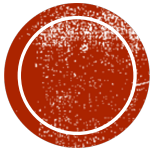


OBSERVER PATTERN

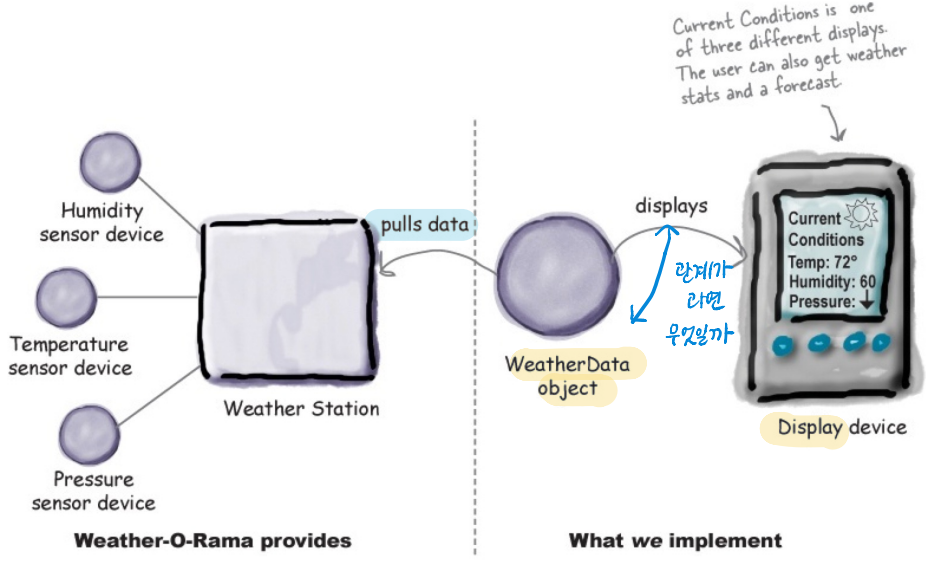
Sequence Diagram

√ 영상참조

a.k.a. Publish-Subscribe

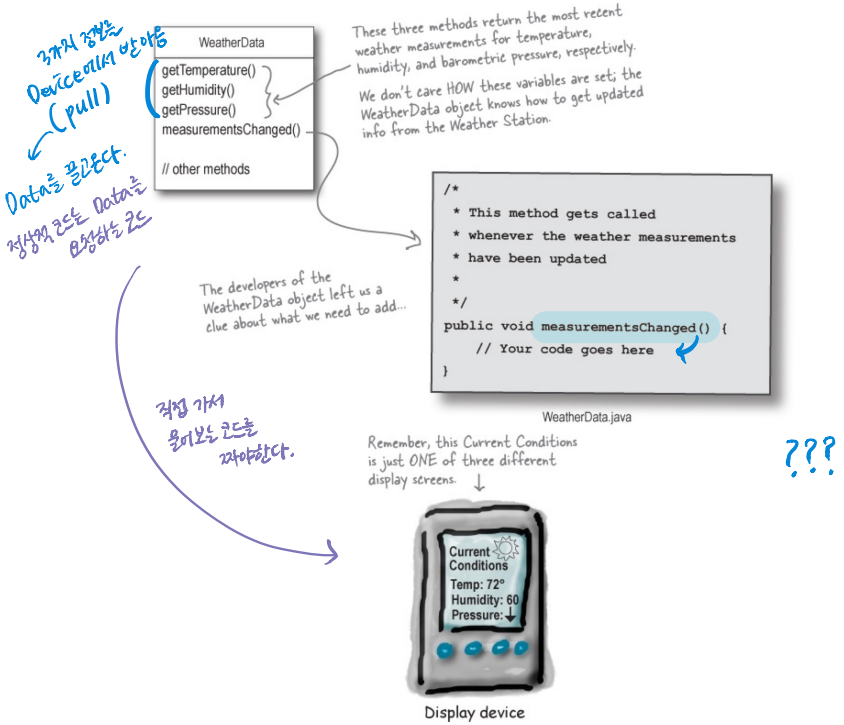
Dependents

WEATHER STATION API



SOURCE CODE FROM

WEATHER-O-RAMA



GIVEN SITUATION

- ☑ The WeatherData class has getter methods for three measurement values: temperature, humidity, and barometric pressure.

  ```
  getTemperature()
  getHumidity()
  getPressure()
  ```

