The Observer Design Pattern

@Ankitz007

Definition

*The **Observer Pattern** is a behavioral design pattern that defines a one-to-many dependency between objects. This means that when one object (the subject) changes its state, all its dependent objects (the observers) are automatically notified and updated.*

📹 Imagine You're a YouTuber (The Subject)

You post new videos whenever you have something cool to share – vlogs, gameplays, tutorials, etc. 🎮📸📚

Now you've got subscribers:

- *A student who watches tutorials 📚*
- *A gamer who loves your let's plays 🎮*
- *A chef who just likes your vibe 👨‍🍳*

These subscribers are the Observers 👀.

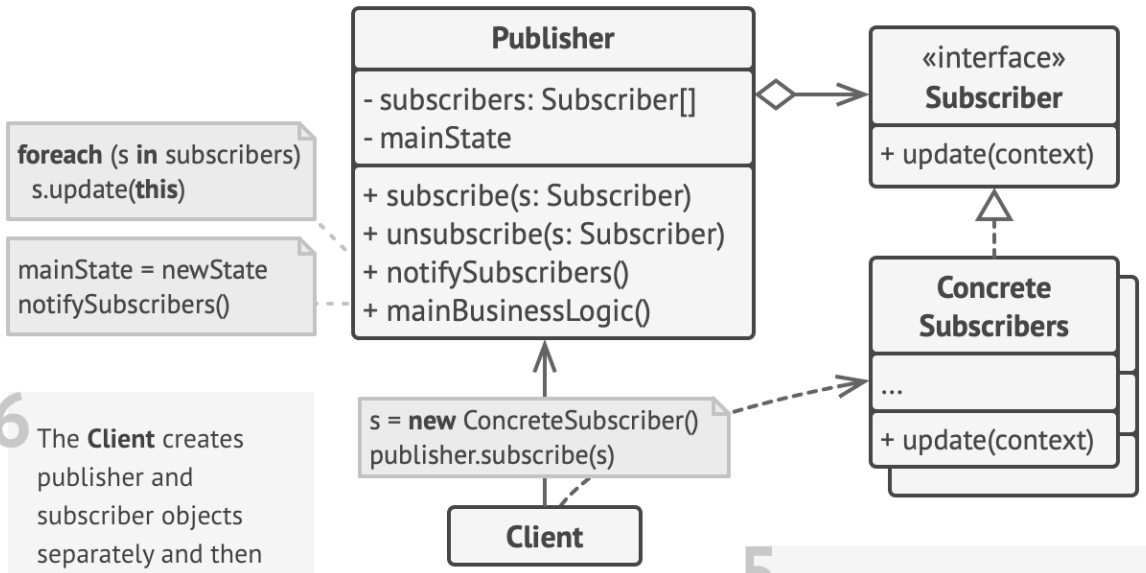
Now, Let's Relate This to our pattern 💻

- *You (the YouTuber) <- the Subject*
- *Subscribers <- Observers*
- *Uploading a new video <- State change*
- *Notifications going out <- Automatic updates to observers*

Structure

1 The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

2 When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.



3 The **Subscriber** interface declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.

4 **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

6 The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Source: *Refactoring.Guru*

@Ankitz007

Key Characteristics

Loose Coupling *- The subject and observers are only aware of each other's interfaces, not their concrete implementations.*

- ***Benefit****: Enhances modularity and maintainability by allowing components to evolve independently without breaking each other.*

Dynamic Relationships *- Observers can be added or removed from the subject at runtime without affecting the subject's core functionality.*

- ***Benefit****: Increases system flexibility and adaptability, making it easy to modify or extend behavior dynamically.*

Automatic Updates *- Observers are automatically notified whenever the subject's state changes, without any manual intervention required.*

- ***Benefit****: Ensures real-time synchronization between objects, minimizing the risk of outdated or inconsistent data.*

Broadcast Communication *- The subject simply iterates over the observer list and calls a common update method on each.*

- ***Benefits****: This ensures consistent notification across multiple dependents efficiently.*

@Ankitz007

When to use?

✅ ***Multiple components need to stay in sync with shared data*** *- Keep several parts of the system updated when data source changes.*
Use Case: *A weather station updates temperature and humidity, instantly pushing updates to apps, displays, and websites.*

✅ ***Real-time event-driven updates*** *- Ideal for systems that require immediate reaction to changes from users or external data streams.*
Use Case*: A stock trading platform pushes live stock price updates to mobile apps, web dashboards, and alert systems simultaneously.*

✅ ***Publish/subscribe communication*** *- Allows a subject to notify all subscribers of changes without knowing who or how many they are.*
Use Case*: A chat server broadcasts new messages to all users subscribed to a specific group chat.*

✅ ***Separation of data logic and UI is desired*** *- Perfect for keeping your UI reactive and modular by syncing it with underlying models automatically.*
Use Case*: A to-do list app updates task counters, views, and notifications whenever a task is added or marked as complete.*

@Ankitz007

When NOT to use?

❌ ***When observer relationships become too complex to manage***

- *If observers depend on multiple subjects in intricate ways, the system becomes hard to debug.*

❌ ***Updates are extremely frequent, and performance is critical***

- *Notifying many observers very often can lead to performance bottlenecks (the "update storm" problem).*

❌ ***Updates are infrequent and real-time updates aren't needed***

- *If updates are rare, using polling or simple checks might be simpler and more efficient than setting up an observer structure.*

❌ ***When changes in the observer might trigger updates back to the subject***

- *Circular update dependencies can cause infinite loops.*

❌ ***When tight synchronization or specific ordering is required***

- *The pattern doesn't guarantee delivery order or precise timing of notifications*

@Ankitz007

Code Example

```python
# Subject
class Subject:
    def __init__(self):
        self._observers = []
    def attach(self, observer):
        self._observers.append(observer)
    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

# Observer
class Observer:
    def update(self, message):
        pass  # To be implemented by concrete observers

# Concrete Observers
class EmailObserver(Observer):
    def update(self, message):
        print(f"📧 Email received: {message}")
class SMSObserver(Observer):
    def update(self, message):
        print(f"📱 SMS received: {message}")

# Usage
if __name__ == "__main__":
    subject = Subject()
    email_observer = EmailObserver()
    sms_observer = SMSObserver()
    subject.attach(email_observer)
    subject.attach(sms_observer)
    # State change triggers notification
    subject.notify("New Promotion: 50% OFF Sale!")
```


@Ankitz007

Real World Examples

Social Media Feeds (Twitter/X, Facebook, Instagram) 📱

- *Subject: A User's profile/account (specifically, the action of posting).*
- *Observer: The feeds of users who follow the Subject user.*
- *Flow: The platform's backend logic updates the feeds (observers) of followers to include new posts.*

Weather Monitoring Systems 🌧️❄️

- *Subject: The Weather Station*
- *Observer: Weather websites, mobile weather apps, agricultural systems, news tickers displaying weather.*
- *Flow: The data source pushes updates to all subscribed applications/displays.*

Event Management System 🎟️

- *Subject: Event (e.g., a live concert or sports game).*
- *Observers: Ticket Vendors, Social Media Feeds, Sponsors.*
- *Flow: When the event status changes (e.g., sold-out, new ticket available), all observers are notified.*

@Ankitz007

Real World Examples

Online Banking System 💰

- *Subject: Bank Account (balance changes).*
- *Observers: Mobile App, Email Alerts, SMS Service.*
- *Flow: When the account balance changes (deposit/withdrawal), all observers are notified and updated with the new balance.*

Health Monitoring System 🏥

- *Subject: HealthMonitor that collects vital signs from sensors*
- *Observers: Doctor's dashboard, Patient app, Emergency alert system*
- *Flow: When vital signs change significantly, all monitoring interfaces update and appropriate alerts trigger.*

IoT Sensor Network

- *Subject: SensorHub that collects data from various sensors*
- *Observers: Data logger, Alert system, Visualization dashboard*
- *Flow: When sensors detect changes, all monitoring systems receive updates to process or display*