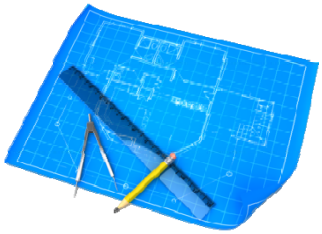


Observer 패턴

NOTE 04

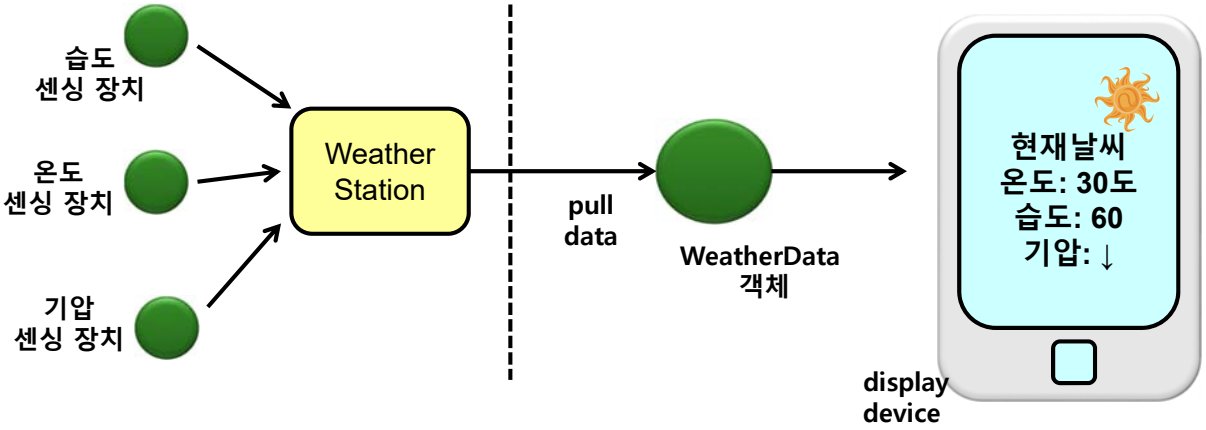
한국기술교육대학교 컴퓨터공학부 김상진

sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

교육목표

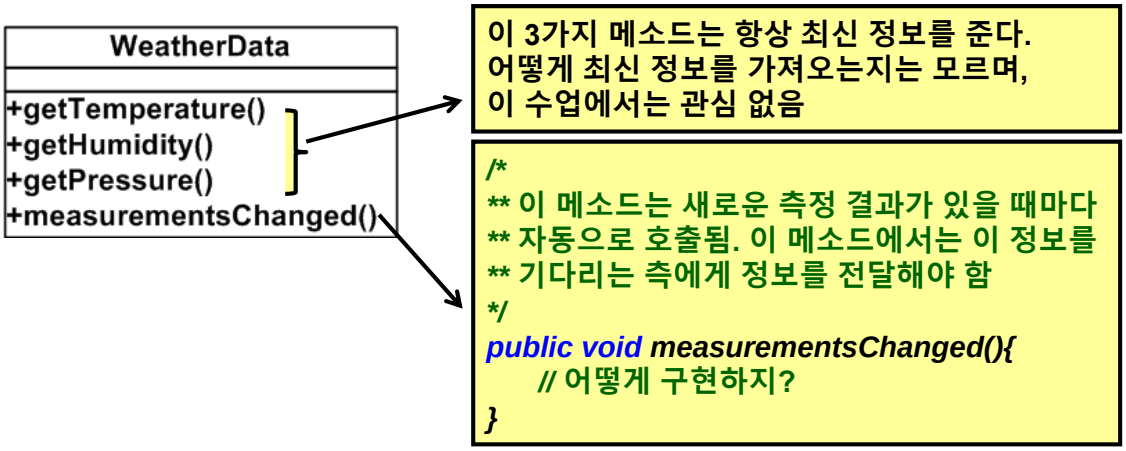
- Observer Pattern(관찰자 패턴) 익히기
  - 객체가 관심 있어 하는 사건의 발생을 알려주어야 할 때
    - 관찰하는 객체가 능동적으로 관찰하는 것이 아니라 관찰 대상으로부터 어떤 사건이 발생하였을 때 수동적으로 통보해주길 기다림
  - 예) 수 많은 GUI
    - JDK Swing/JavaFX 라이브러리에서 가장 많이 사용되는 패턴 중 하나

Weather Monitoring App. (1/2)



- 이 응용은 현재 날씨, 날씨통계정보, 일기예보 세 종류의 출력장치를 지원해야 하며, 확장 가능하도록 API를 제공해야 함

Weather Monitoring App. (1/2)



- 우리의 목표: measurementsChanged 메소드의 구현
  - 3 종류의 출력장치를 지원해야 하며, 확장 가능(새로운 형태의 출력 장치에 대해서도 지원이 가능해야 함)해야 함

시도 1.

```
public void measurementsChanged(){
    float temperature = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();
    currentConditionsDisplay.update(temperature, humidity, pressure);
    statisticDisplay.update(temperature, humidity, pressure);
    forecastDisplay.update(temperature, humidity, pressure);
}
```

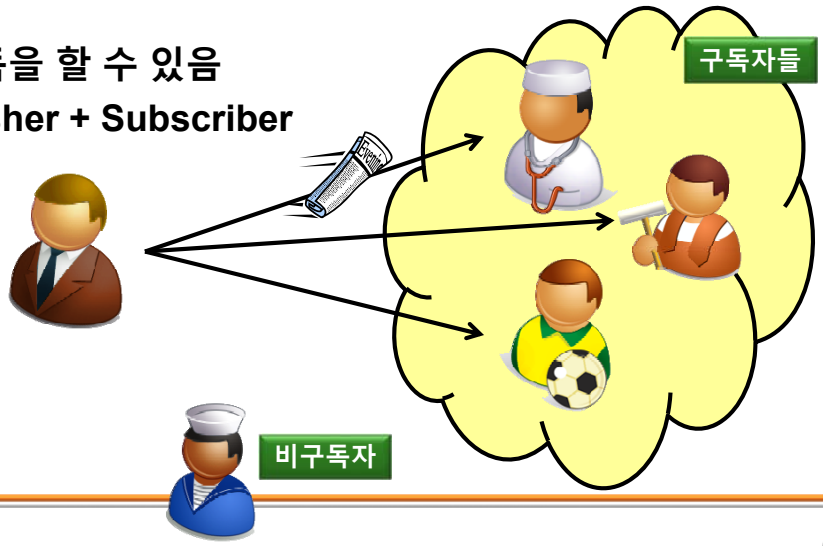
변할 수 있는 부분

Based on our first implementation, which of the following apply?

A. *We are coding to concrete implementations not interface*
B. *For every new display element we need to alter code*
C. *We have no way to add(or remove) display elements at run time*
D. *The display elements don't implement a common interface*
E. *We haven't encapsulated the part that changes*
F. *We are violating encapsulation of the WeatherData class*

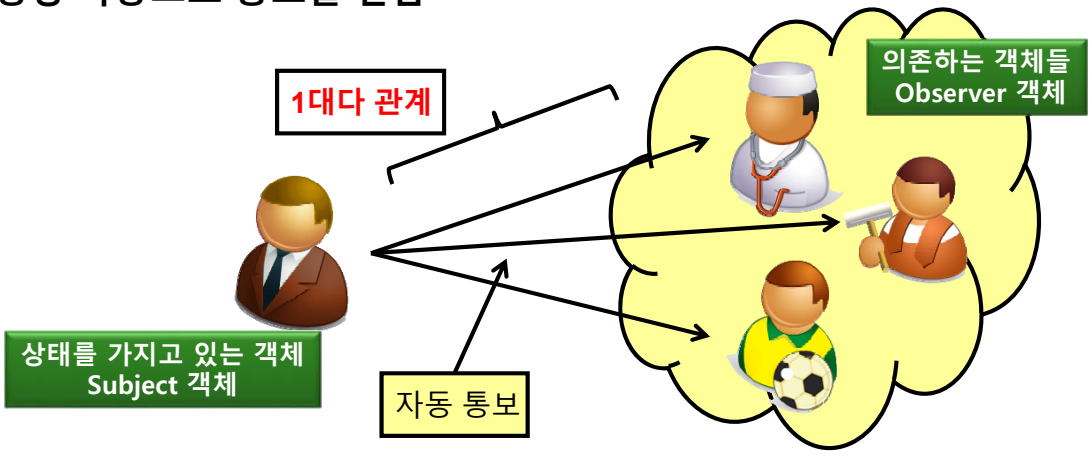
Observer Pattern (1/3)

