Behavioral Design Patterns

How to design interactions

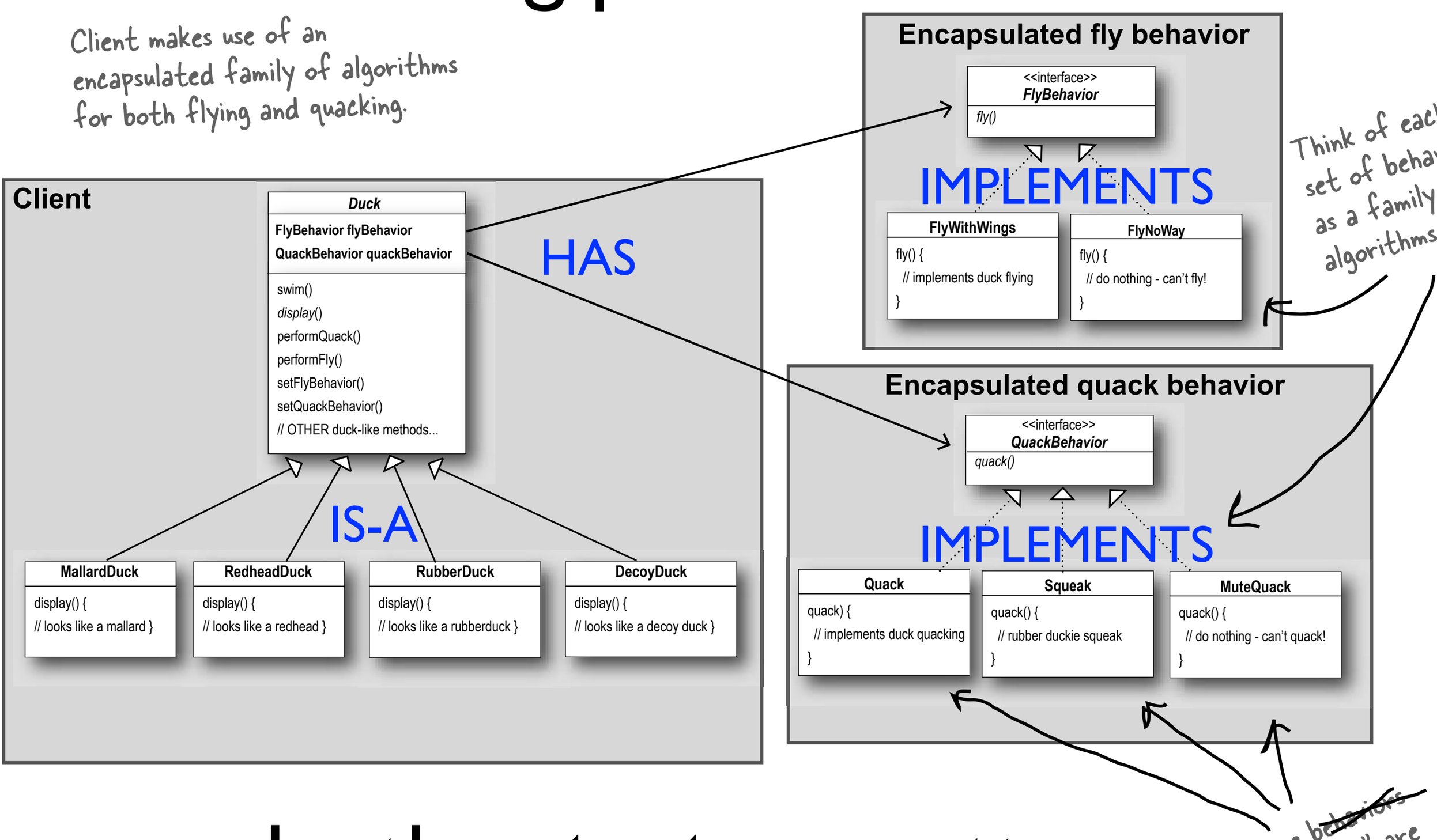
Today's lecture is based on

Slides from:

Tom Zimmermann
Microsoft Research

The big picture

Client makes use of an encapsulated family of algorithms for both flying and quacking.

