Observer

A behavioral pattern

Learning goals

1. Understand the idea of behavioral patterns.
2. Learn the idea, structure, and Java implementation of the Observer design pattern.
3. Learn to apply the Observer DP in your own programming.

Behavioral patterns

- **Behavioral patterns** provide guidelines for implementing the communication and interplay of objects.
- These patterns focus on making the assignment of responsibilities between objects easier.
- Idea: provide clever structures (composition, association) to make it possible for objects to achieve complex goals in a well-managed way.
- Behavioral DPs: Observer, Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, State, Strategy, Template Method, Visitor.

Idea of Observer

- The Observers want to be notified as soon as a Subject changes.
- Example: in the GUI following the MVC model:
  - The model classes hold the data of interest.
  - As the data changes, the GUI (that displays the data) should be notified of the change.
- In the Observer DP, the Observers register themselves for the Subject whose changes they are interested in.
- Once the Subject's status changes, all Observers will be notified immediately.
- Real life equivalents
  - Subscribing a newsletter
  - Following a user in Instagram

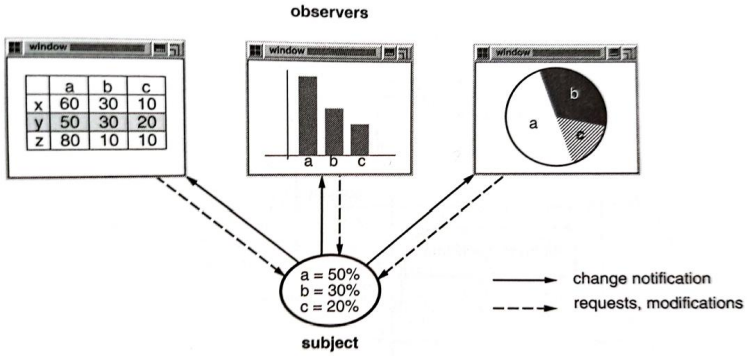


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 293

General structure

Structure

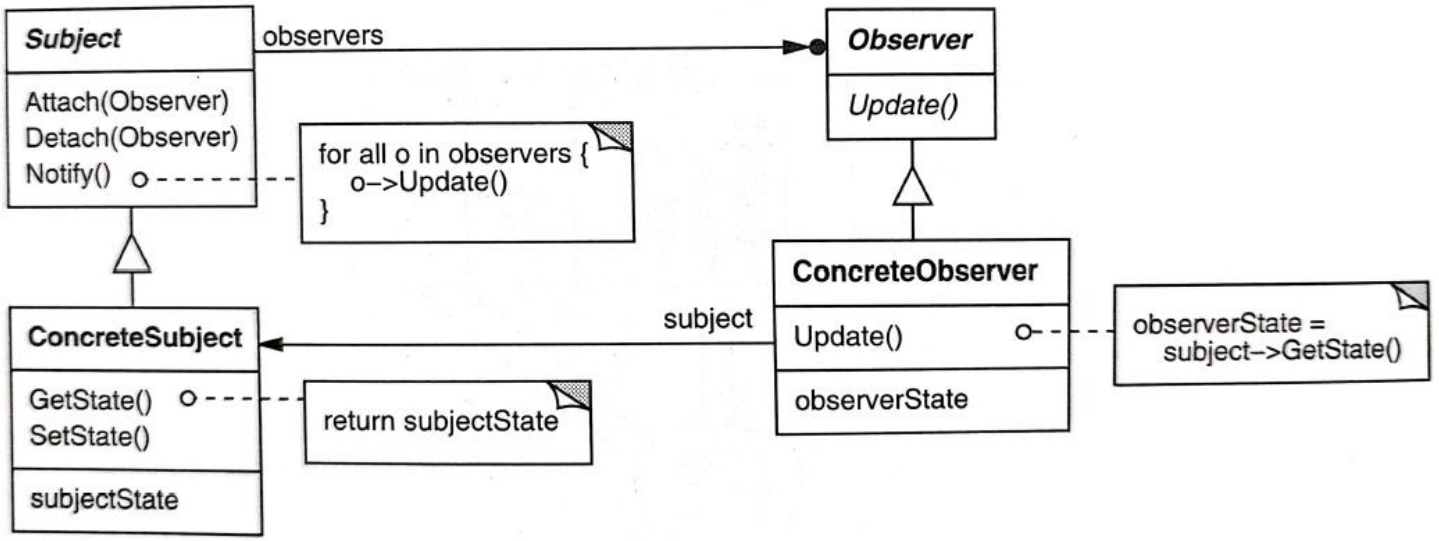
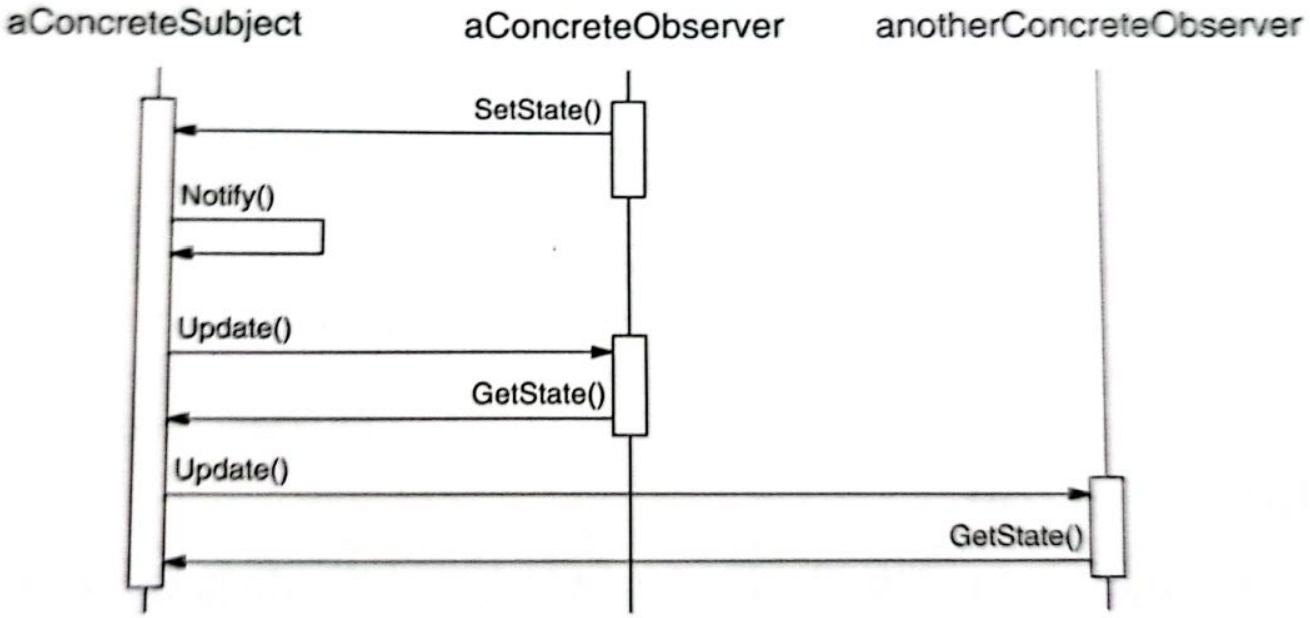