- Observer 패턴은 다음과 같은 시나리오에서
  - 신문사는 신문을 발행함
  - 고객은 새롭게 발행될 때마다 신문을 받아보기 위해서는 구독을 해야 함
  - 언제든지 구독을 중단할 수 있으며, 중단하면 더 이상 신문이 배달되지 않음
  - 누구나 신문 구독을 할 수 있음
- Observer = Publisher + Subscriber

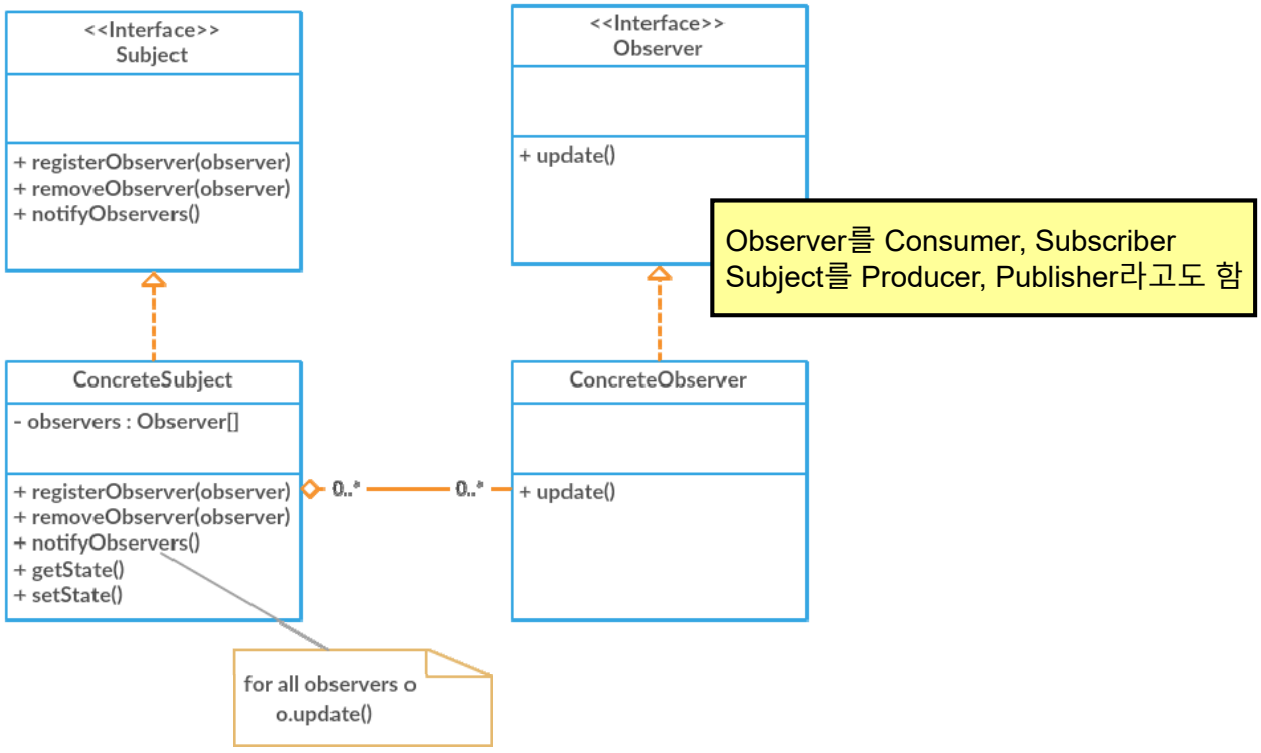


Observer Pattern (2/3)

- **Observer Pattern: 한 객체와 다중 객체간에 의존관계를 정의하며, 한 객체에 의존하는 모든 객체들은 의존하는 객체의 상태가 변하면 항상 자동으로 통보를 받음**



Observer Pattern (3/3)

