Design Patterns & Software Architecture

Observer

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Observer

Observer design pattern

Observer design pattern: Let's start with an example

You have been asked to develop a simple stock trading system. The initial requirements are:

1. A stock has a symbol name and a current price.
2. An investor may monitor the current values of her stocks.
3. When a stock price changes, then all monitoring investors of the stock are informed of the change.

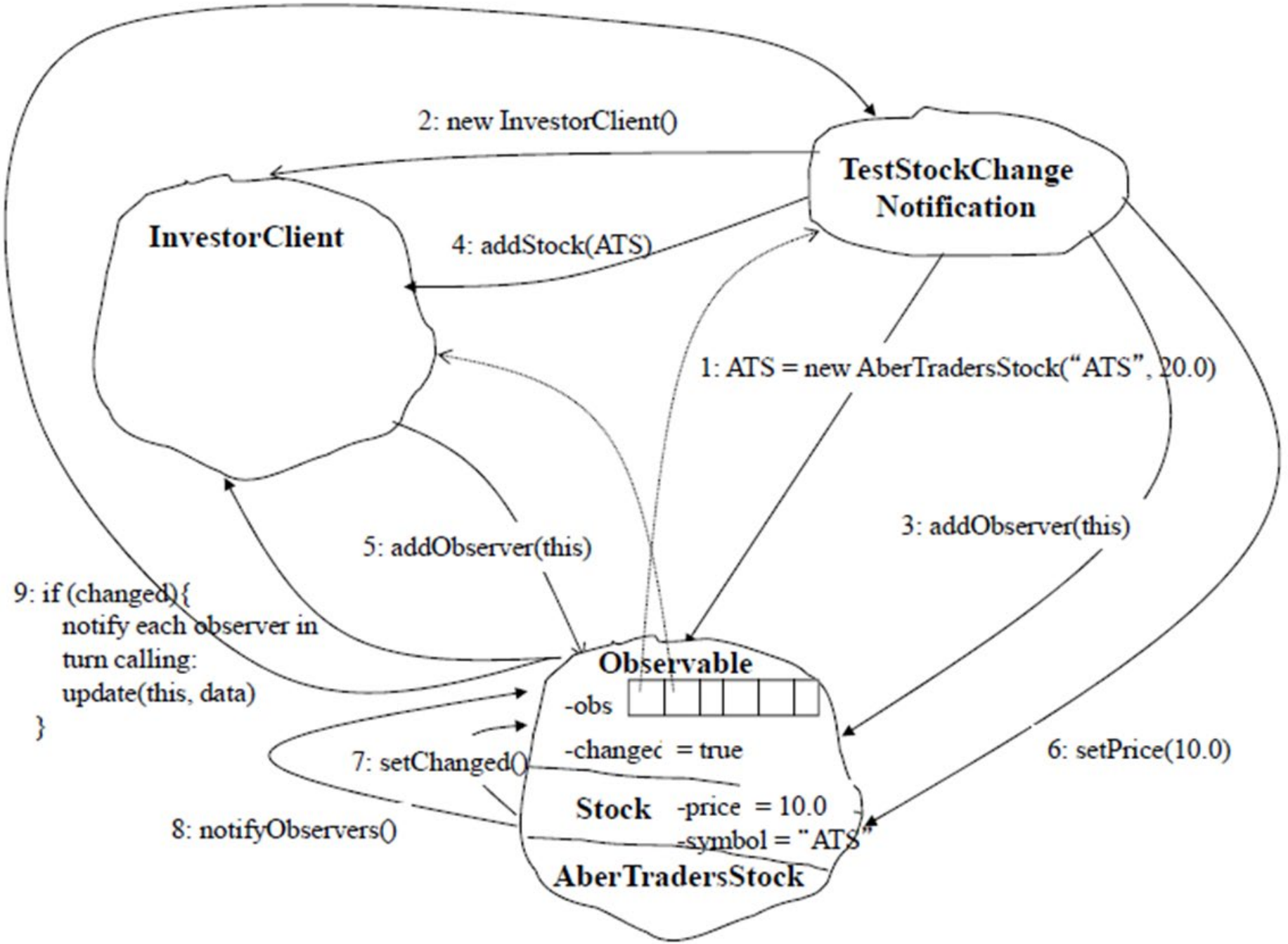
Observer design pattern: Changes to the example

- What happens if we want to add a stock monitor app?

- What happens if we want to add different type of tradeble things with a price, like commodities?

