# Behavioural design patterns



### Behavioural Design Patterns

> Behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

### Observer

> The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

### Strategy

> The strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family.

## Observer Pattern

Imagine it is iPhone season again, where Apple releases a new version of the iPhone and millions of individuals can't wait to get there hands on it. The stock at the local store has not yet been updated, but you want to be the first to know when the new iPhone is available. You can go to the store every day to check if the stock has been updated, but that is wasteful. Another option is that the store sends everyone an

email when the new phones come in. Again, it is wasteful since not everyone is interested in the new iPhone. The best option is that you register or subscribe to the store's mailing list. When the new iPhone comes in, the store sends an email to everyone on the mailing list. This is the motivation behind the Observer pattern.

We now want to build a Bitcoin tracking application that sends out emails or tweets when the price of Bitcoin changes. We have a data model for Bitcoin that contains the current price of Bitcoin. Apart from this we have a `BitcoinTracker` class that is responsible for setting the price of the Bitcoin.

```
class BitcoinTracker:
    def __init__(self, bitcoin):
        self.bitcoin = bitcoin

    def set_price(self, price):
        self.bitcoin.set_price(price)
```

Now we want to send an email when the price of Bitcoin changes. We can do this by calling the `sendEmail` method in the setter method of the `BitcoinTracker` class.

```
class BitcoinTracker:
    def __init__(self, bitcoin):
        self.bitcoin = bitcoin

    def set_price(self, price):
        self.bitcoin.set_price(price)
        self.send_email()
```

The above implementation works but it is not ideal. The `BitcoinTracker` class has two responsibilities. It is responsible for setting the price of Bitcoin and it is responsible for sending an email. The `BitcoinTracker` class violates the Single Responsibility Principle. Similarly, we can also send a tweet when the price of Bitcoin changes. We can do this by calling the `sendTweet` method in the setter method of the `BitcoinTracker` class. This now violates the Open-Closed Principle. We will have to change the code in multiple places if we want to add a new feature.

```
class BitcoinTracker:
    def __init__(self, bitcoin):
        self.bitcoin = bitcoin

    def set_price(self, price):
        self.bitcoin.set_price(price)
        self.send_email()
        self.send_tweet()
```

Another option is to have a secondary class fetch the price of Bitcoin and send an email when the price changes. This is known as polling. The problem with any polling approach is that it is wasteful. The secondary class will have to poll the BitcoinTracker class every few seconds to check if the price has changed.

```
class BitcoinPoller:
    def __init__(self, bitcoin_tracker):
        self.bitcoin_tracker = bitcoin_tracker
        self.previous_bitcoin = bitcoin_tracker.get_bitcoin()

    def poll(self):
        current_bitcoin = self.bitcoin_tracker.get_bitcoin()
        if current_bitcoin.price != self.previous_bitcoin.price:
            self.send_email()
        self.previous_bitcoin = current_bitcoin
```

The two approaches we have discussed so far are not ideal. The first approach violates the Single Responsibility Principle. The second approach is wasteful. The Observer pattern suggests that you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## Implementation

1. `Observable interface` - This interface defines the methods that the subject class must implement. The subject class is responsible for notifying the observers when the state of the subject changes.

```
from abc import ABC, abstractmethod

class Observable(ABC):
    @abstractmethod
    def add_observer(self, observer) -> None:
        pass

    @abstractmethod
    def remove_observer(self, observer) -> None:
        pass

    @abstractmethod
    def notify_observers(self) -> None:
        pass
```

2. `Observer interface` - This interface defines the methods that the observer class must implement. The observer class is responsible for updating itself when the state of the subject changes.

```
public interface class Observer {
    void notify();
}
```

```
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def notify(self) -> None:
        pass
```

3. `Concrete observables` - These are the classes that implement the `Observable` interface. The `BitcoinTracker` class is a concrete observable. The `BitcoinTracker` class is responsible for notifying the observers when the state of the subject changes.

```
class BitcoinTracker(Observable):
    def __init__(self, bitcoin):
        self.bitcoin = bitcoin

    def set_price(self, price):
        self.bitcoin.set_price(price)
        self.notify_observers()
```

