Observer Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the use of the Observer Design Pattern
- Use and implement the Observer Pattern

Inspiration

"One principle problem of educating software engineers is that they will not use a new method until they believe it works and, more importantly, that they will not believe the method will work until they see it for themselves."
– Humphrey, W.S.

The Importance of Shared Vocabulary

- Design Patterns are important because they provide a shared vocabulary to software design
  - (In addition, to being really useful solutions to tricky design problems!)
- Compare:
  - So I created this broadcast class. It tracks a set of listeners and anytime its data changes, it sends a message to the listeners. Listeners can join and leave at any time. It's really dynamic and loosely-coupled.
- With:
  - I used the Observer Design Pattern

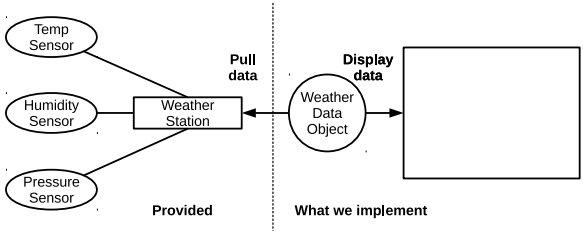
The Importance of Shared Vocabulary

- Shared pattern vocabularies are powerful
  - You communicate not just a name, but a whole set of qualities, services, and constraints associated with the pattern
- Patterns allow you to say more with less
  - Other developers quickly pick up on the design you are proposing
- Talking about patterns, lets you “stay in the design” longer
  - You don’t have to get into nitty gritty details, just how your classes map into the roles provided by the pattern
- Shared vocabularies can empower your development team
  - Experienced team members can talk about design more quickly; junior programmers are motivated to get up to speed, so they can influence the design of the target system.

Observer Pattern

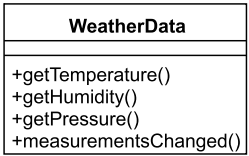
- Don’t miss out when something interesting (in your system) happens!
  - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
  - Its dynamic in that an object can choose to receive notifications or not at run-time.
  - Observer happens to be one of the most heavily used patterns in Java

Example Weather Monitoring}



We need to pull information from the station and then generate “current conditions, weather stats, and a weather stats, and a weather forecast”.

WeatherData Skeleton



We receive a partial implementation of the WeatherData class from our client.

They provide three getter methods for the sensor values and an empty `measurementsChanged()` method that is guaranteed to be called whenever a sensor provides a new values

We need to pass these values to our three displays… so that's simple!

First pass at measurementsChanged

```
public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay
        .update(temp, humidity, pressure);
    statisticsDisplay
        .update(temp, humidity, pressure);
    forecastDisplay
        .update(temp, humidity, pressure);
}
```

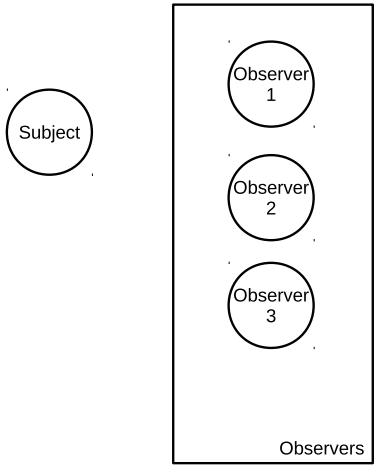
Problems?

1. **The number and type of displays may vary**. These three displays are hard coded with no easy way to update them.
2. **Coding to implementations, not an interface!** Although each implementation has adopted the same interface, so this will make translation easy!

Observer Pattern

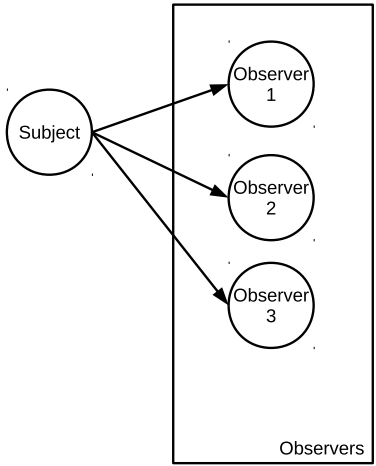
- This situation can benefit from use of the observer pattern.
  - This pattern is similar to subscribing to a hard copy newspaper
    - A newspaper comes into existence and starts publishing editions
    - You become interested in the newspaper and subscribe to it
    - Any time an edition becomes available, you are notified (by the fact that it is delivered to you)
    - When you don't want the paper anymore, you unsubscribe
    - The newspaper's current set of subscribers can change at any time
  - Observer is just like this but we call the publisher the "subject" and we refer to subscribers as "observers"

Observer in Action



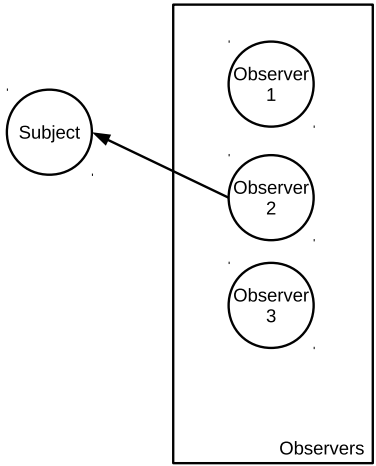
Subject maintains a list of observers

Observer in Action



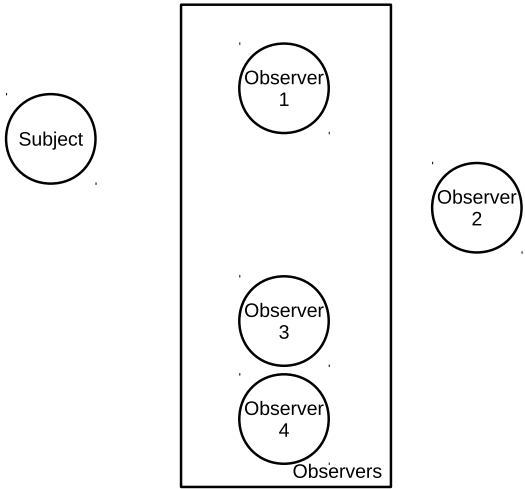
If the Subject changes, it notifies its observers

Observer in Action



If needed, an observer may query its subject for more information

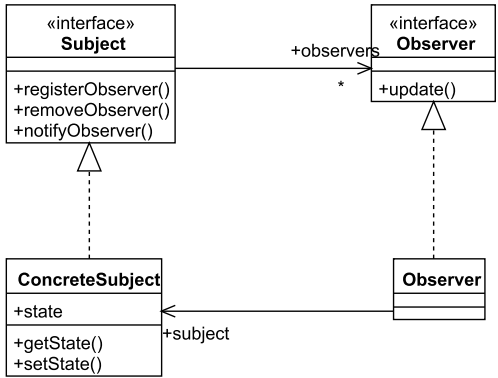
Observer in Action



At any point, an observer may join or leave the set of observers

Observer Definition and Structure

- The Observer Pattern defines a one-to-many dependency between a set of objects, such that when one object (the subject) changes all of its dependents (observers) are notified and updated automatically.



Observer Benefits

- Observer affords a loosely coupled interaction between subject and observer
  - This means they can interact with very little knowledge about each other.
- Consider
  - The subject only knows that observers implement the Observer interface
    - We can add/remove observers of any type at any time
    - We never have to modify subject to add a new type of observer
  - We can reuse subjects and observers in other contexts
    - The interfaces plug-and-play where ever observer is used
  - Observers may have to know about the ConcreteSubject class if it provides many different state-related methods
    - Otherwise, data can be passed to observers via the update() method

Are there any questions?

ROAR