Observer Pattern

The Observer Pattern defines a one-to-many dependency relationship between objects, where one object (the Subject) maintains a list of dependents (Observers) and notifies them of state changes. This pattern enables dynamic relationships between objects without requiring them to be tightly coupled.

Back to Agenda Page

How Does the it Work?

- Subject: Represents the core component that maintains a list of observers and sends notifications when its state changes.
- Observer: Defines an interface that allows observers to receive updates from the subject.
- ConcreteObserver: Implements the Observer interface and registers with a subject to receive notifications.

Back to Agenda Page

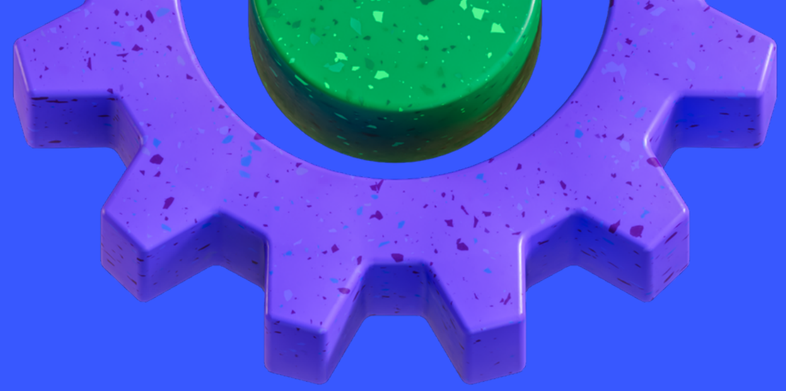
Example Scenario: News Agency

Consider a news agency where readers subscribe to receive updates about breaking news, sports, and entertainment. The news agency acts as the subject, and readers are observers who receive notifications whenever new articles are published in their subscribed categories.

Back to Agenda Page

Benefits of Using the Observer Pattern

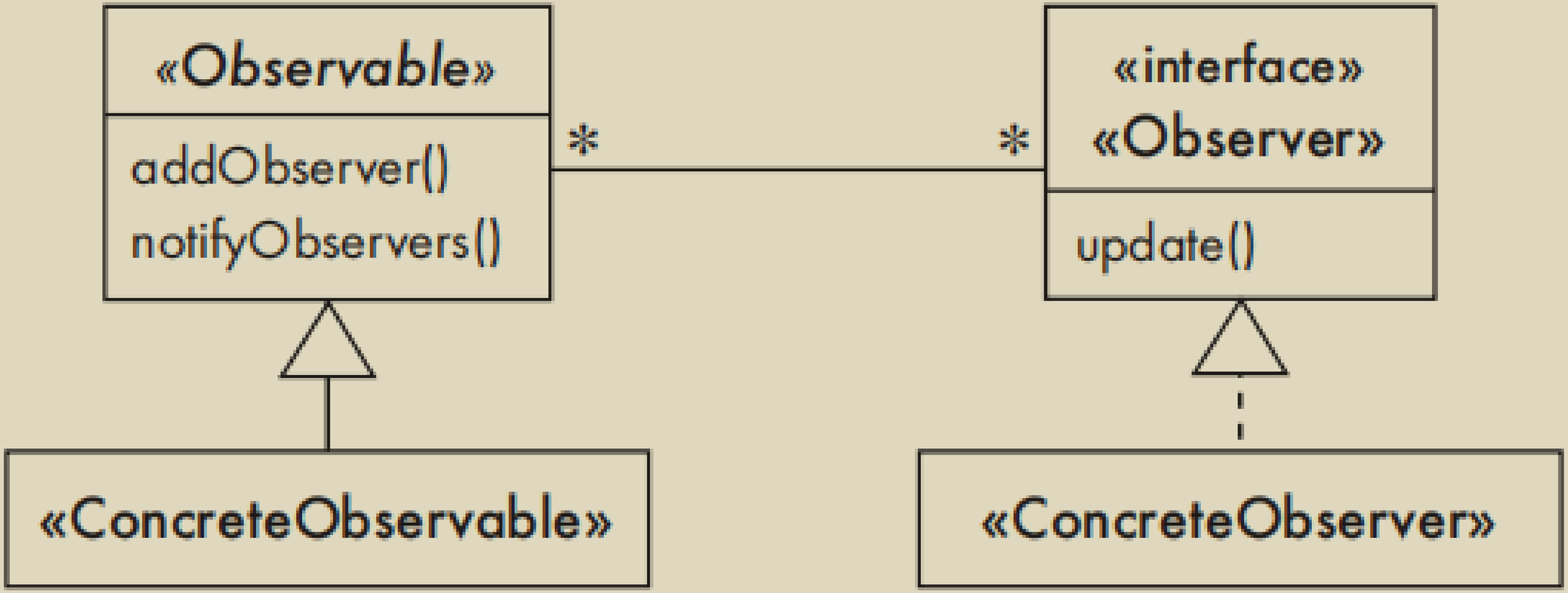
- Loose Coupling: Allows for flexible relationships between the news agency and readers, enabling easy addition or removal of subscribers without affecting other parts of the system.
- Real-time Updates: Enables timely delivery of news articles to subscribers as soon as they are published by the news agency.
- Scalability: Supports a dynamic number of subscribers and diverse subscription categories.



Common Use Cases

The Observer Pattern is applicable in various scenarios, including:

- Implementing news subscriptions and notifications.
- Building chat applications with real-time messaging.
- Integrating event-driven architectures in distributed systems.


«Observable»
addObserver()
notifyObservers()
*
*
«interface»
«Observer»
update()
«ConcreteObservable»
«ConcreteObserver»

Conclusion

TThe Observer Pattern promotes modularity and flexibility in software design by facilitating dynamic relationships between publishers and subscribers. By leveraging this pattern, developers can create scalable and responsive systems that efficiently manage communication between components.