To simplify the code and provide better interfaces, we can borrow from the registry pattern and register observers and even add utility methods to the `Observable` interface.

```
from abc import ABC, abstractmethod

class Observable(ABC):
    def __init__(self):
        self.observers = []

    def register(self, observer):
        self.observers.append(observer)

    def deregister(self, observer):
        self.observers.remove(observer)

    def notify_observers(self):
        for observer in self.observers:
            observer.notify()
```

4. `Concrete observers` - These are the classes that implement the `Observer` interface. The `EmailSender` class is a concrete observer. The `EmailSender` class is responsible for updating itself when the state of the subject changes.

```
class EmailSender(Observer):
    def __init__(self, bitcoin):
        self.bitcoin = bitcoin

    def notify(self):
        self.send_email()
```

5. `Client` - The client is responsible for creating the subject and the observers. The client is also responsible for registering the observers with the subject.

```
def main():
    bitcoin_tracker = BitcoinTracker()
    email_sender = EmailSender()
    bitcoin_tracker.register(email_sender)
```

### Recap

- There are often times when you want to notify other objects when the state of an object changes. The Observer pattern suggests that you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- In-place updates and polling are not ideal. In-place updates violate the Single Responsibility Principle whereas polling is wasteful.
- To implement the Observer pattern, define an `Observable` interface that defines the methods that the subject class must implement.
- Define an `Observer` interface that defines the methods that the observer class must implement. The subject class is responsible for notifying the observers when the state of the subject changes. The observer class is responsible for updating itself when the state of the subject changes.

## Strategy design pattern

> The strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern encapsulates alternative algorithms (or strategies) for a specific task and supports their interchangeable use. The strategy pattern lets the algorithm vary independently from clients that use it.

Today we are building a navigation system for a car. The navigation system should be able to calculate the shortest route between two points and tell the driver how to get there.
We start with a simple implementation that only supports driving on roads and hence we design a simple `RoadNavigation` class that calculates the shortest route between two points on a road.

```
class Navigator:
    def navigate(self, from: Point, to: Point) -> None:
        ...
```

Our application is a hit and we get a lot of requests to support other types of navigation such as using a bike or walking. In order to support various types of navigation we introduce a new `ModeType` enum that defines the different types of navigation.

```
class Navigator:
    def navigate(self, from: Point, to: Point, mode: ModeType) -> None:
        if mode == ModeType.ROAD:
            ...
        elif mode == ModeType.BIKE:
            ...
        elif mode == ModeType.WALK:
            ...
```

This works but it is not very flexible. If we want to add a new type of navigation we have to modify the `navigate` method. This is an example of a violation of the open-closed principle. Similarly, the method has multiple reasons to change, and we have to test it for all the different types of navigation.

We could create an interface and have a separate class for each type of navigation. This would solve the problem of having to modify the `navigate` method, but inheritance is static and we cannot change the behavior of the `navigate` method at runtime. Also code duplication is a problem.
This is where the strategy pattern comes in. The strategy pattern allows us to encapsulate the different types of navigation in separate classes and select the appropriate one at runtime. The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called strategies. The original class, called context, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

## Implementation

1. `Strategy` interface - defines an algorithm interface common to all supported versions.

```
from abc import ABC, abstractmethod

class NavigationStrategy(ABC):
    @abstractmethod
    def navigate(self, from: Point, to: Point) -> None:
        pass
```

2. `Concrete Strategy` classes - implement the algorithms using the Strategy interface.

```
class RoadNavigation(NavigationStrategy):
    def navigate(self, from: Point, to: Point) -> None:
        ...
```

3. `Context` class - maintains a reference to a Strategy object and defines an interface that lets the strategy access its data.

```
class Navigator:
    def __init__(self, strategy: NavigationStrategy):
        self.strategy = strategy

    def navigate(self, from: Point, to: Point) -> None:
        self.strategy.navigate(from, to)
```

4. `Client` - creates and configures the context and the strategy objects.

```
def main():
    navigator = Navigator(RoadNavigation())
    navigator.navigate(Point(0, 0), Point(10, 10))
```

To create a strategy object we can also use a factory method.

## Recap

- The strategy pattern encapsulates alternative algorithms (or strategies) for a specific task and supports their interchangeable use.
- Implementing it in place violates the open-closed principle and makes the code harder to maintain.

- Inheritance is static, and we cannot change the behavior of the system at runtime.
- To use the strategy pattern we create a strategy interface and a set of classes that implement it.
- The context class maintains a reference to a strategy object and delegates the work to it.