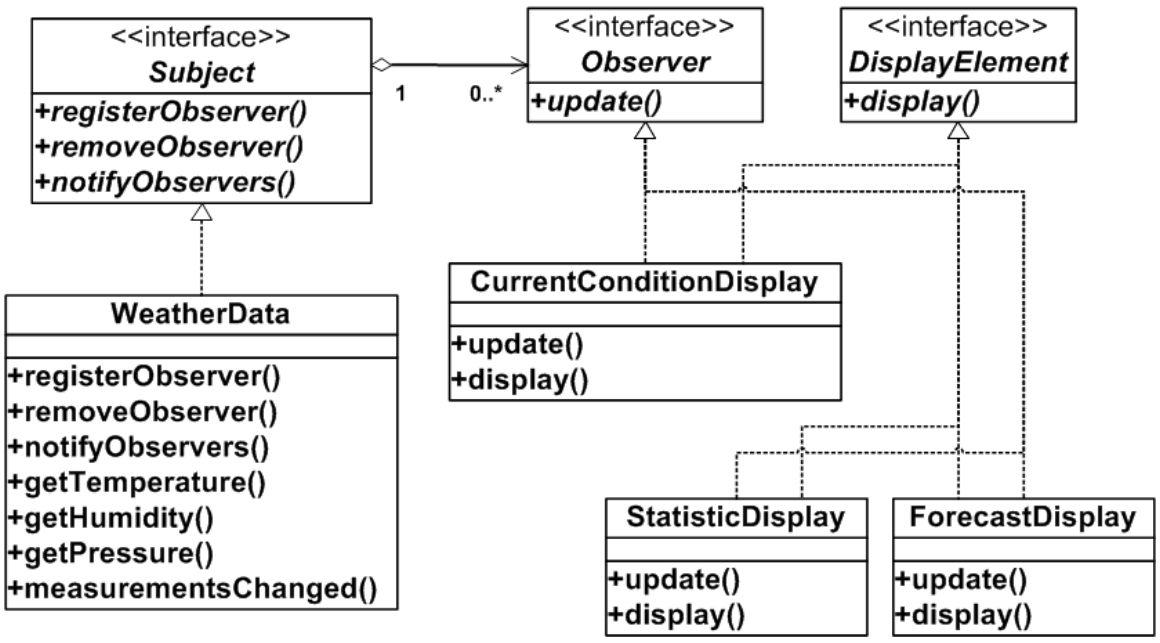


Subject과 Observer 간에 관계

- Subject은 Observer에 대해 알고 있는 유일한 정보는 Observer interface를 구현하고 있다는 것임 (update 메소드를 가지고 있음)
- 새 Observer를 쉽게 추가할 수 있으며, 기존 Observer를 쉽게 제거할 수 있음
- 새로운 종류의 Observer를 추가하기 위해 Subject를 변경할 필요가 없음
- Observer나 Subject의 수정은 서로에게 영향을 주지 않음.
  단, interface는 충실히 구현할 경우

Design Principle 4
Strive for loosely coupled designs between objects that interact

Weather Monitoring App.의 구현



Weather Monitoring App.의 구현

```
public interface Subject{
        void registerObserver(Observer o);
        void removeObserver(Observer o);
        void notifyObservers();
} // interface Subject
```

```
public interface Observer{
        void update(float temperature, float humidity, float pressure);
} // interface Observer
```

```
public interface Display{
        void display();
} // interface Display
```

이렇게 하는 것이 과연 바람직한가?
향후 변경될 소지는 없나?

Weather Monitoring App.의 구현

```
public class WeatherData implements Subject{
        private ArrayList<Observer> observers = new ArrayList();
        private float temperature;
        private float humidity;
        private float pressure;
        public void registerObserver(Observer o){
                observers.add(o);
        }
        public void removeObserver(Observer o){
                int i = observers.indexOf(o);
                if(i>=0) observers.remove(i);
        }
        public void notifyObservers(){
                for(Observer o: observers){
                        o.update(temperature, humidity, pressure);
                }
        }
        public void measurementChanged(){
                notifyObservers();
        }
        …
}
```

중복문제?

Weather Monitoring App.의 구현

```
public class CurrentConditionDisplay implements Observer, DisplayElement {
        private float temperature;
        private float humidity;
        private float pressure;
        @Override
        public void display() {
                System.out.printf("현재온도: %.2f%n", temperature);
                System.out.printf("현재습도: %.2f%n", humidity);
                System.out.printf("현재기압: %.2f%n", pressure);
        }
        @Override
        public void update(float t, float h, float p) {
                temperature = t;
                humidity = h;
                pressure = p;
                display();
        }
}
```

멤버로 Subject의 추가는?
나중에 구독을 취소하고 싶으면

Weather Monitoring App.의 구현

```
public class WeatherORamaTest {
        public static void main(String[] args) {
                WeatherData weatherData = new WeatherData();
                CurrentConditionDisplay currentDisplay = new CurrentConditionDisplay();
                weatherData.registerObserver(currentDisplay);
                weatherData.setMeasurement(30, 65, 30.4f);
                weatherData.setMeasurement(28, 55, 29.2f);
        } // main()
} // class WeatherORamaTest
```

Why push instead of pull?

- Push는 Subject가 통보할 때 Observer에게 정보까지 제공하는 것이고, pull은 Subject가 Observer에게 변경사실이 있다는 것만 통보하고 정보는 Observer가 별도로 요구하여 받음
- Observer 측면에서 push 방식이 더 편리함
  - Pull을 해야 하는 경우 subject에 대한 더 많은 정보를 알아야 하며, 호출해야 하는 메소드도 많을 수 있음 (보다 더 tight-coupling됨)
  - 반론. Observer에 따라 필요로 하는 정보가 다를 수 있어 불필요한 정보를 전달받을 수도 있음
  - 극단적인 경우 다중 쓰레드 환경에서 pull 방식을 사용하면 통보와 데이터 질의 사이에 상태 변화가 발생할 수도 있음
  - Subject 측면에서 pull 방법이 편리하고, Object 측면에서는 push 방법이 편리함

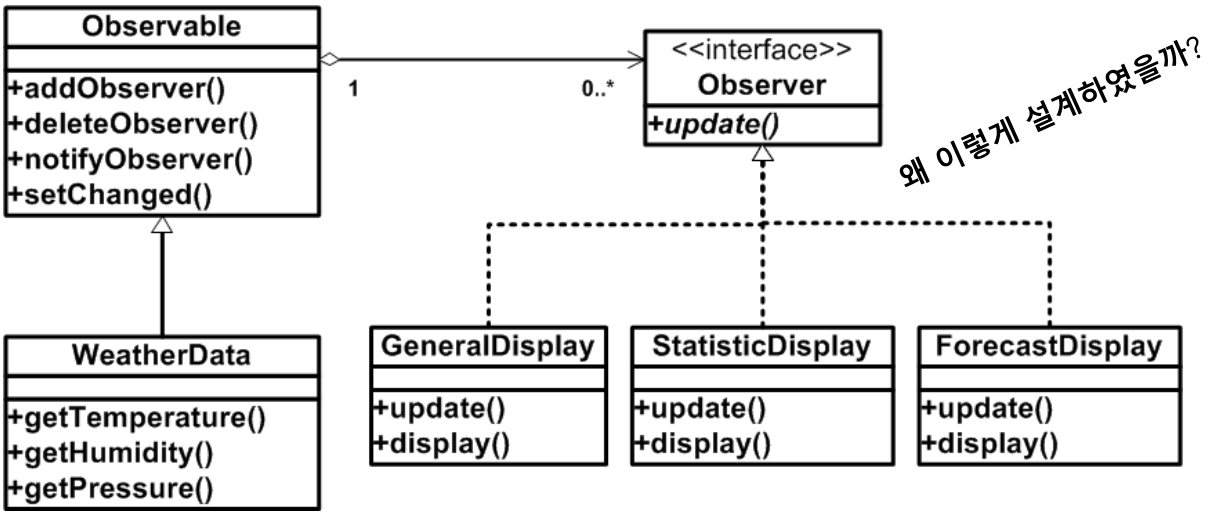
```
@Override
public void update(WeatherData w) {
        temperature = w.getTemperature();
        display();
}
```


한 Observer가 다중 Subject를 관찰할 때

- Observer 측면에서는 어떤 Subject가 통보하였는지 식별할 수 있어야 함
  - 각 Subject마다 호출하는 update 메소드가 달라야 함
    - 가장 쉬운 방법은 Subject 자체(Subject들이 모두 다른 타입일 경우)를 전달하고 pull 방법으로 데이터를 취하는 것임
    - 현재 자바 라이브러리 GUI처럼 source를 얻을 수 있도록 할 수 있음. 이 경우 observer가 관찰하는 모든 Subject의 공통 타입을 정의하여 사용할 수도 있음

Java에서 제공하는 Observer 패턴 (1/5)

- java.util 패키지에 제공되는 Observer 인터페이스와 Observable 클래스를 이용하여 observer 패턴을 활용할 수 있음
  - 주의. Observer는 클래스가 아니고 인터페이스이며, Observable은 인터페이스가 아니고 클래스임. (이름이 이상해?)



Java에서 제공하는 Observer 패턴 (2/5)

- java.util를 활용하여 객체를 subject로 만들기 위해서는 Observable를 상속해야 함
  - Observer들에게 변경을 알려주기 위해서는
    - 단계 1. setChanged() 호출 (이것의 용도?)
    - 단계 2. notifyObservers() 또는 notifyObservers(Object arg)를 호출함
      - Push, Pull 방법 모두 구현가능
        - Push: arg를 통해 원하는 데이터를 push함
        - Pull: Observable로부터 원하는 데이터를 observer가 직접 추출해야 함
- java.util를 활용하여 객체를 observer로 만들기 위해서는 Observer 인터페이스를 구현하면 됨
  - update(Observable o, Object arg)
    - notifyObservers()를 호출하면 arg는 null값을 가짐

Java에서 제공하는 Observer 패턴 (3/5)

```
setChanged(){
        changed = true;
}
```

```
notifyObservers(Object arg){
        if(changed){
                for every observers on the list{
                        call update(this, arg)
                }
                changed = false;
        }
}
```

```
notifyObservers(){
        notifyObservers(null);
}
```

- setChanged가 필요한 이유
  - 유연성 제공
  - 자주 통보되는 것을 조절할 수 있음

Java에서 제공하는 Observer 패턴 (4/5)

- java.util 패키지에서 제공하는 Observer 패턴의 문제점
  - Observable은 인터페이스가 아니라 클래스임
    - 문제점. 자바는 다중상속을 제공하지 않으므로 다른 클래스를 상속받아야 하는 클래스는 subject가 될 수 없음
  - Observable의 setChanged 메소드는 protected 메소드임
    - 상속받지 않고는 Observable 클래스의 인스턴스를 생성할 수 없음 (즉, Has-a 관계로 활용불가)
    - 설계원리 3(favor Has-a over Is-a)에 위배됨
      - 상속해야 하는 것을 has-a로 바꿀 수 있음
    - 돌아가는 방법

Java에서 제공하는 Observer 패턴 (5/5)

- java.util 패키지에서 제공하는 Observer 패턴의 문제 극복 방안

```
public class MySubject implements Subject {
   private … data;
   private final ImprovedObservable observable =
      new ImprovedObservable();
   public void addObserver(Observer o) {
      observable.addObserver(o);
   }
   public void notifyObservers() {
      observable.notifyObservers(this);          pull
      // observable.notifyObservers(data);       push
   }
   …
}
```

```
class ImprovedObservable
      extends Observable {
   @Override
   public void setChanged() {
      super.setChanged();
   }
}
```

```
public class … implements Observer{
      public void update(Observable o, Object arg) {
            MySubject s = (MySubject)arg;
            // … d = (…)arg;
            …
      }
}
```

MySubject는 Observable 타입이 아님

Observer 패턴의 예

- JButton과 같은 GUI 객체는 addActionListener 메소드를 가지고 있음
  - 즉, JButton은 많은 observer를 등록할 수 있는 subject임
  - ActionListener 인터페이스는 actionPerformed라고 하는 event가 발생할 때 자동으로 호출되는 메소드를 가지고 있음
    - 이 메소드는 observer 패턴의 update 메소드에 해당됨
- 가장 중요한 패턴 중 하나인 MVC(Model-View-Controller) 패턴에서 모델과 뷰는 Observer 패턴을 많이 활용함. 모델, 즉 데이터가 변화면 그것의 뷰는 자동 갱신되어야 함

Observer 패턴 (1/3)

- 패턴의 특성: Behavioral
- 수준: Component
- Applicability
  - 최소 하나 이상의 subject가 존재할 때
  - 한 객체의 상태 변화가 얼마나 많은 다른 객체에 영향을 주어야 하는지 알 수 없을 때
  - Observer들이 하나의 응용 또는 여러 응용에 따라 다양해질 수 있을 때
- 장점
  - Subject에 대한 수정 없이 Observer를 추가할 수 있으며, 동적으로 추가/제거가 가능
    - 반면, Observer는 상대적으로 Subject에 대한 의존도가 있음
  - Subject는 Observer가 update 메소드를 가지고 있다는 것 외에는 알 필요가 없음

Observer 패턴 (2/3)