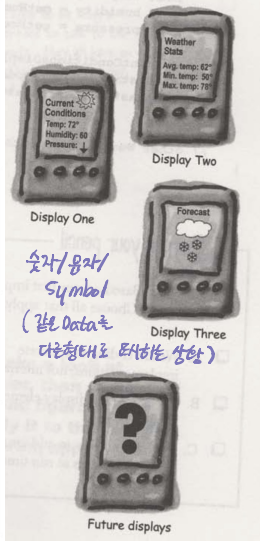
  ↔ pull method

- ☑ The measurementsChanged() method is called any time new weather measurement data is available. (We don't know or care how this method is called; we just know that it *is*.)

  ```
  measurementsChanged()
  ```

- ☑ We need to implement three display elements that use the weather data: a *current conditions* display, a *statistics display,* and a *forecast* display. These displays must be updated each time WeatherData has new measurements.

- ☑ The system must be expandable — other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial *three* display types (current conditions, statistics, and forecast).



* pull 하는 상황을 push로 바꾸면 참 좋겠다는 생각.

DIRECT APPROACH

```
public class WeatherData {

    // instance variable declarations

    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);
    }

    // other WeatherData methods here
}
```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

Data를 pull해서 각 객체에 보내주므로.

↔ Direct approach

DEPENDENCY INVERSION PRINCIPLE

PROGRAM TO AN INTERFACE NOT AN IMPLEMENTATION

```
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);
}
```

Area of change. We need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program. → 따로 처리해주는 코드를 작성해보자.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method that takes the temp, humidity, and pressure values.

OBSERVER PATTERN

신문사 ← 구독자들 계속 늘어남

아침마다 신문이 오는 과정

- Intention: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

  자동으로 수정됨을 알려주고 싶다.

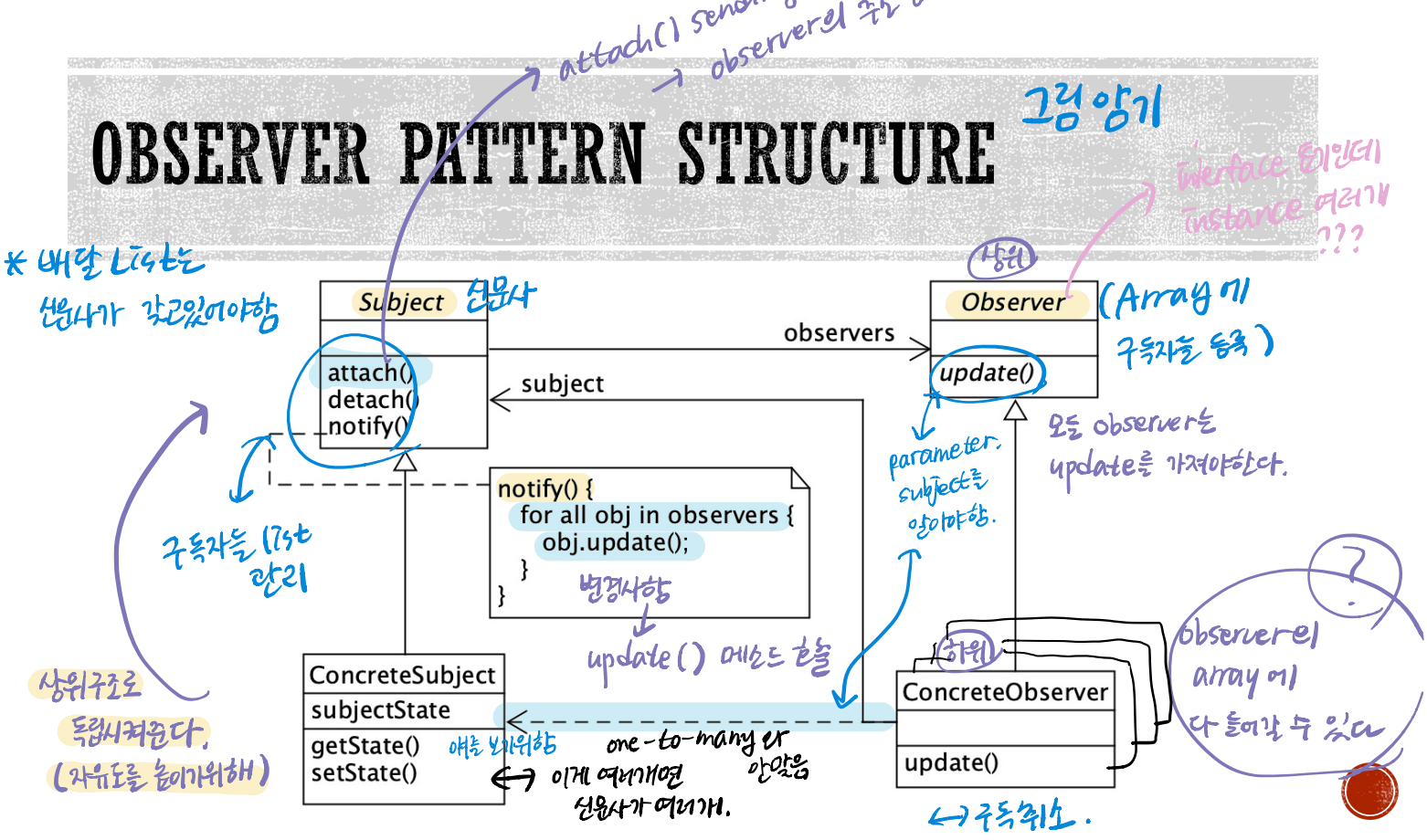
- Motivation: You need to notify a varying list of objects that an event has occurred
  - Observers delegate the responsibility for monitoring for an event to a central object (the Subject)

- Consequences: Subject may tell Observers about events they do not need to know if some Observers are interested in only subset of events. Extra communication may be required if subjects notify Observers which then go back and request additional information.