Let’s find a design pattern

Will now present, on the board,
and using Eclipse,
a solution that utilises
the observer design pattern…


2: new InvestorClient()
TestStockChange
Notification
InvestorClient
4: addStock(ATS)
1: ATS = new AberTradersStock("ATS", 20.0)
3: addObserver(this)
5: addObserver(this)
9: if (changed){
notify each observer in
turn calling:
update(this, data)
}
Observable
-obs
-changed = true
7: setChanged()
6: setPrice(10.0)
Stock -price = 10.0
-symbol = "ATS"
8: notifyObservers()
AberTradersStock

Exercise: could fill in this template?

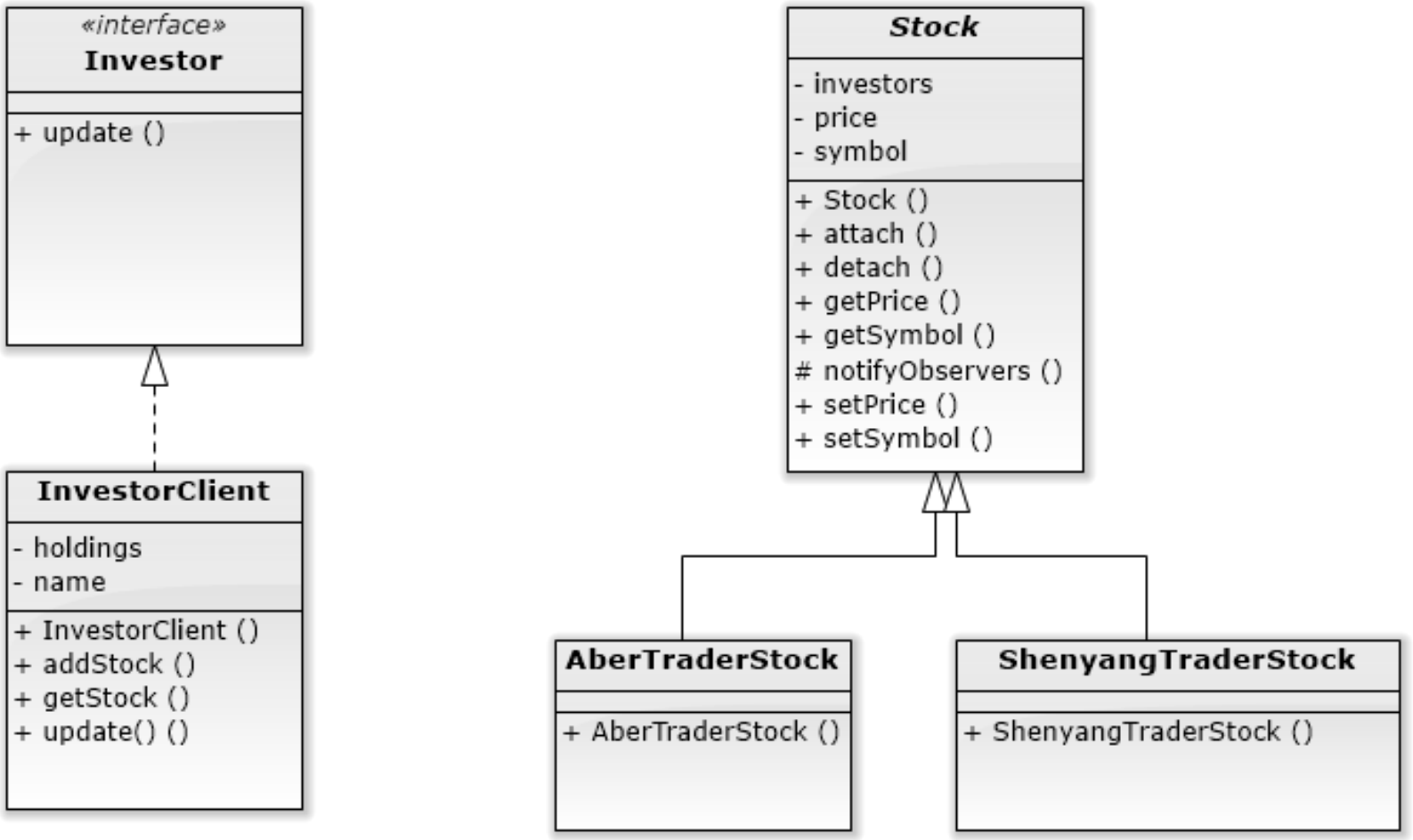
- Pattern name
- Problem
- Solution
- Consequences

- Remember:
  - The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
  - The **problem** describes when to apply the pattern. It explains the problem and its context.
  - The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation.
  - The **consequences** are the results and trade-offs of applying the pattern.

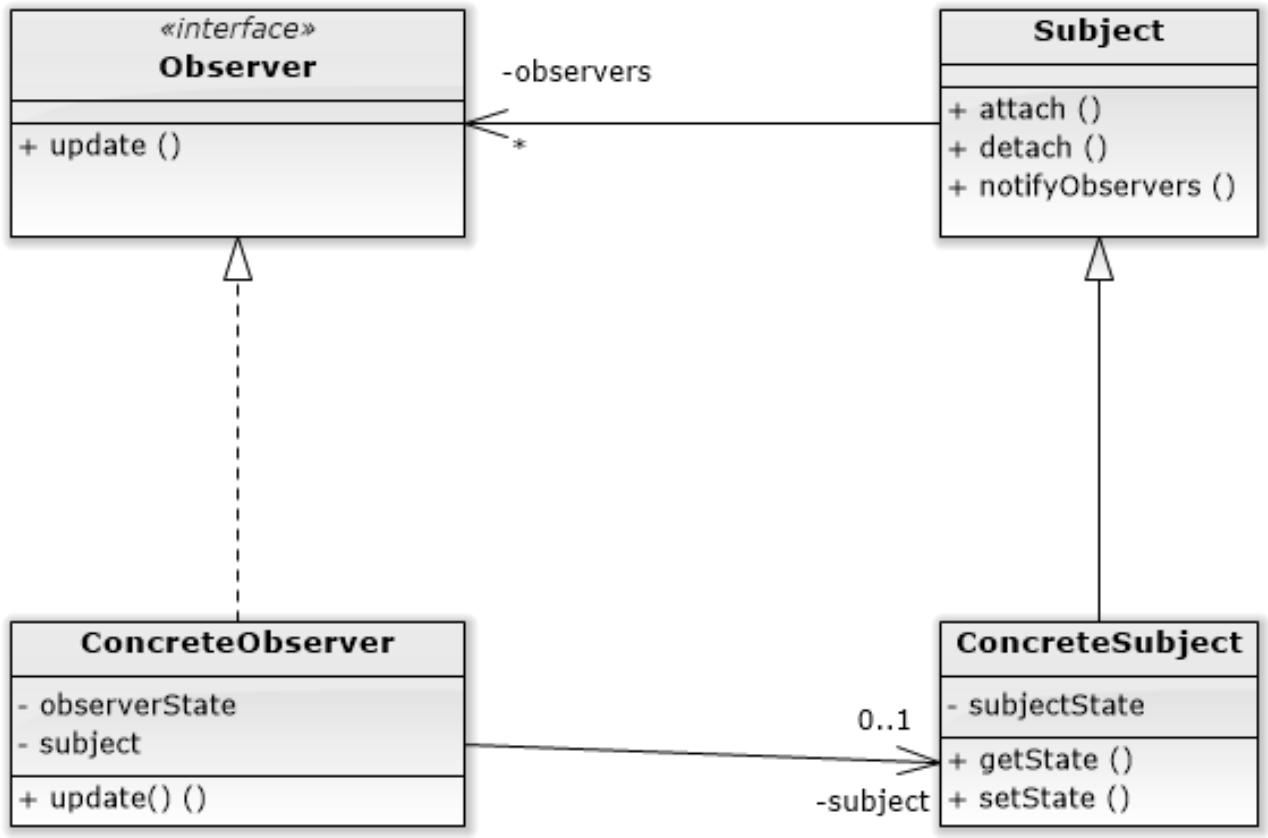
Observer pattern definition

- **Problem:**
  - **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
  - **Motivation**: When objects exchange data to maintain consistency, the high cohesion/low coupling principle encourages these objects to cooperate in a loosely coupled way. An example of one way to achieve this is shown on the next slide.

Motivation


«interface»
Investor
+ update ()
InvestorClient
- holdings
- name
+ InvestorClient ()
+ addStock ()
+ getStock ()
+ update() ()
Stock
- investors
- price
- symbol
+ Stock ()
+ attach ()
+ detach ()
+ getPrice ()
+ getSymbol ()
# notifyObservers ()
+ setPrice ()
+ setSymbol ()
AberTraderStock
+ AberTraderStock ()
ShenyangTraderStock
+ ShenyangTraderStock ()

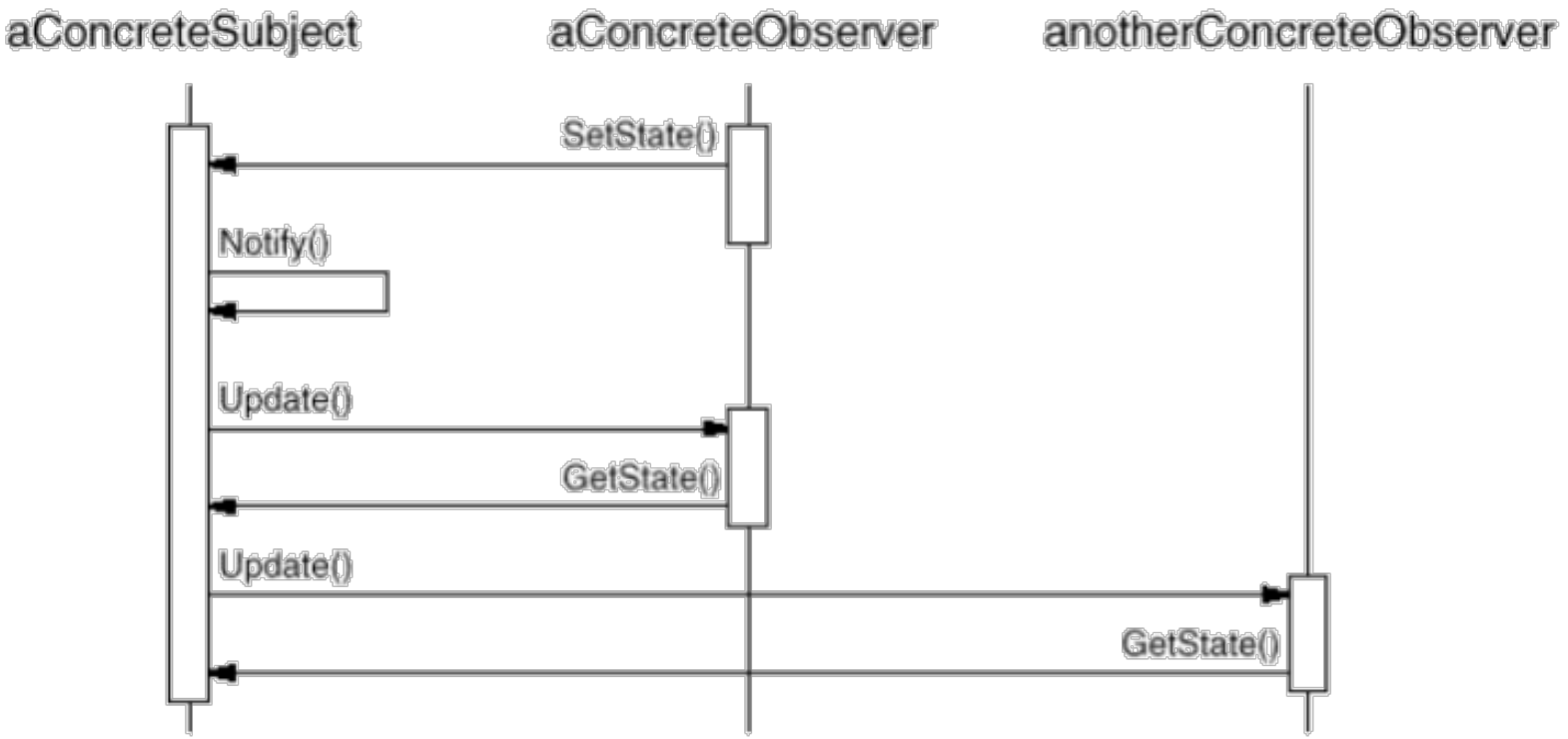
- **Solution:**
  - **Structure:**



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 328

- **Solution:**
  - **Participants:**
    - Subject (Stock)
      - Knows its observers…
      - Provides an interface for attaching/detaching observers.
    - Observer (Investor)
      - Defines an updating interface for notification…
    - ConcreteSubject (AmsTradersStock/ShenyangTradersStock)
      - Stores state of interest to ConcreteObserver…
      - Sends notification to observers when state changes.
    - ConcreteObserver (InvestorClient)
      - Implements Observer in order to keep its state consistent with subject's.