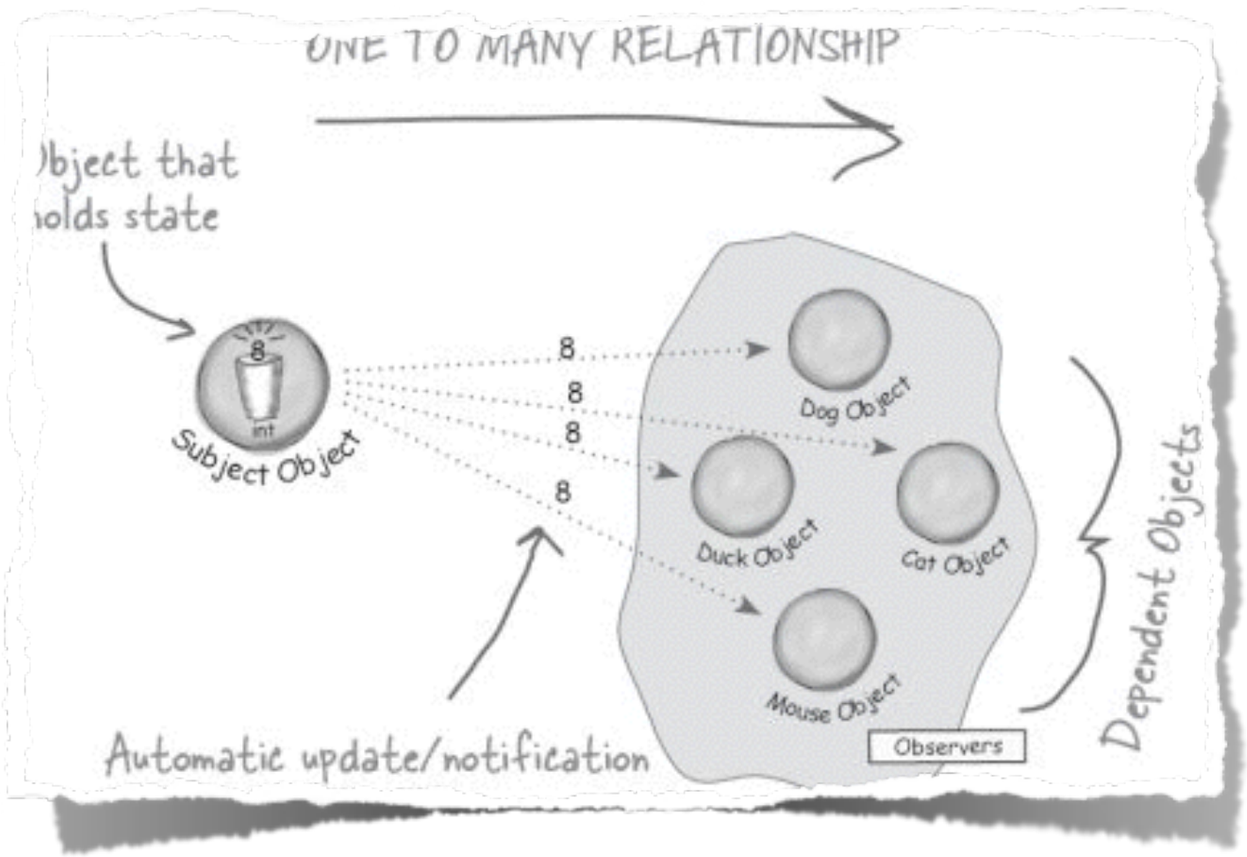


... aka the **strategy** pattern

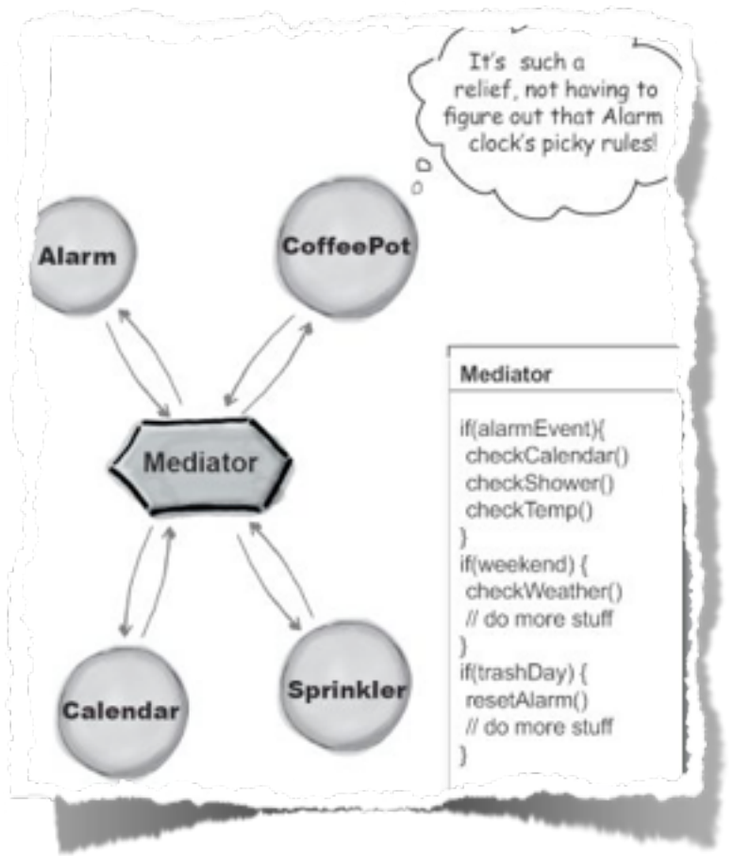
What's the key technique used by design patterns?

- Programming to the interface!!

Today



vs.



Observer

Mediator

Don't miss out when something interesting happens!

- We've got a pattern that keeps your objects in the know when something they might care about happens.
- Objects can even decide *at runtime* whether they want to be kept informed.
- The Observer Pattern is one of the *most heavily used* patterns in the JDK, and it's incredibly useful.
- With Observer, you'll be the life of the Patterns Party.

Meet the observer pattern

- Publisher goes into business and starts publishing a newspaper
- You subscribe and get every new edition delivered to you
- People, hotels, and airlines constantly subscribe and unsubscribe

Marquee display

IPhone 7

```
public publishHeadline(){
   iphone.setContent(news);
   marquee.setContent(news);
   desktop.setContent(news);
 }
```

Desktop

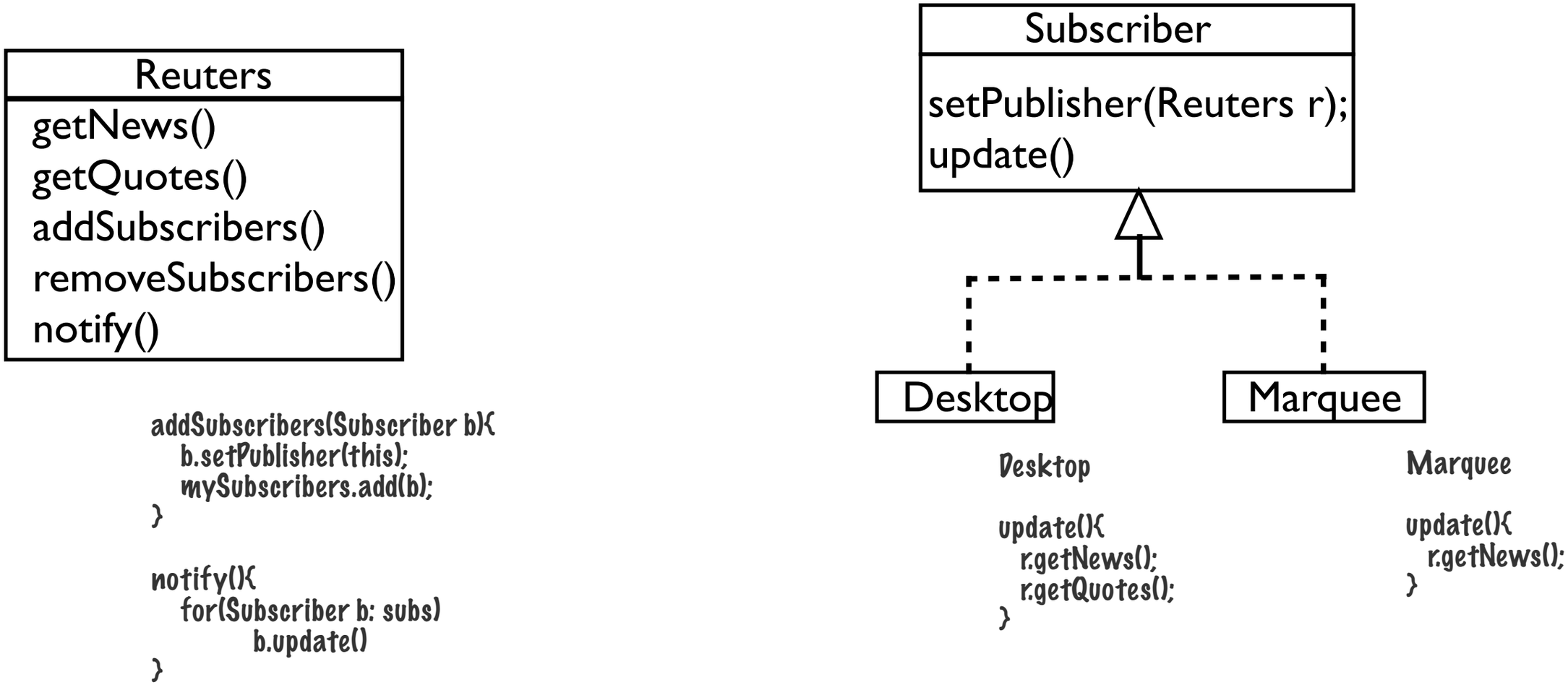
1. What about 10 displays, 20 iphones, and 50 computers?
2. What about supporting a BlackBerry device?
3. What about not supporting IPhone 7 (because we don't have it)?
4. What about displaying stock quotes? And only on Desktops?

All of the above require changes to the source code. Joe is fired again.

Same story once again

- What's wrong with Joe's design?
  - "Hardcoding" the knowledge of implementation
  - "Hardcoding" the knowledge of the updated information
- What is unchanged and what's changing?
  - The types of contents and the display methods are stable.
  - Type of display devices and what device will display are constantly changing

Our new design

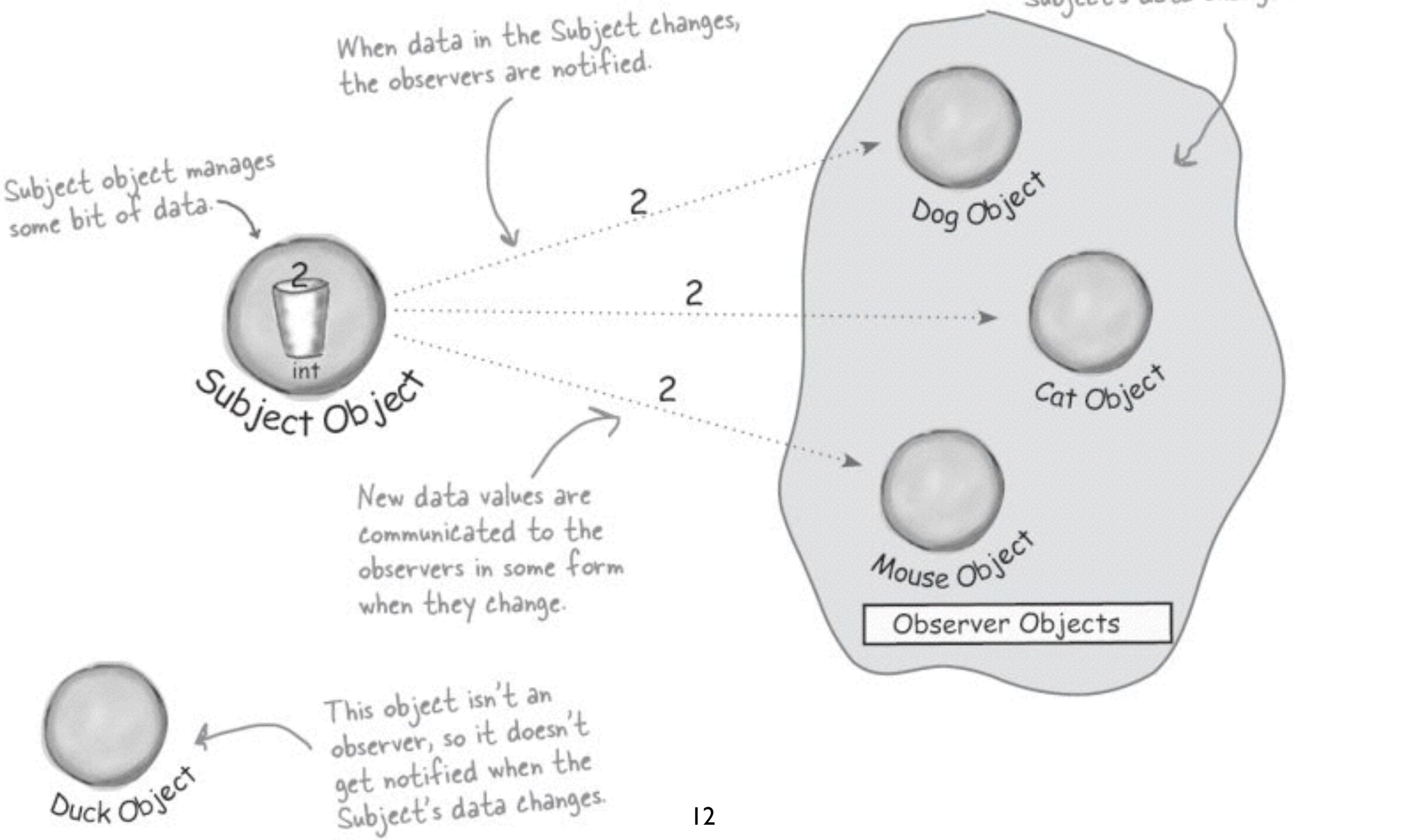


SUBJECT OBSERVER

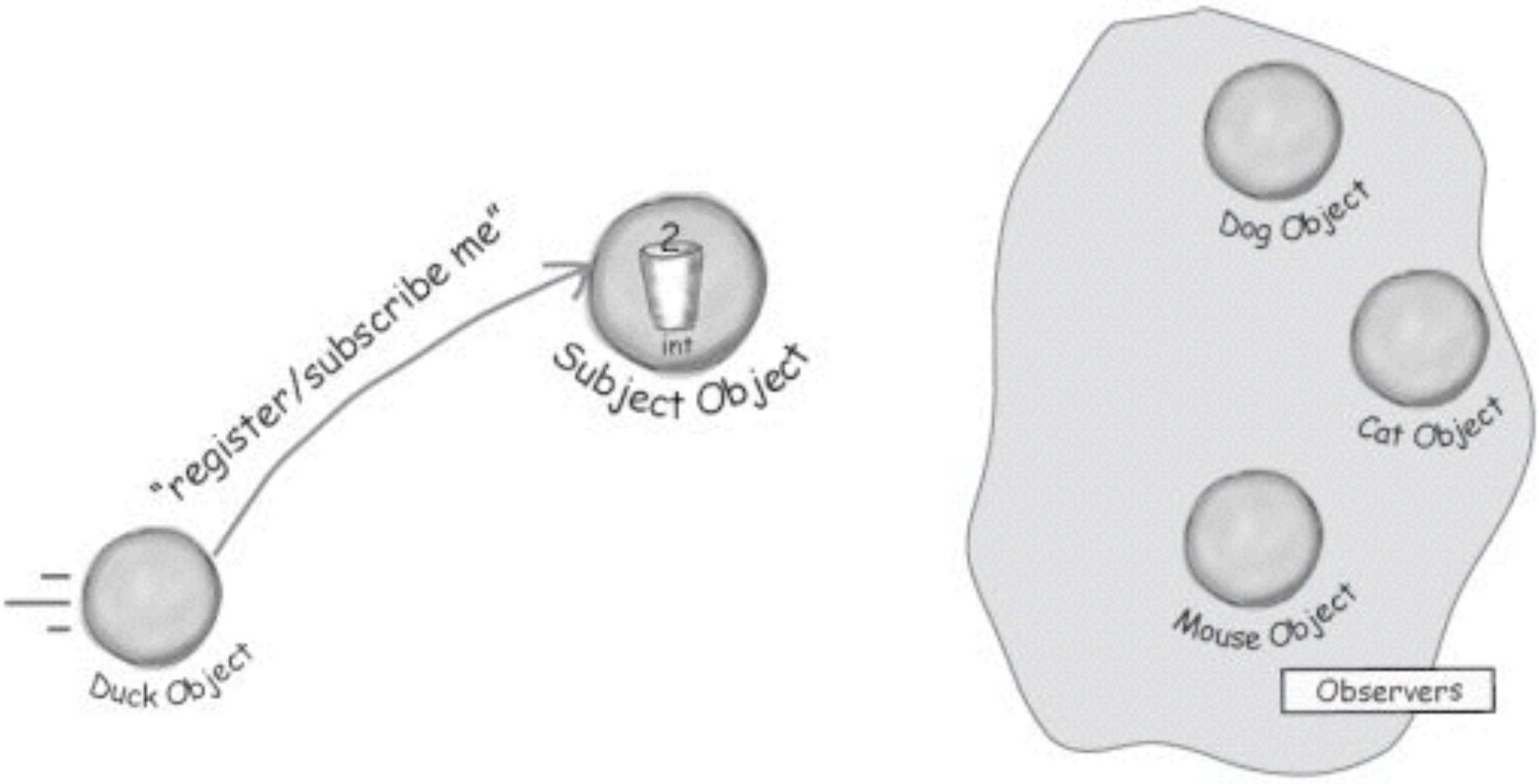
~~Publisher~~ + ~~Subscribers~~

= Observer Pattern

A closer look

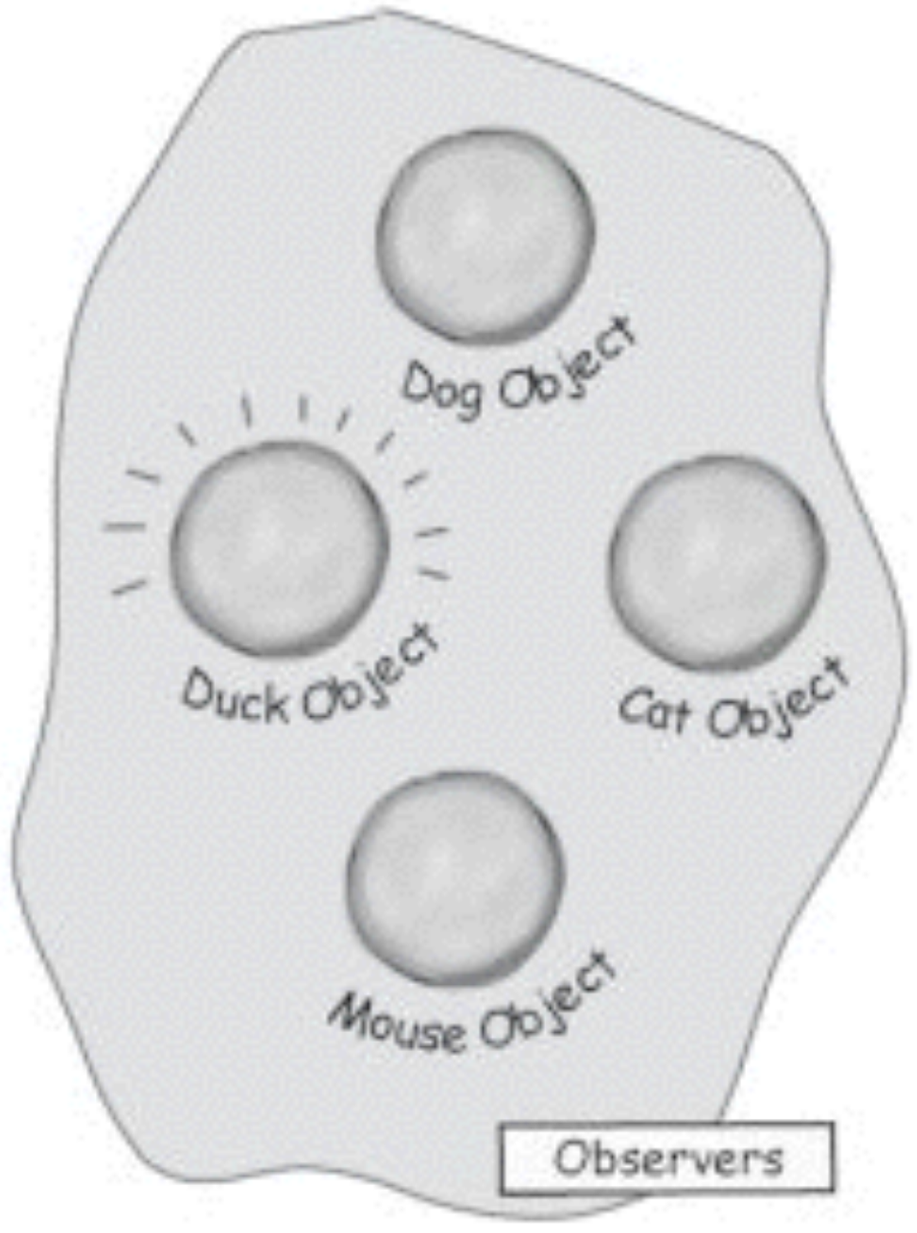


How it works

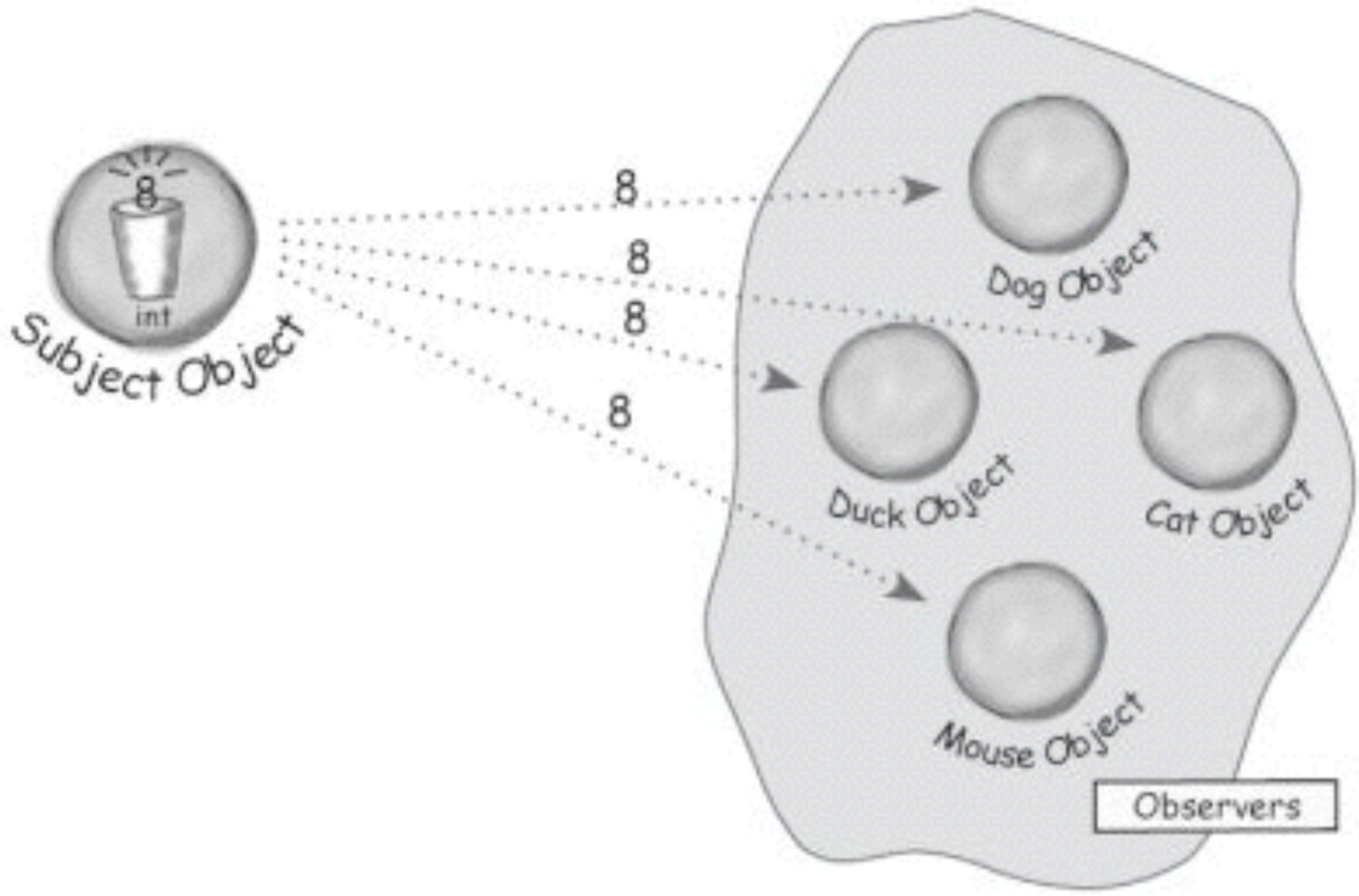

"register/subscribe me"
2
int
Subject Object
Duck Object
Dog Object
Cat Object
Mouse Object
Observers