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 294

Roles

- **Subject**: Base class for objects being followed.
  - Implements methods for the bookkeeping of the observers as well as notifying them
- **Concrete Subject:** Implements the concrete operations of the subject (other than managing the observers).
- **Observer**: Interface that declares the method for receiving the notification
- **Concrete Observer**: Implements the method for receiving the notification. As part of this, fetches the current, changed status of the Subject.

Sequence diagram



- All observers are notified as the Subject's state changes.
- Then, all observers query for the new state to make sure that they have the latest status of the Subject.

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 295

Who initiates the notifications?

- Option 1: The state-setting methods in the Subject initialize the notifications.
  - In Java, the setter method in the Subject may then call **notifyAll()** method.
  - This guarantees that all changes of the subject are immediately propageted to the observers.
  - May create a lot of method calls.
- Option 2: The clients are responsible for initializing the notifications.
  - A client can make may changes to the state of the Subject, and then call **notifyAll()** only once.
  - This may make updates smoother, as the Subject's state can be updated efficiently as a batch process, after which the observers are notified only once.

Push vs pull models

- In the **pull model**, the observers are responsible for querying the new state of the Subject after receiving an **update()** method call.
  - Only a barebones update is sent, telling that the Subject's data has changed.
- In the **push model**, the updated data is sent as the parameter of the **update()** method call.
- Also intermediate models are possible: the amount of data sent as the **update()** call's parameter may vary.

Other variants

- The Observer may be interested in only certain types of status changes
  - This can be communicated in the registration phase.
- An Observer may need to observe more than one Subject.

Java considerations

- In Java,
  - Subject can often be implemented as a non-abstract class.
  - Observer is typically an interface with one declared method, **update()**.
  - The ConcreteObservers just implement the Observer interface, specifying the update() method.
  - The constructor in the ConcreteObserver can register the object as an observer by calling the Subject's method.
- JDK 8 had a similar Observable/Observer implementation in **java.util** package.
  - That was marked as deprecated due to the limitations in guaranteeing the order of notifications.
  - Yet, the design pattern is up-to-date and very frequently used.
  - Alternative, advanced implementations exist, or the user can easily code a solution that matches their needs.