Observer Design Pattern

Purpose and Motivation

The Observer pattern defines a one-to-many dependency between the subject(one) and the observers(many) so that when the subject changes state, all its observers are notified and updated accordingly. The main motivation behind this design pattern is to maintain consistency between related objects while also avoid coupling as much as possible.

Intended Use Cases

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

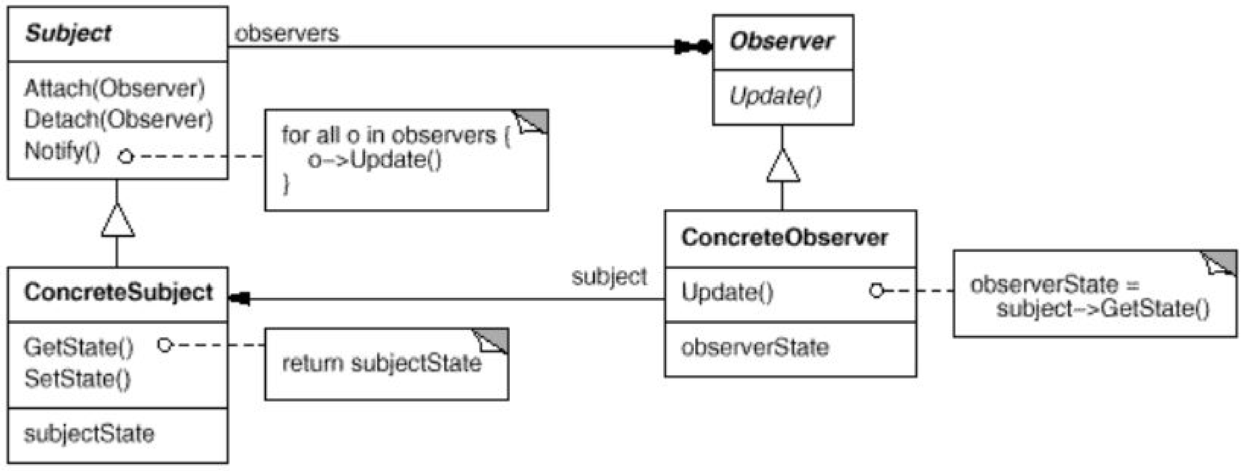
Vocabulary

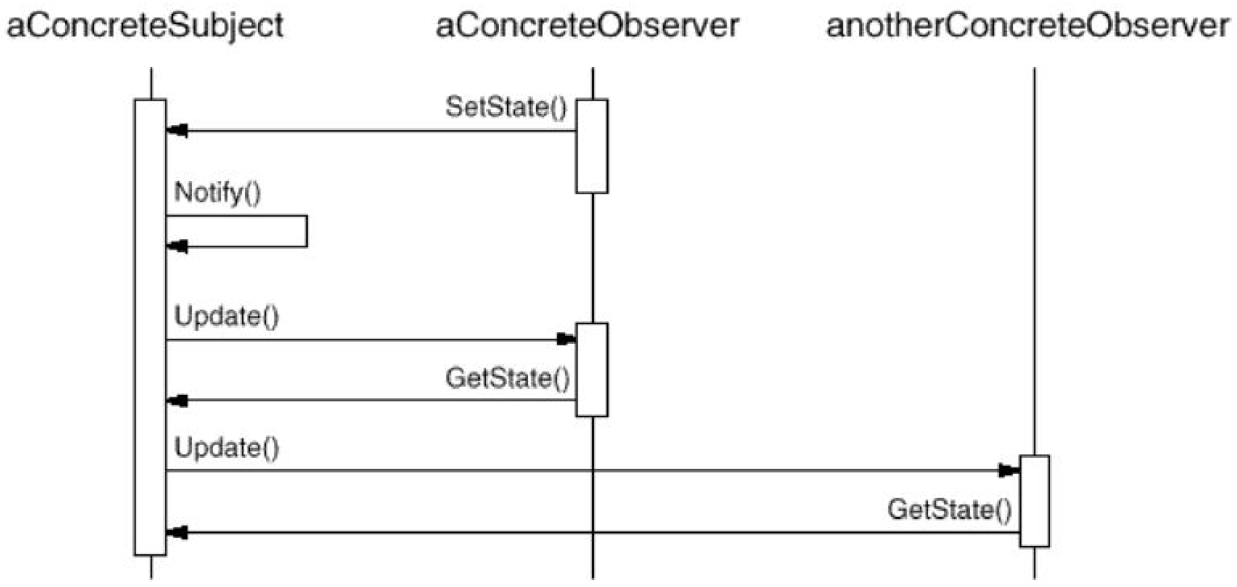
Publish-Subscribe (different name for this design pattern)

- Subject
    - The information keeper of data or of business logic.
- Observer
    - The entities that need to be updated when the Subject changes.
- Register/Attach
    - Observers register themselves to the Subject because they want to be notified when there is a change.
- Event
    - Events act as a trigger in the Subject such that all the observers are notified.
- Notify
    - Depending on the implementation, the Subject may “push” information to the observers, or, the observers may “pull” if they need information from the Subject.
- Update

- Observers each update their state independently from other observers, depending on the triggered event.

Structure and Runtime Behaviour





Known Consequences

Positive Consequences

- ***Abstract coupling between Subject and Observer***. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract 275 Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
- ***Support for broadcast communication***. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

Negative Consequences

- ***Unexpected updates***. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

NFPs

Improved NFPs

- **Scalability** - Attaching more observers of a subject can be fairly easy which makes it further scalable. Each subject can have a list of observers observing it, so add one more to the list doesn't affect the subject or the other observers on the list.

Inhibited NFPs

- **Efficiency** - Inefficiency is a potential problem in the observer pattern, especially when the subject is also an observer. The reason why it can be a problem is that observers may receive notification repeatedly when subjects and observers are linked and intertwined in a sophisticated chain.

Credits

Students:

Frontier (Zi Bai, Tina Li, Ryan Wong)

Biting Bulldogs (Breandan Choi, Karan Patel, Wangsen Tian, Siman Wu)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal