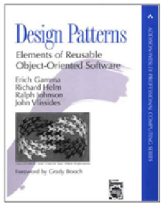


Design Pattern Definitions from the GoF Book

The Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Structural Patterns

- **The Decorator Pattern**
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

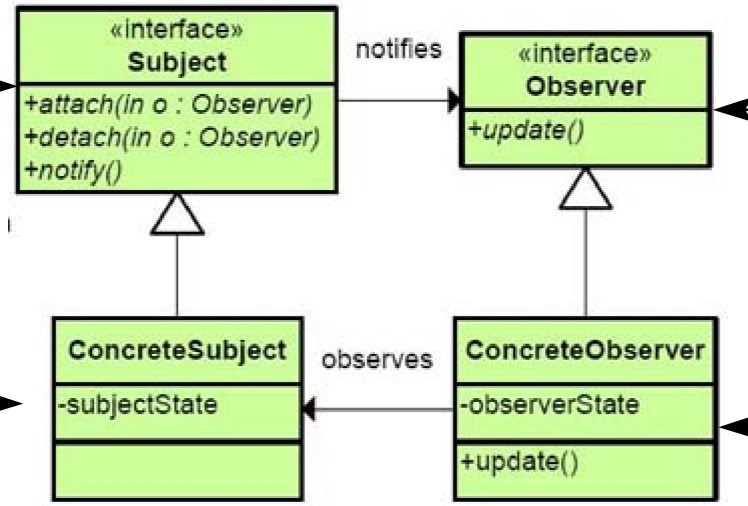
Creational Patterns

- **The Factory Method Pattern**
- **The Abstract Factory Pattern**
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Design Patterns: Observer
Quick Overview

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Defines the interface the Observers use to access the data provided by a Subject.



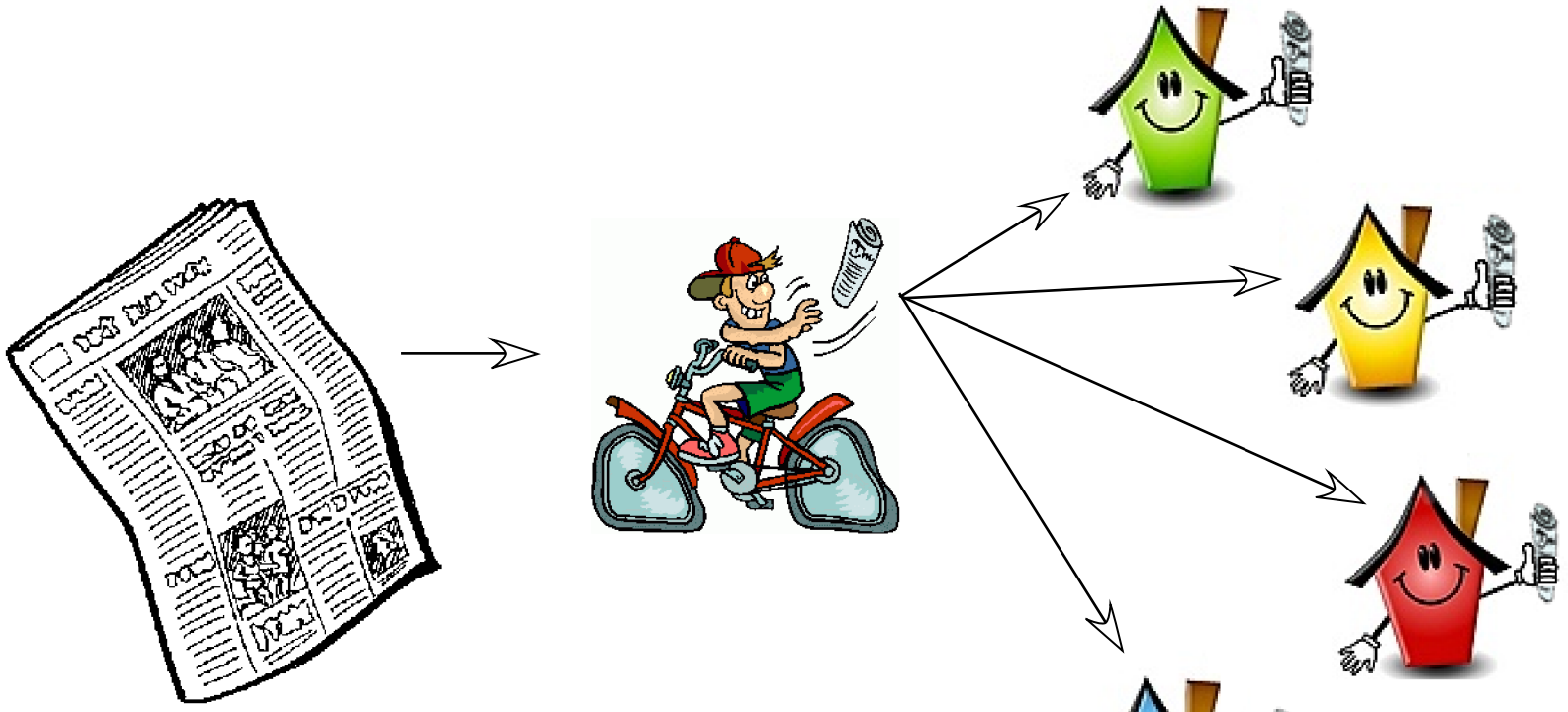
Defines the interface for an Observer that a Subject uses to let it know when data changes.

Actual provider of the data when it's "state" changes.

Defines the actual Observer object who wants to know when the data changes.

Design Patterns: Observer

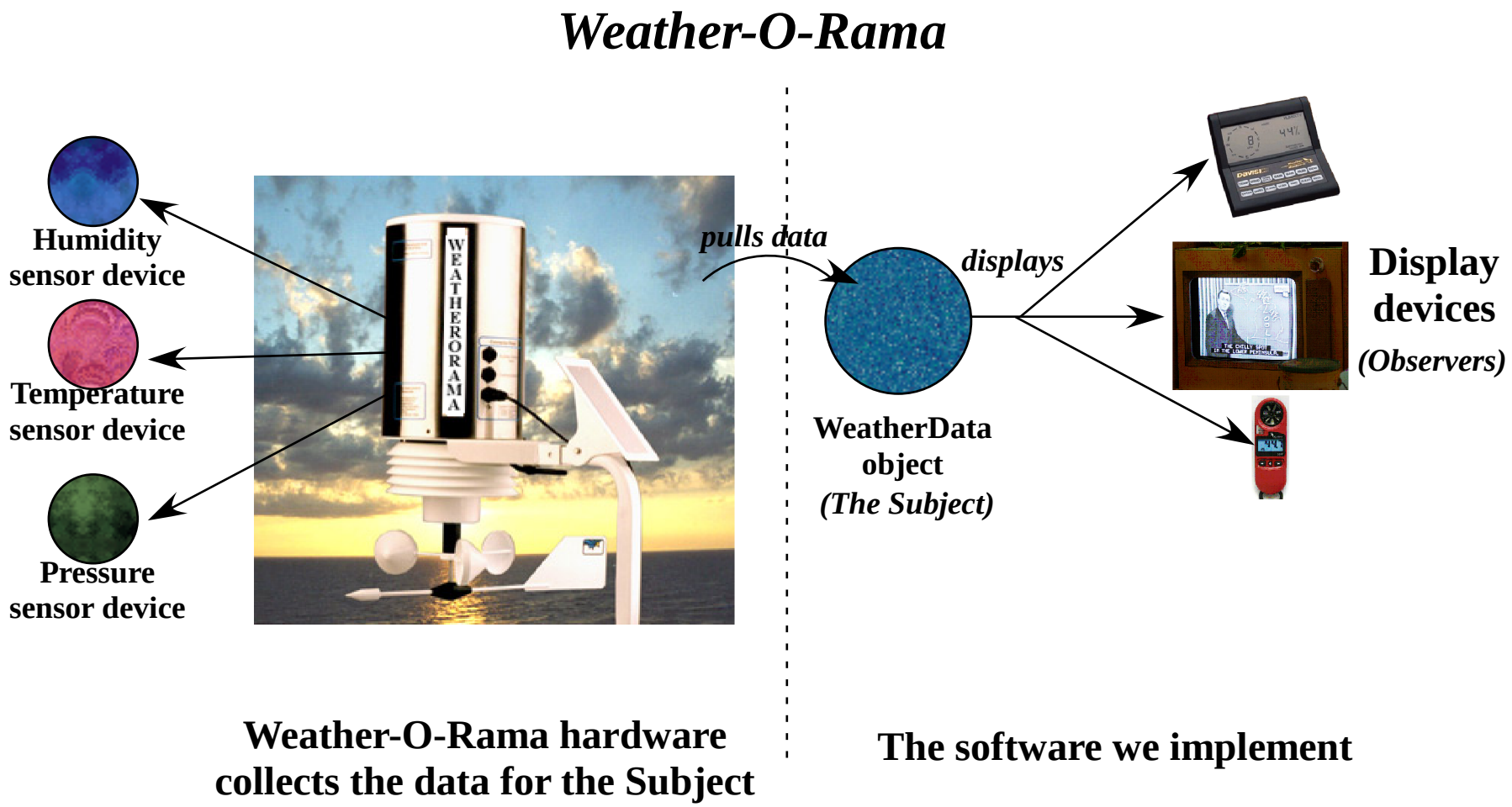
Publisher-Subscriber Relationship



The Subject

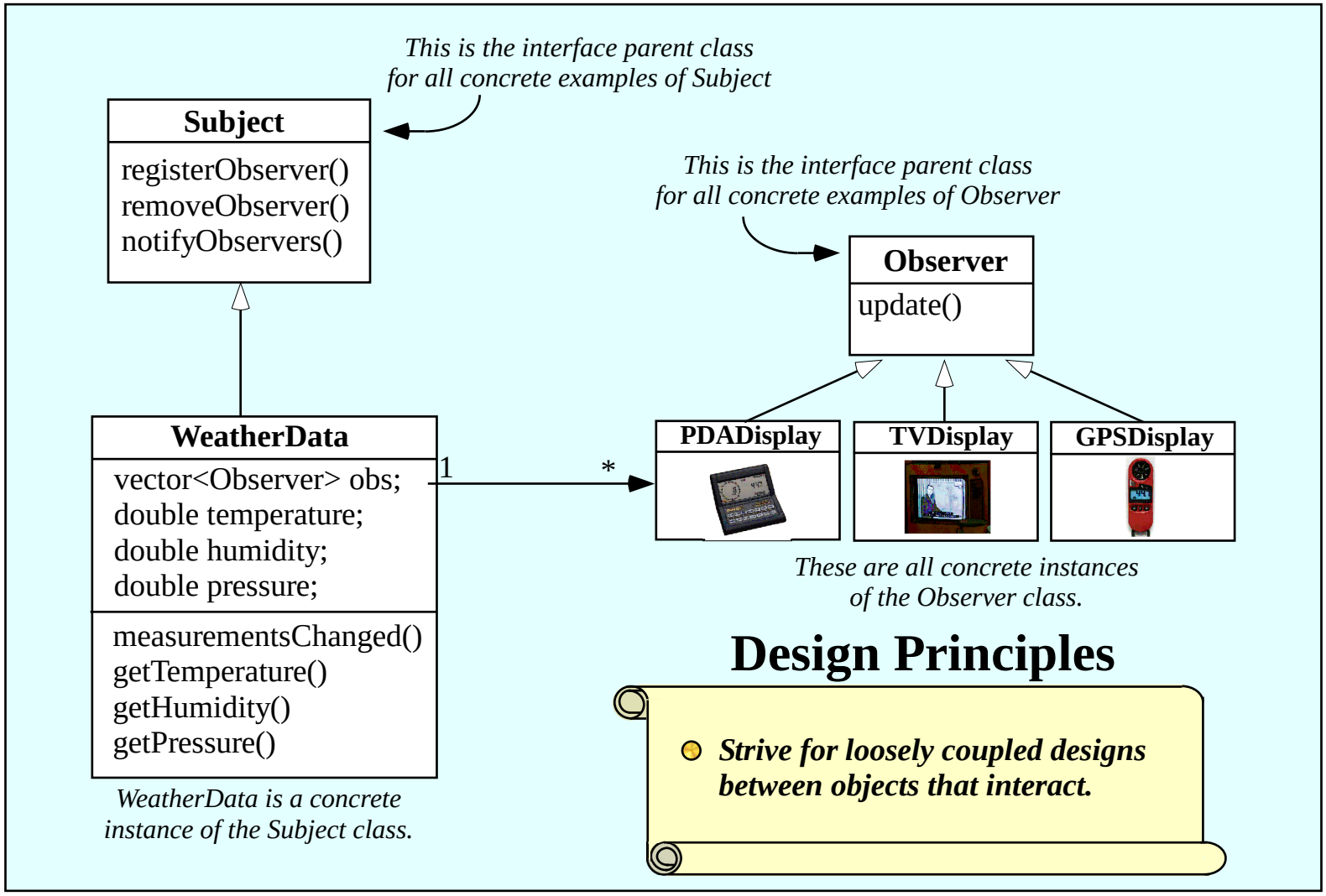
The Observers

Design Patterns: Observer



Design Patterns: Observer

Class Diagram



Design Principles

- ***Strive for loosely coupled designs between objects that interact.***

Design Patterns: Observer

A Burning Question

How do the observers actually get the data?

1. Subject PUSHES data to observers

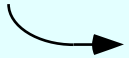
When WeatherData gets new data from the weather station it calls notifyObservers inherited from the Subject class

WeatherData
vector<Observer> obs; double temperature; double humidity; double pressure;
notifyObservers()

```
void WeatherData::notifyObservers()
{
   for(vector<Observer>::iterator itr=obs.begin(); itr!=obs.end(); itr++)
   {
      itr->update(temperature, humidity, pressure);
   }
}
```

2. Observers PULL data from the subject

WeatherData calls notifyObservers as above



WeatherData
vector<Observer> obs; double temperature; double humidity; double pressure;

// other get functions here
notifyObservers()
getTemperatureReading()
getHumidityReading()
getPressureReading()

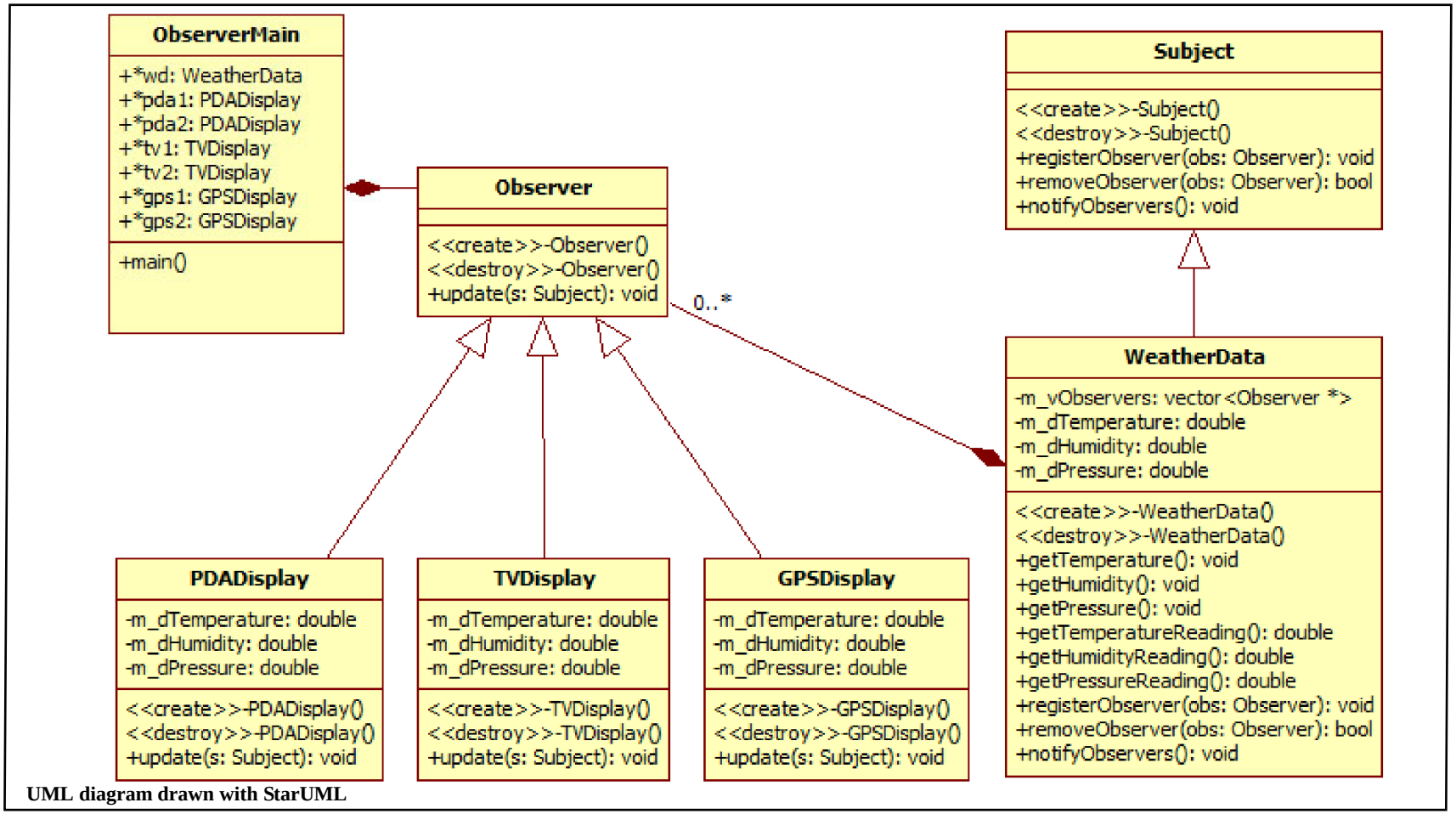
```
void WeatherData::notifyObservers()
{
   for(vector<Observer>::iterator itr=obs.begin(); itr!=obs.end(); itr++)
   {
      itr->update(this);
   }
}
```

PDADisplay
void GPSDisplay::update(Subject *s) **{** ** this->temp = s->getTemperatureReading()** **}**

Observers can now pull what data they need

Design Patterns: Observer

Code Sample



UML diagram drawn with StarUML

ObserverMain

Instantiates Subject as WeatherData

Instantiates and registers Observers as:

PDADisplay, TVDisplay, GPSDisplay

At one second intervals:

Calls WeatherData->notifyObservers

Weather Data calls Observer->update(this)

Randomly subscribe/unsubscribe observers

Let's look at the code and run the demonstration.