  ?? 바로 적용이 안되는것인가 싶은

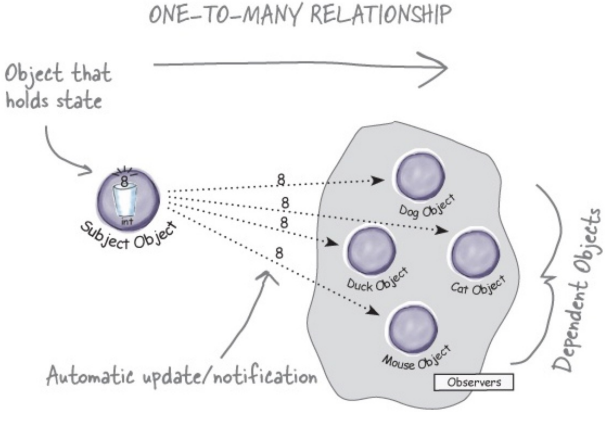
OBSERVER PATTERN STRUCTURE

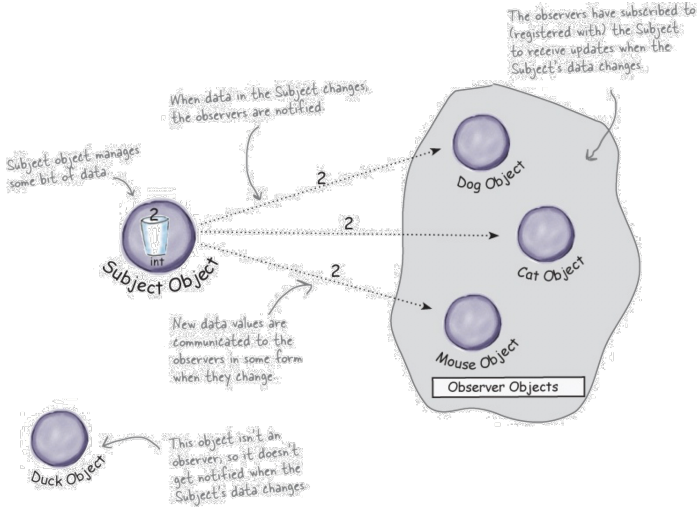
그림 암기

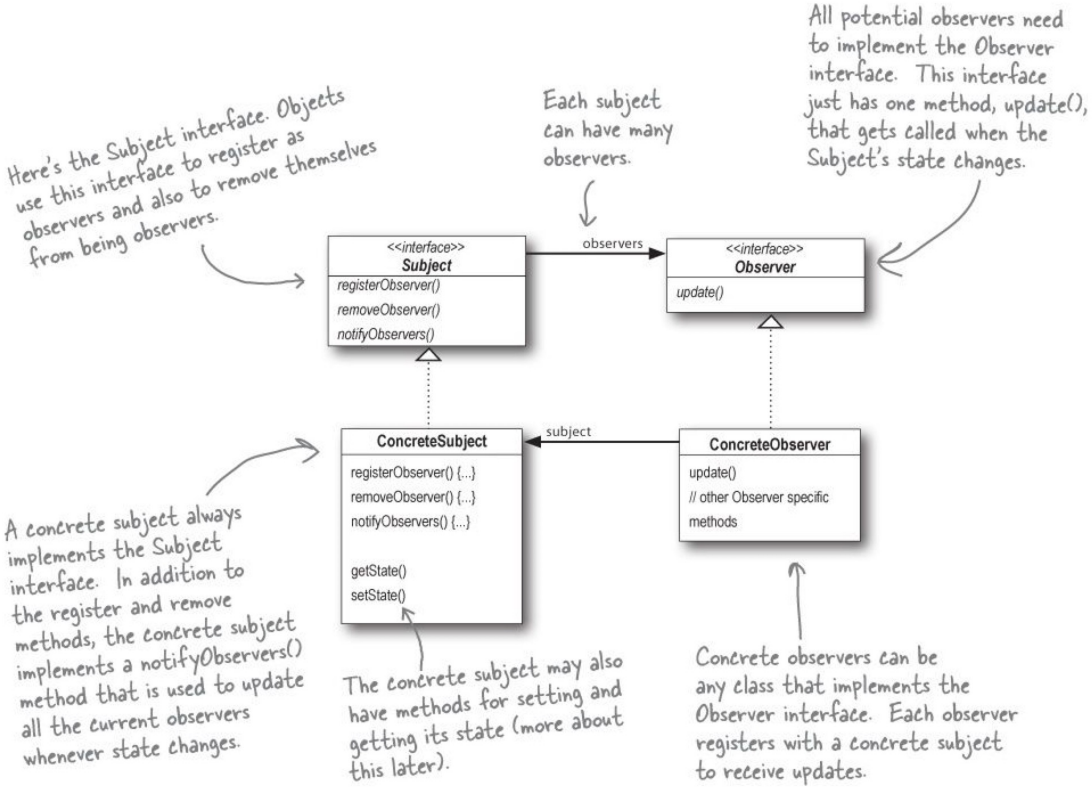


* 느슨하게 연결된 설계를 class들을 결합해보다 ↔ Dependency Inversion principle

APPLYING OBSERVER...






Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.
Each subject can have many observers.
All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific methods
A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.
The concrete subject may also have methods for setting and getting its state (more about this later).
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

HIGH COHESION & LOOSELY COUPLED

- Observer Pattern is loosely Coupled
- Only thing the subject knows about an observer is observer's interfaces, No information about implementation is needed
- Never need to modify the subject to add new types of observers, independency between the subject and the observers
- Changes in one components does not affect the other side

OBSERVER IN JAVA

*thread