- **Solution**
  - **Colaborations**



- **Consequences**:
  The observer pattern leads to…
  - Abstract coupling between Subject and Observer…
  - Support for broadcast communication…
  - Unexpected updates…

Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 329-330

- **Applicability**:
  Use the Observer pattern in any of following situations:
  - When an abstraction has two aspects, one dependent on the other…
  - When a change to one object requires the changing of an unknown number of other objects.
  - When an object should be able to notify other objects without needing to know the implementation types of those objects…

- **Consequences**
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication
  - Unexpected updates
    (costs of updates can sometimes be unexpectedly high)

- **Implementation**:
  Some issues
  - Observing more than one subject instance…
  - Who triggers the notification?...
  - Make sure Subject state is self-consistent before notification…
  - Avoid observer-specific update protocols: the push and pull models…
  - Specifying modifications of interest explicitly…

Java has it’s own Observable pattern

Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class Observable

java.lang.Object
    java.util.Observable

```
public class Observable
extends Object
```

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface `Observer`. After an observable instance changes, an application calling the `Observable`'s `notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism is has nothing to do with threads and is completely separate from the `wait` and `notify` mechanism of class `Object`.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the `equals` method returns true for them.

Since:

JDK1.0

See Also:

`notifyObservers()`, `notifyObservers(java.lang.Object)`, `Observer`, `Observer.update(java.util.Observable, java.lang.Object)`

interface in java.util

Java has it's own Observable pattern

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`addObserver(Observer o)`** Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
`protected void`	**`clearChanged()`** Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the `hasChanged` method will now return `false`.
`int`	**`countObservers()`** Returns the number of observers of this `Observable` object.
`void`	**`deleteObserver(Observer o)`** Deletes an observer from the set of observers of this object.
`void`	**`deleteObservers()`** Clears the observer list so that this object no longer has any observers.
`boolean`	**`hasChanged()`** Tests if this object has changed.
`void`	**`notifyObservers()`** If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`void`	**`notifyObservers(Object arg)`** If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`protected void`	**`setChanged()`** Marks this `Observable` object as having been changed; the `hasChanged` method will now return `true`.

Java has it's own Observable pattern

Can you spot a difference?

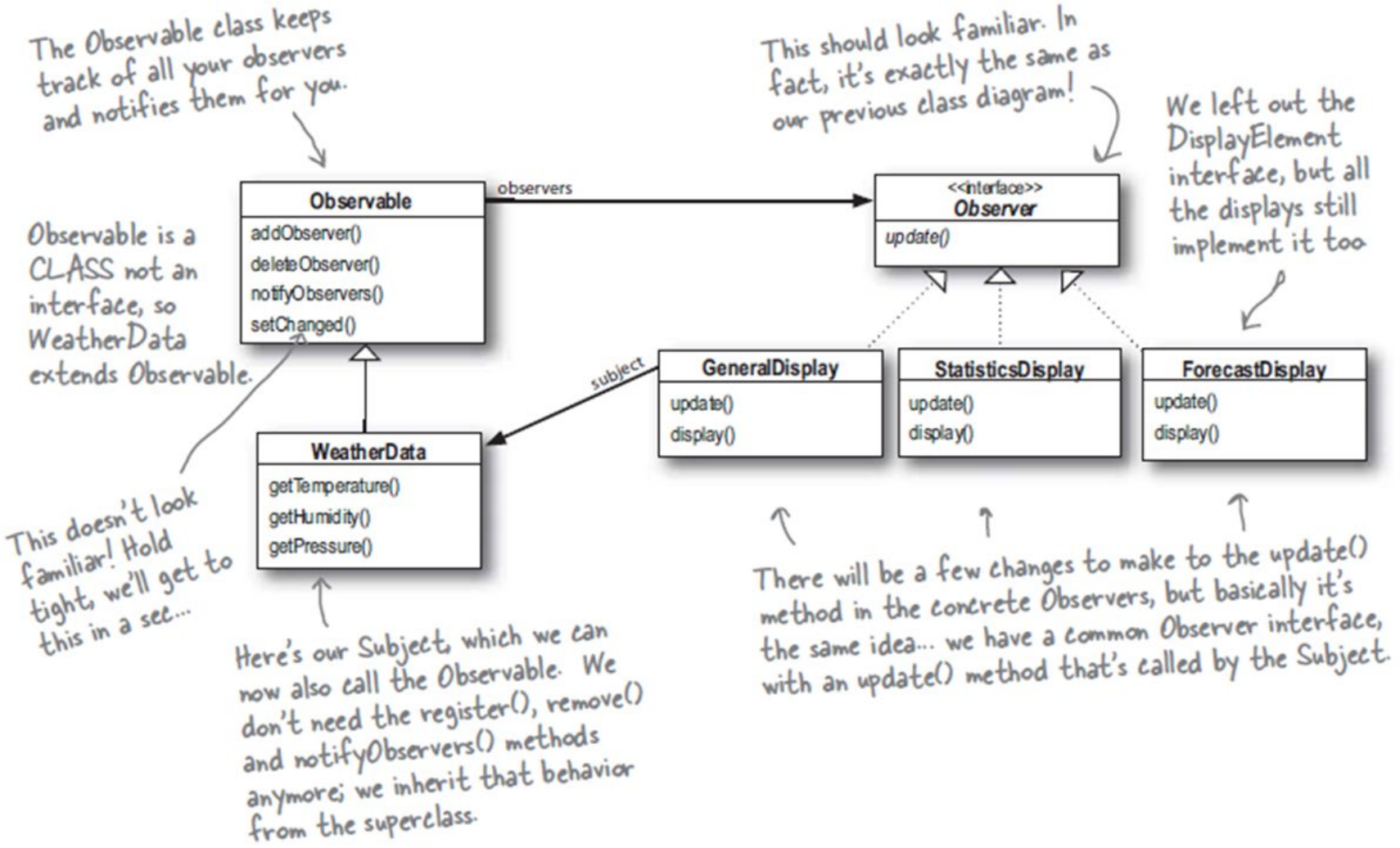
Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`addObserver(Observer o)`** Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
`protected void`	**`clearChanged()`** Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the `hasChanged` method will now return `false`.
`int`	**`countObservers()`** Returns the number of observers of this `Observable` object.
`void`	**`deleteObserver(Observer o)`** Deletes an observer from the set of observers of this object.
`void`	**`deleteObservers()`** Clears the observer list so that this object no longer has any observers.
`boolean`	**`hasChanged()`** Tests if this object has changed.
`void`	**`notifyObservers()`** If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`void`	**`notifyObservers(Object arg)`** If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`protected void`	**`setChanged()`** Marks this `Observable` object as having been changed; the `hasChanged` method will now return `true`.