- Principal Challenge
  - Subject가 Observer에게 전달하는 정보의 수준
    - 범용적이면 불필요한 정보의 전달이 존재할 수 있고, 반대로 구체적일 수록 Subject와 Observer의 구현이 모두 복잡해질 수 있음
- 구현 시 고려사항
  - 한 Observer가 여러 개의 Subject를 구독하고 있을 때: observer가 어떤 subject가 update를 호출하였는지 알아야 할 필요가 있음
  - Observer가 관심 가지고 있는 상태 변화를 구독할 때 알리는 경우도 있음
- 주의사항
  - Subject가 Observer에게 통보하고, 이 Observer가 Subject가 되어 제차 통보하고 이것이 반복되는 연쇄적 통보(cascading notification) 모델은 다르게 모델링하는 것이 적절함

Observer 패턴 (3/3)

- 패턴의 변형
  - Push가 아니라 pull 방식
  - 다중쓰레드 subject인 경우 사건발생 큐를 지원가능
- 관련 리펙토링
  - Replace Hard-Coded Notifications with Observers
    - 1:1 관계가 1:다로 바뀌면…
- 관련 패턴
  - Mediator 패턴. Subject와 Observer 사이에 위치하여 중계역할을 하는 객체를 둘 수 있으며, 이 객체는 통보되는 빈도 및 통보하기 전에 필요한 사전처리를 할 수 있음


```
template <class T> class Subject{
private: vector<Observer<T> *> observers;
public:
    void registerObserver(Observer<T>* o){
       observers.push_back(o);
    }
    void removeObserver(Observer<T>* o){
       for(auto pIter = begin(observers);
             pIter!=end(observers);pIter++)
             if(*pIter==o){
                   observers.erase(pIter); break;
             }
    }
    void notifyObservers(string data){
       for(auto o: observers)
             o->update(data);
    }
};
class SoccerServer: public Subject<string>{
private: string currentScore;
public:
    void updateScore(string score){
       currentScore = score;
       notifyObservers(currentScore);
    }
};
```

```
template <class T> class Observer{
    public:
       virtual void update(T data) = 0;
       virtual ~Observer() {}
};

class SoccerObserver
    : public Observer<string>{
public:
    void update(string result){
       cout << "current score: "
             << result << endl;
    }
};

int main() {
  SoccerServer server;
  SoccerObserver client;
  server.registerObserver(&client);
  server.updateScore("Korea 1: Brazil 0");
  server.removeObserver(&sObserver);
  server.updateScore("Korea 2: Brazil 1");
}
```

Python

```
from abc import ABCMeta,
    abstractmethod

class Subject(object):
    __metaclass__ = ABCMeta
    def __init__(self):
        self.observers = set()
    def registerObserver(self, observer):
        self.observers.add(observer)
    def removeObserver(self, observer):
        self.observers.remove(observer)
    @abstractmethod
    def notifyObservers(self):
        pass

class Observer(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def update(self, data):
        pass
```

```
class SoccerServer(Subject):
    def __init__(self):
        Subject.__init__(self)
        self.score = ""
    def updateScore(self, score):
        self.currentScore = score
        self.notifyObservers()
    def notifyObservers(self):
        for observer in self.observers:
            observer.update(self.score)

class SoccerObserver(Observer):
    def __init__(self):
        Observer.__init__(self)
    def update(self, score):
        print "현재 점수: "+score

server = SoccerServer()
client = SoccerObserver()
server.registerObserver(client)
server.updateScore("Korea 1: Brazil 0")
server.removeObserver(client)
server.updateScore("Korea 2: Brazil 1")
```


Javascript

```
var inherits = function(child, parent) {
    child.prototype
        = Object.create(parent.prototype);
    child.prototype.constructor = child;
};

var Subject = function(){
    this.observers = new Set();
};
Subject.prototype.registerObserver =
    function(observer){
        this.observers.add(observer);
    };
Subject.prototype.removeObserver =
    function(observer){
        this.observers.delete(observer);
 };
Subject.prototype.notifyObservers =
    function(){};

var Observer = function(){};
Observer.prototype.update =
    function(data){
};
```

```
var SoccerServer = function(){
    Subject.call(this);
    this.score = "";
};
inherits(SoccerServer, Subject);
SoccerServer.prototype.updateScore =
    function(score){
        this.score = score;
        this.notifyObservers();
};
SoccerServer.prototype.notifyObservers
    = function(){
        for(observer of this.observers)
            observer.update(this.score);
};

var SoccerObserver = function(){
    Observer.call(this);
    this.score = "";
};
inherits(SoccerObserver, Observer);
SoccerObserver.prototype.update =
    function(score){
        this.score = score;
        console.log("현재 점수: "+this.score);
};
```