Observer Pattern

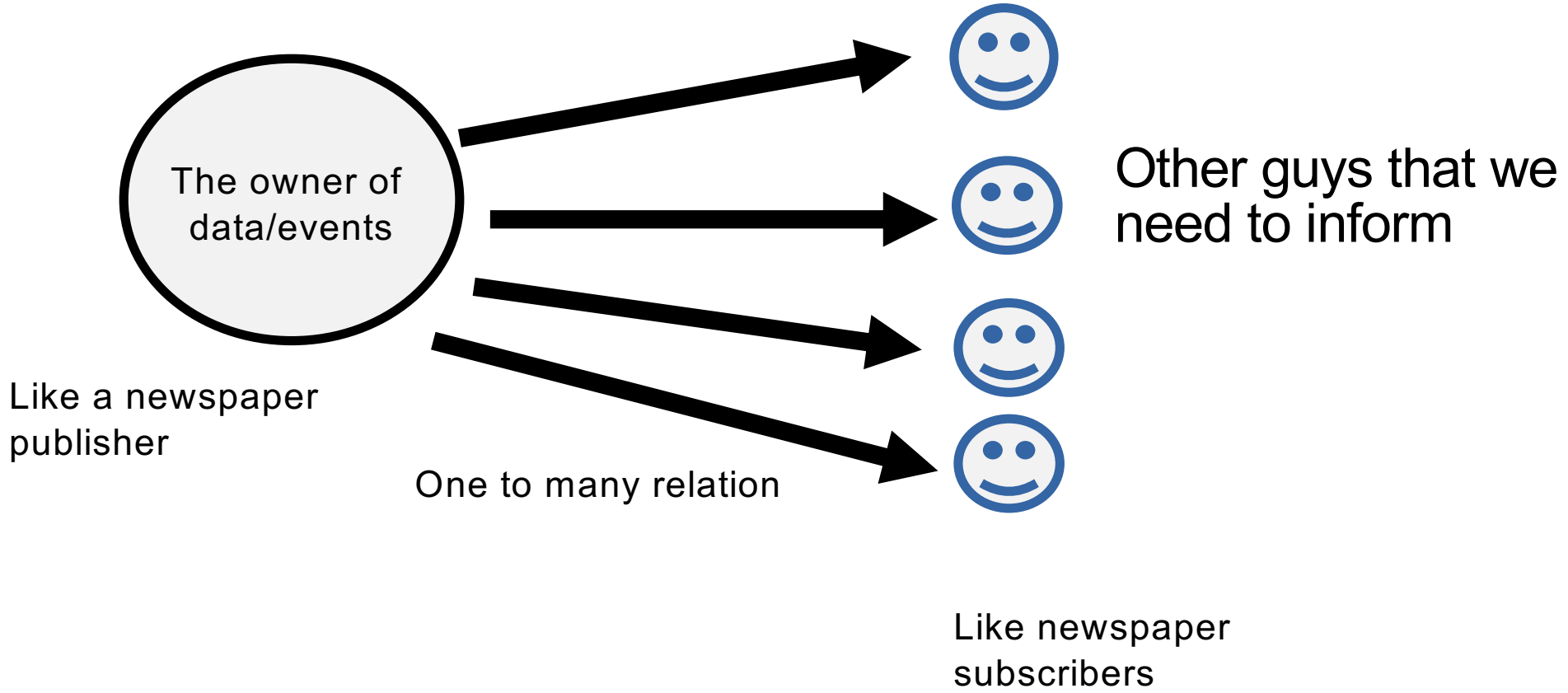
Use Case Problem

- Sometimes in our application a component has data about
  - things that are happening frequently like events
  - data items that will be repeatedly generated like sensor data
  - object states that are changing frequently

And we need to inform multiple parties.

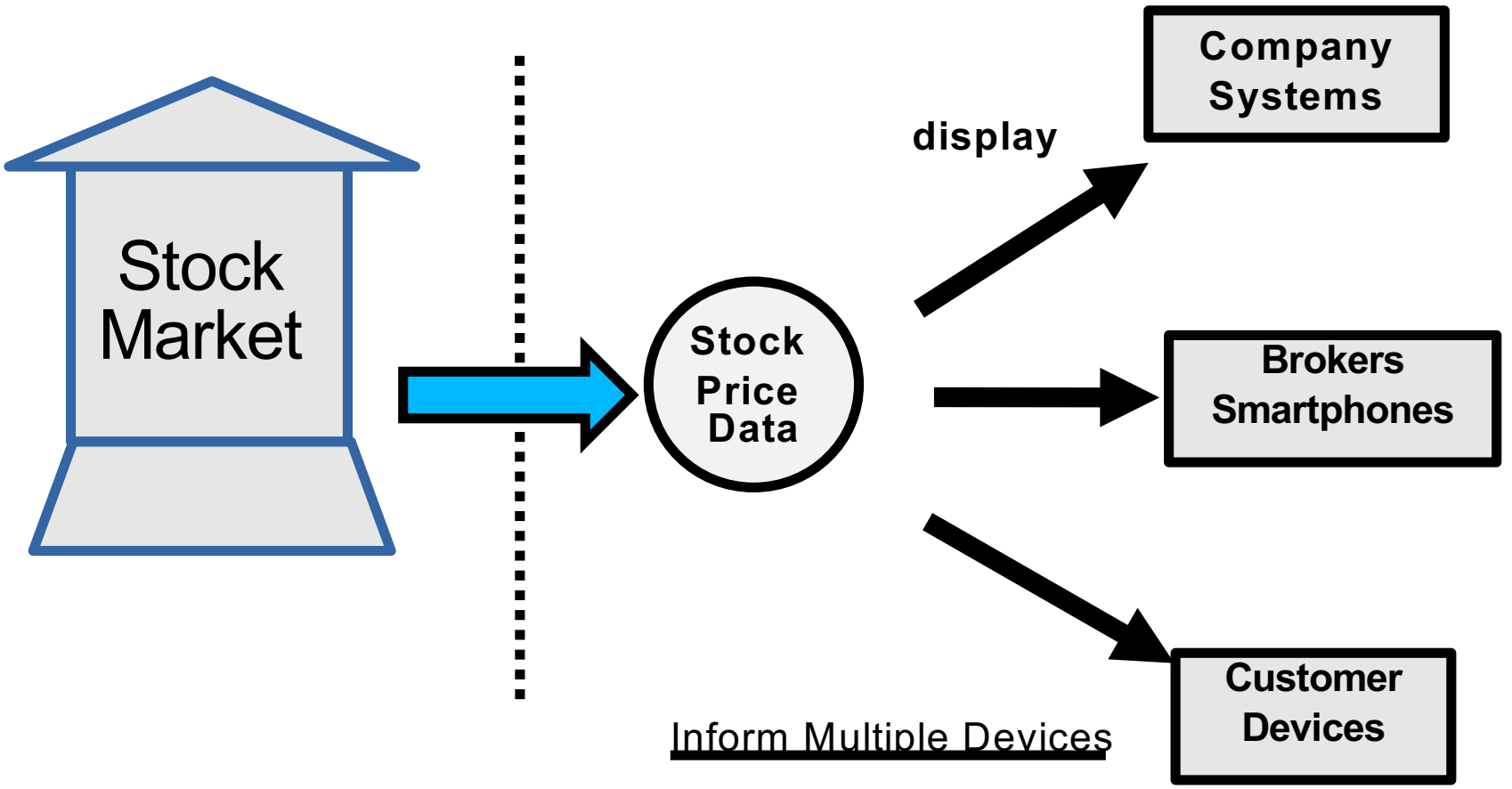
Use Case Problem

- We need to inform multiple other objects/components about these updates frequently and inform them new changes and updates as fast as possible.



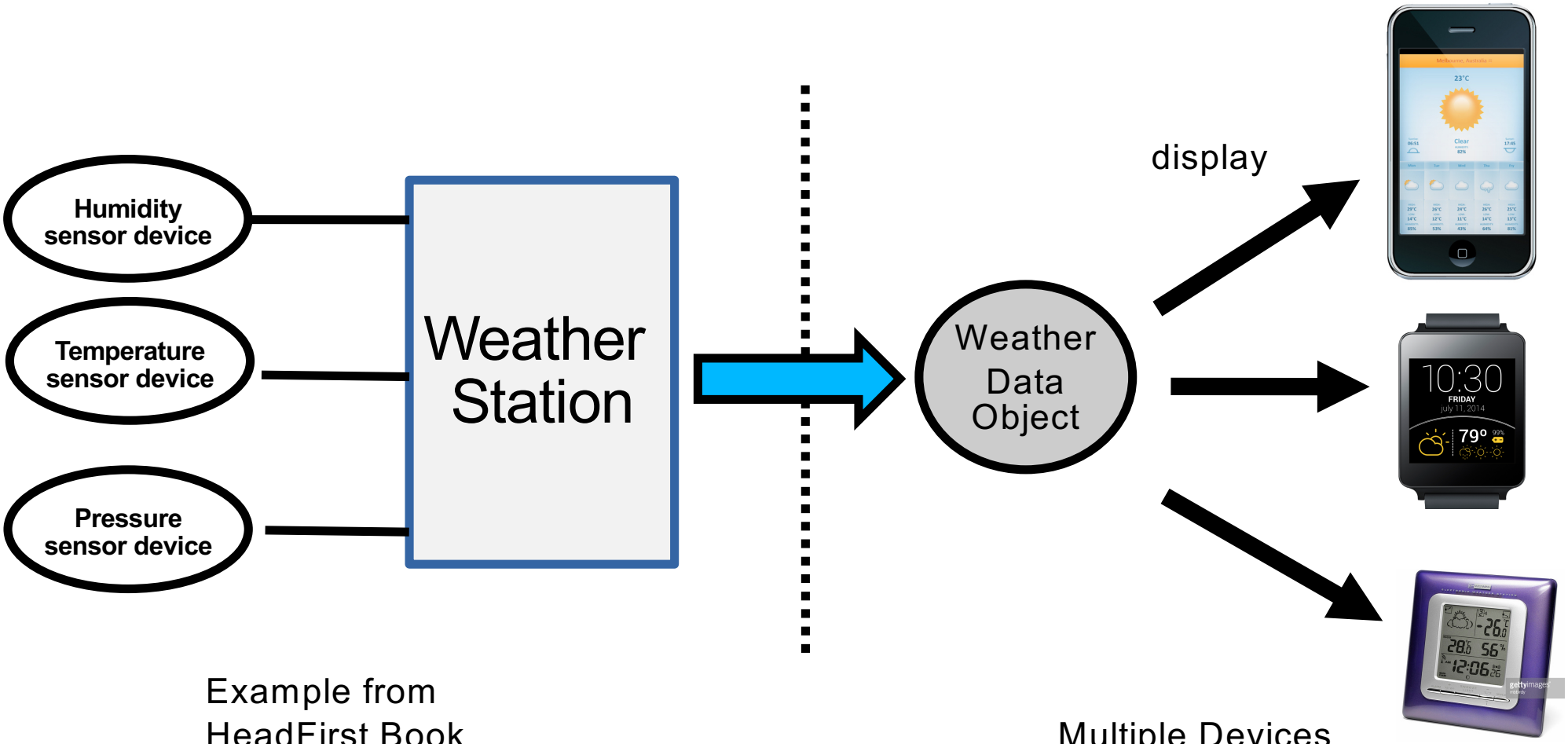
Example: Stock Market Exchange – Stock Prices

- Stock market prices are rapidly changing data.
- We need to inform multiple parties about the current prices.



Example -2 : Weather Station

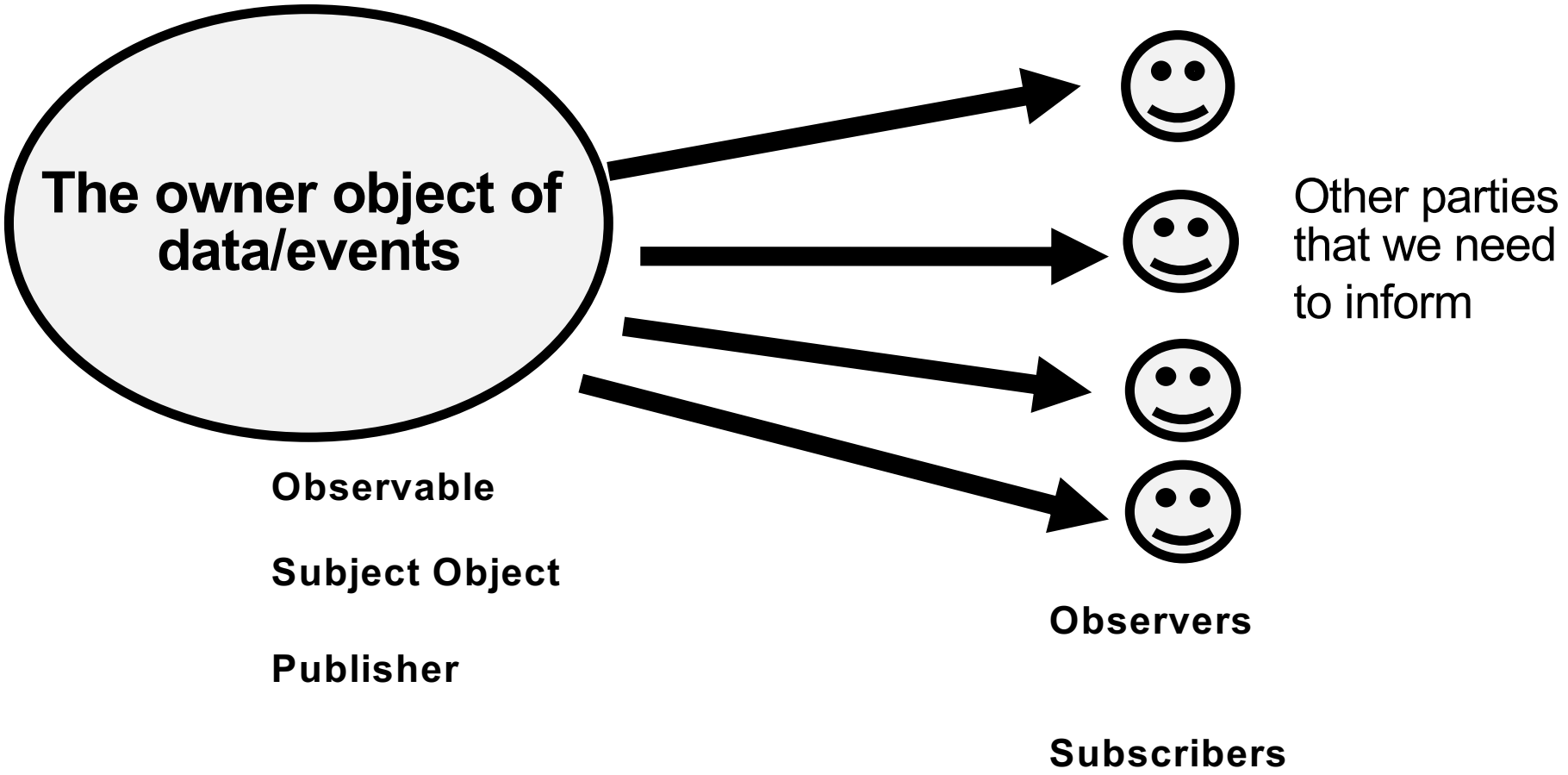
- Data is generated in weather station. We want to display the current weather conditions on multiple devices.



Example from HeadFirst Book

Multiple Devices

Conventions and Names



We will use these names.

Examples – Use Cases

- Sensor-based systems (like monitoring systems):
  - Systems that produce data by using sensors.

    The data needs to be sent to other components to process and shown on multiple devices.
- Chat rooms:
  - a new message in the chat room needs to be broadcasted to all participants
- Online Forums and Groups
- Social Networks, like twitter, facebook, youtube, …
- Online Services like Uber, Lyft, …

The problem is very common.

“Observer Pattern” has verity of applications.

Problem

- You need to notify a varying list of objects that an event has occurred.
- How should we get other components updated considering the requirements and optimal costs?
- Some of the important requirements:
  - Subject **should know less** about the subscribers.
  - We should **be able add new observers** at any time.
  - We should **not need to modify the subject** to add new types of observers.
  - **Changes** to either the subject object or an observer **should not affect** the other.

We want to have Loosely coupled designs.

First Solution

- Polling solution. Have a reference to the “event” publisher and call a method to get as a return value the event.
- Ask if there is any update in time intervals loops

- **Disadvantages:**
  - **Wasting of resources** if there is no change
  - We **can not get the notification in time** because we have to wait until time out is reached and ask again.

What are our choices?

- We saw that polling approach is a waste of resources.

- Any other ideas? …
- How would you implement this?

Solution

Whenever we have any updates we will inform the interested parties (subscribers).

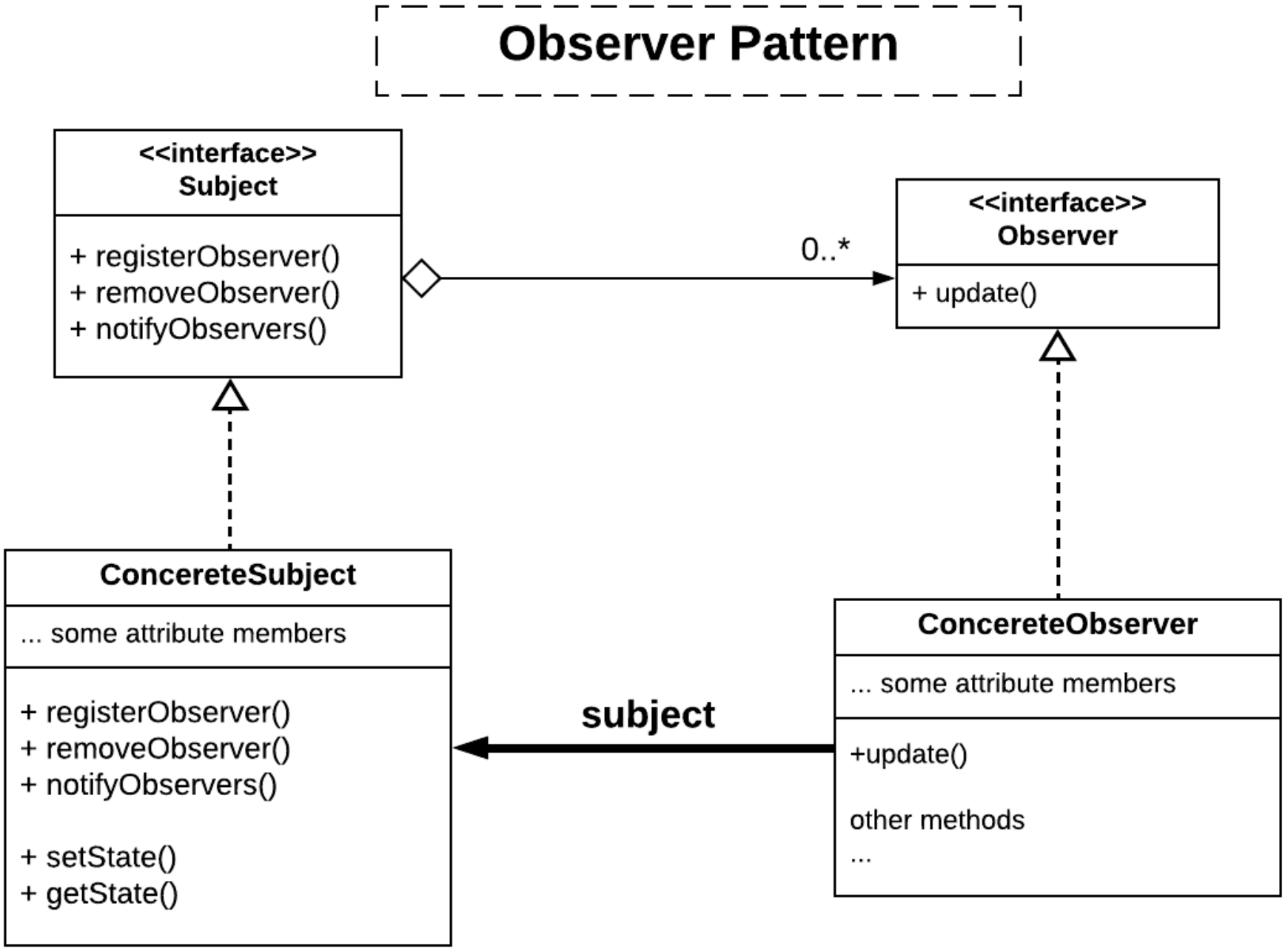
- **Push vs. Poll**

Pushing the new events from the publisher to subscribers.

Observer Pattern

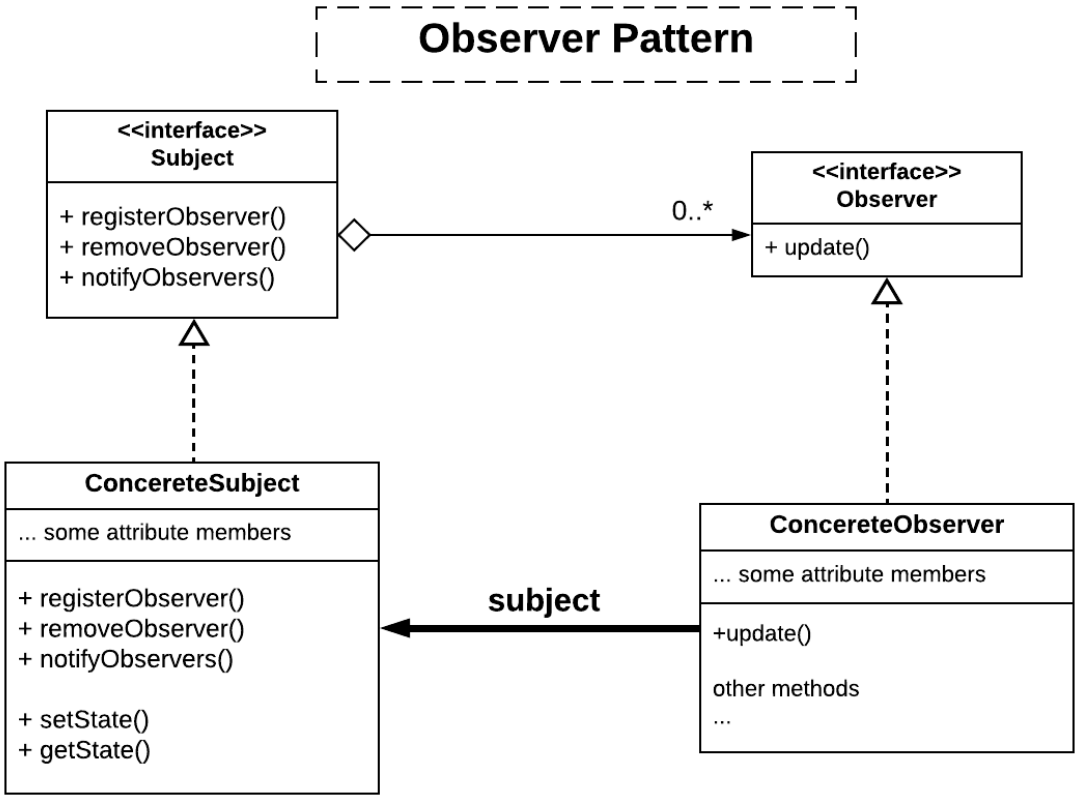
- The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

- Observers **delegate** the **responsibility** for monitoring for an event to a central object named “the Subject” or the “Observable”.

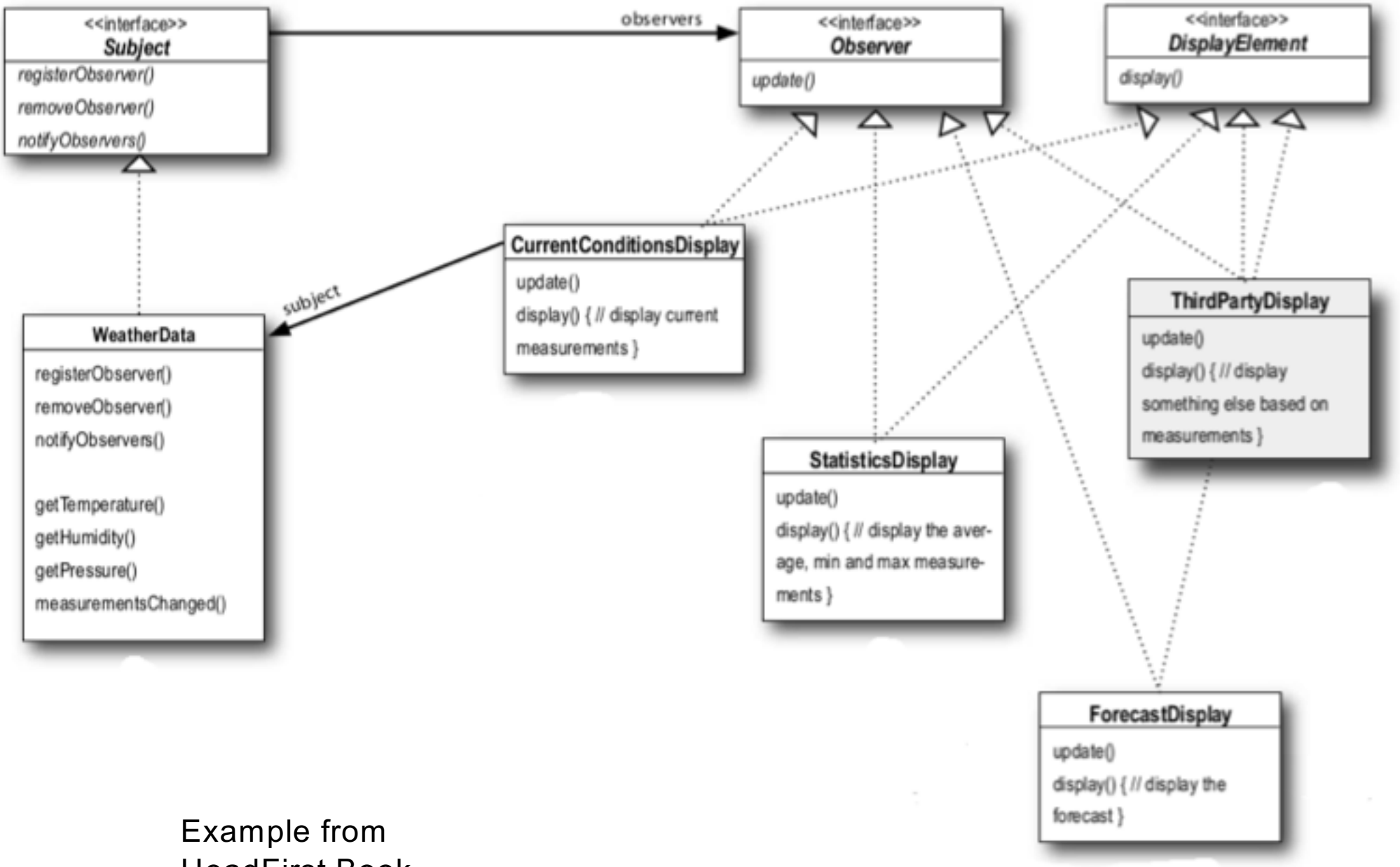

Observer Pattern
<<interface>>
Subject
+ registerObserver()
+ removeObserver()
+ notifyObservers()
0..*
<<interface>>
Observer
+ update()
ConcereteSubject
... some attribute members
+ registerObserver()
+ removeObserver()
+ notifyObservers()
+ setState()
+ getState()
subject
ConcereteObserver
... some attribute members
+update()
other methods
...

Loosely coupled

- **Loosely coupled designs** allow us to build flexible object-oriented applications that can push updates and handle changes because **they have reduced the interdependency level between objects.**



Example: Weather Station



Example from
HeadFirst Book

When to use the Observer Pattern?

- Use the Observer pattern when:
  - **a change** to one object **requires** changes in others so that the changes needed to be forwarded to others, specially when you don't know how many objects need to be changed.
  - **an object should be able to notify other objects** without having knowledge about them and being able to make assumptions about them. Simply, you don't want these objects be tightly coupled.
  - an **abstraction has two aspects, one dependent on the other.** We can **encapsulating** the **aspects in separate objects** so that we can vary them and reuse independently.

Consequences - Positives

The observer pattern lets the programmer vary subjects and observers independently.

Subjects can be reused without reusing their observers, and vice versa.

Observers can be modified without changing the subject or other observers.

+ Abstract coupling between Subject and Observer.

+ Support for broadcast communication.

Polymorphic Encapsulation

- **Subject does not to know what kind of observer it is communicating with**. If we have a new observer, no change is need to be done on Subject side.

This kind of interaction in observer pattern is also known as publish-subscribe (Pub/Sub).

Publish-Subscribe system is a topic in distributed event-based systems.

Consequences - Negatives

- Unexpected updates.

Any changes in Subject cause lots of push events in Observers and Observers have no knowledge about what exactly is changed there.

Without additional protocol to help observers discover what exactly is changed, observers are forced to work hard to deduce the changes.

java.util.**Observable** and java.util.**Observer**

Java.util package includes an Interface and a Class

public **class Observable** and public **interface Observer**

You need to extend the class Observerable and then implement the interface observer

Better implement your own Observerable class than extending java.util.Obserable Class

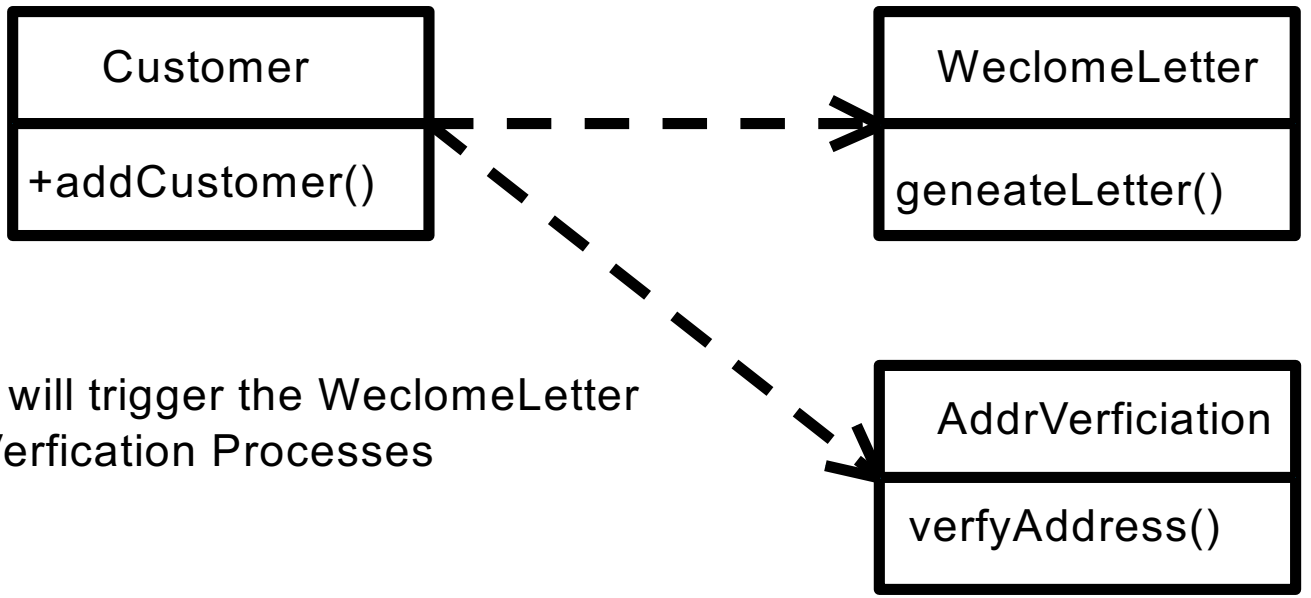
```
import java.util.Observable;
import java.util.Observer;

public class StockMarketDisplay implements Observer, Display {

        public StockMarketDisplay(observable) {
                observable.addObserver(this);
        }
        public void update(Observable observable, Object arg) {
                        … do something …
                        // display this
                        display();
                } }
        public void display() {    print or display somewhere …   } }
```

Summary

- The observer pattern defines **a one-to-many push dependency** relation between an observable and many observers so that when the observable has changes (state, events, etc.), all its dependents are notified and updated automatically.
- Objects (observers and observable) are **responsible for themselves**
- Observer class represent the concept of objects that **needed to be notified** about events and it provide a common interface for the subject to notify the observer objects.

Example – 3 Implement this example

For example in a company or an online shop we have new customers. Each time that we have new customer, we want to send out a welcome letter and another email to verify the address (like email address) of the customer.



addCustomer will trigger the WeclomeLetter and AddressVerfication Processes

Example from GangOfFour Book