Observable in a schema

from Freeman, Head First Design Patterns, p. 64



Design Principle:
Strive for loosely coupled designs

Strive for loosely coupled designs between objects that interact.

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 53

Reading

For this lesson you had to read:

- Chapter 1 (Welcome to Design Patterns) of Head First Design Patterns

For next lesson please read:

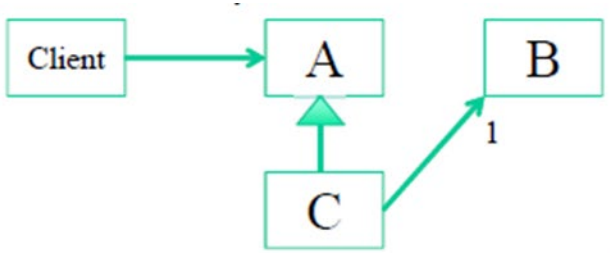
- Chapter 2 (Keeping your Objects in the Know) of Head First Design Patterns
- Chapter 3 (Decorator Objects) of Head First Design Patterns

Please remember, Tuesday there will be a small test!

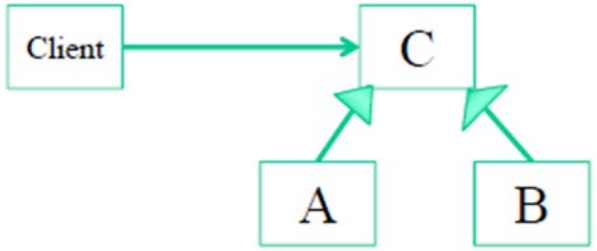
Questions

Study the following three designs. Which of the designs represents the Strategy design pattern?
(Choose one)

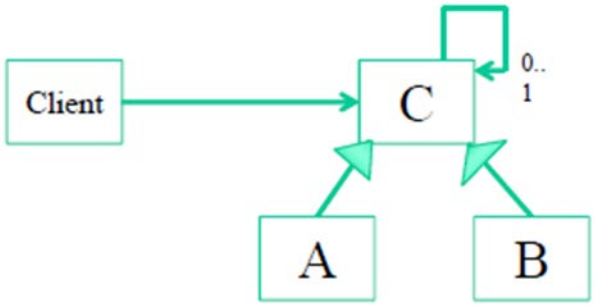
A)



B)



C)



Questions

A

Yes

Light blue

B

No

Red

C

Pink

D

Dark blue