How it works

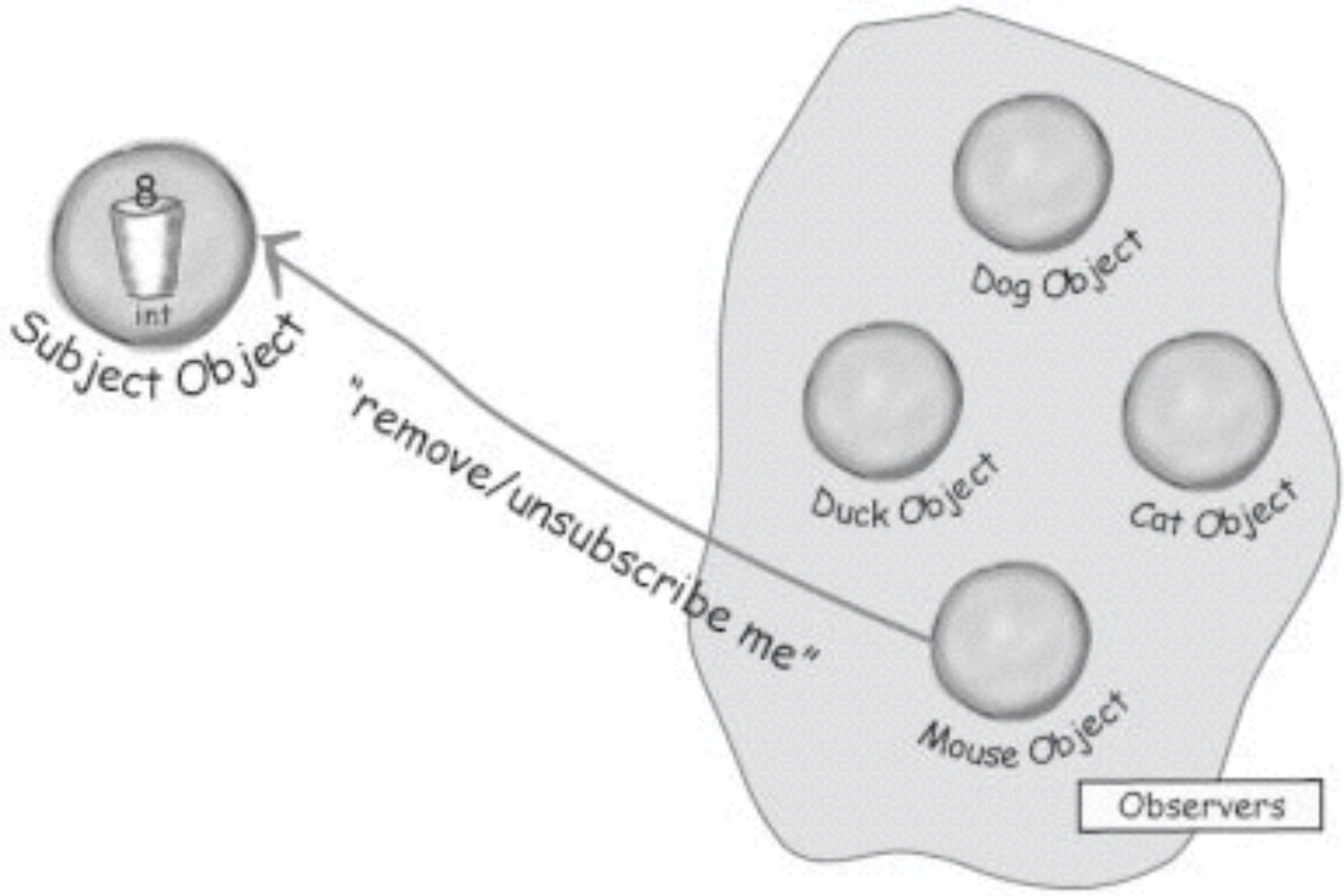

2
int
Subject Object


Dog Object
Duck Object
Cat Object
Mouse Object
Observers

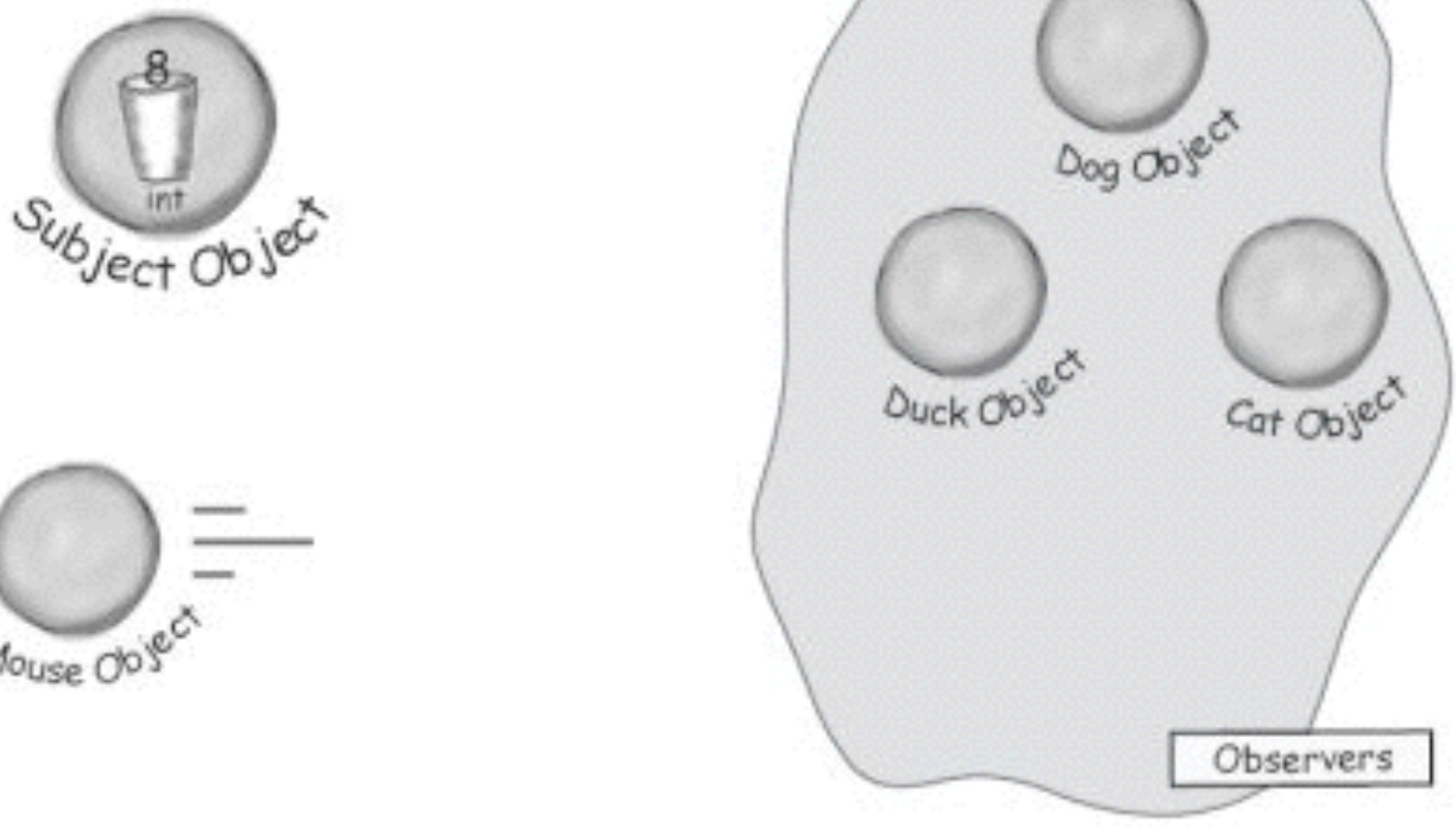
How it works


8
int
Subject Object
8
8
8
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

