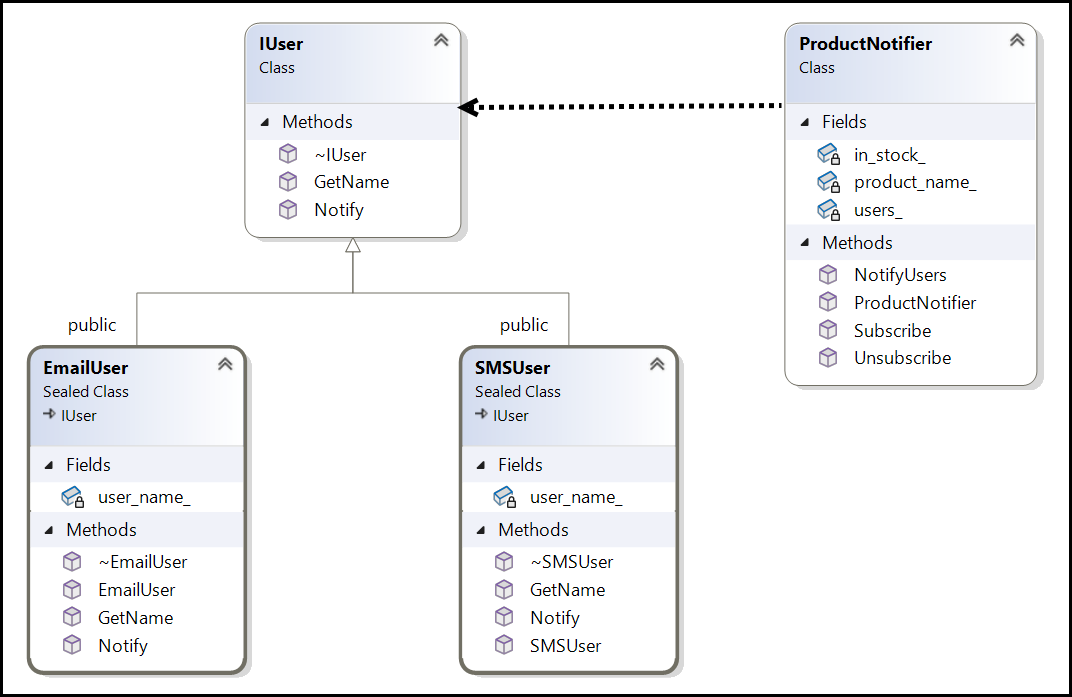
* Since ProductNotifier should notify the User who has registered, it should remember the user. Hence, we need a list to store IUser objects.

|  |
| --- |
| 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.

## State Desing Pattern

* The State Pattern is a behavioral design pattern that allows an **object to change its behavior** when its **internal state changes**.
* The object **will appear to change its class because** it delegates the behavior to different state **objects at runtime**.
* State pattern is particularly useful for **objects that need to change their behavior dynamically** based on their state, and it promotes better organization by encapsulating state-specific behaviors into separate classes.

**Problem Statement:**

Design a document editor application that can transition between three distinct states: **Editing**, **Read-only**, and **Locked**. The application should ensure smooth transitions between these states and enforce the appropriate restrictions on user actions based on the current state.

**Editing**: The user can edit the document.

**Read-only**: The user can only view the document but cannot make changes.

**Locked**: The document is locked and cannot be viewed or edited.

* Context (DocumentContext):
  + This class represents the object whose behavior changes based on its current state.
  + It holds a std::unique\_ptr<IState> pointer, currentState, which can point to any state that implements the IState interface.
  + Through view(), edit(), and lock() methods, DocumentContext delegates behavior to the currentState object. This means it doesn’t directly handle these actions; instead, it asks the state to perform them.

|  |
| --- |
| class DocumentContext  {  std::unique\_ptr<IState> currentState;  public:  DocumentContext(std::unique\_ptr<IState> initialState);  void setState(std::unique\_ptr<IState> newState);  void view();  void edit();  void lock();  }; |

* + setState(): This method lets DocumentContext transition to a different state by assigning a new IState object to currentState.
  + view(), edit(), and lock(): These methods delegate behavior to currentState. If edit() or lock() is not supported by the current state, they throw an error (caught by DocumentContext).
* State Interface (IState):
  + IState is an abstract base class that defines the interface for different states.
  + It declares three methods: view(), edit(), and lock(). These methods provide a default implementation (for example, throwing an error in edit and lock), which specific states can override.
  + Each state that implements IState can override these methods to provide unique behavior.

|  |
| --- |
| #include <stdexcept>  class DocumentContext;  class IState {  public:  virtual ~IState() = default;  virtual void view(DocumentContext& context) = 0;  virtual void edit(DocumentContext& context) {  throw std::runtime\_error(  "Editing is not allowed in the current state.");  }  virtual void lock(DocumentContext& context) {  throw std::runtime\_error(  "Locking is not allowed in the current state.");  }  }; |

* Concrete State Classes (EditState, ViewState, LockedState):
  + These are the specific states that DocumentContext can be in, each implementing specific behaviors for view(), edit(), and lock().

|  |
| --- |
| class EditState : public IEditableState  {  public:  void view(DocumentContext& context) override;  void edit(DocumentContext& context) override;  };  class ViewState : public IViewableState {  public:  void view(DocumentContext& context) override;  };  class LockedState : public ILockableState {  public:  void view(DocumentContext& context) override;  void lock(DocumentContext& context) override;  }; |

A screenshot of a computer

Description automatically generated

* Below code demonstrates how the DocumentContext behaves differently based on the current state:

|  |
| --- |
| DocumentContext document(std::make\_unique<EditState>());  std::cout << "Current State: Editing\n";  document.view(); // Should allow viewing  document.edit(); // Should allow editing  // Change to ReadOnly state  document.setState(std::make\_unique<ViewState>());  std::cout << "\nCurrent State: ReadOnly\n";  document.view(); // Should allow viewing in read-only mode  document.edit(); // Should not allow editing  // Change to Locked state  document.setState(std::make\_unique<LockedState>());  std::cout << "\nCurrent State: Locked\n";  document.view(); // Should deny viewing  document.edit(); // Should deny editing |