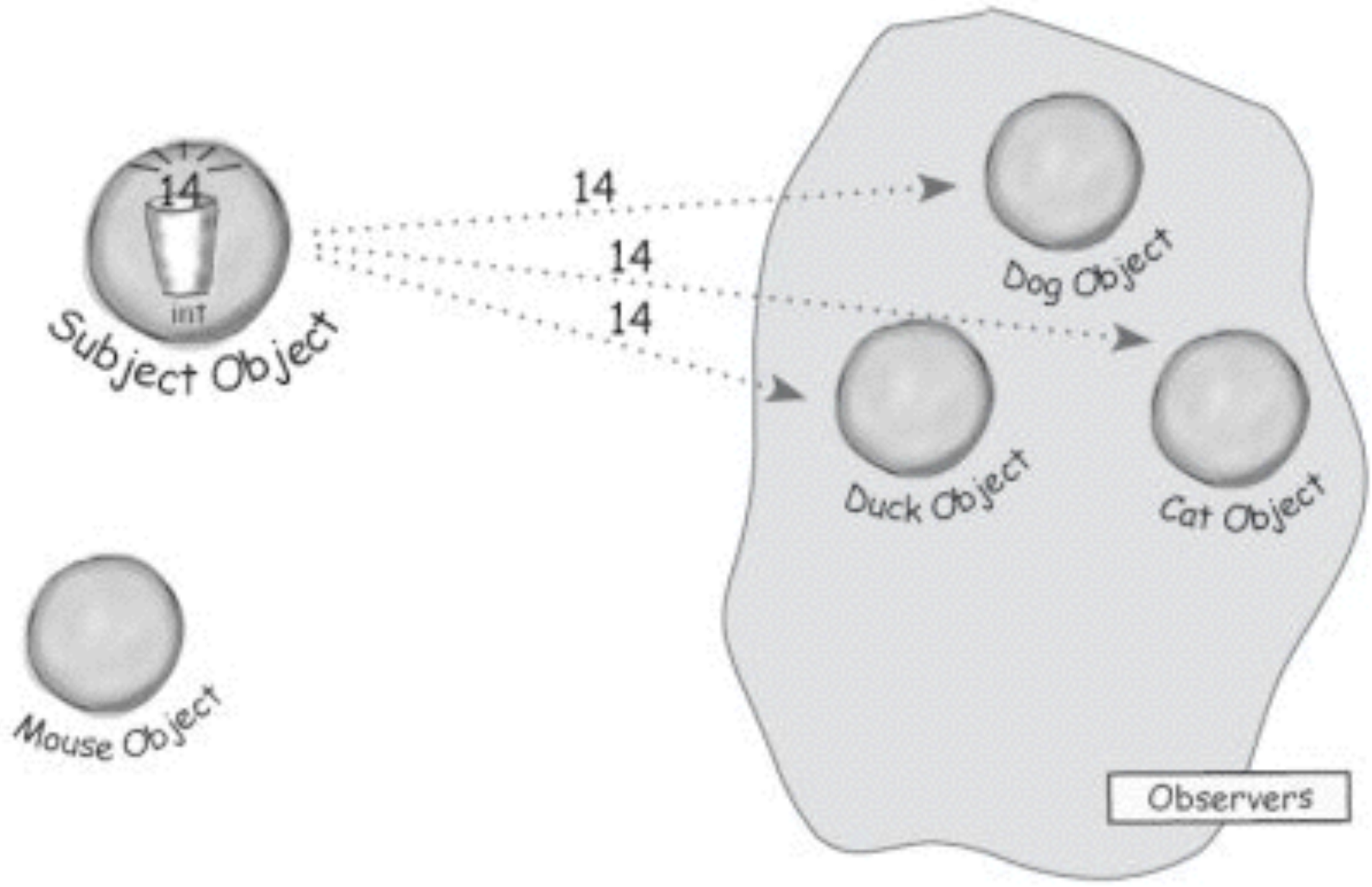
How it works


8
int
Subject Object
"remove/unsubscribe me"
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

How it works


8
int
Subject Object
Mouse Object
Dog Object
Duck Object
Cat Object
Observers

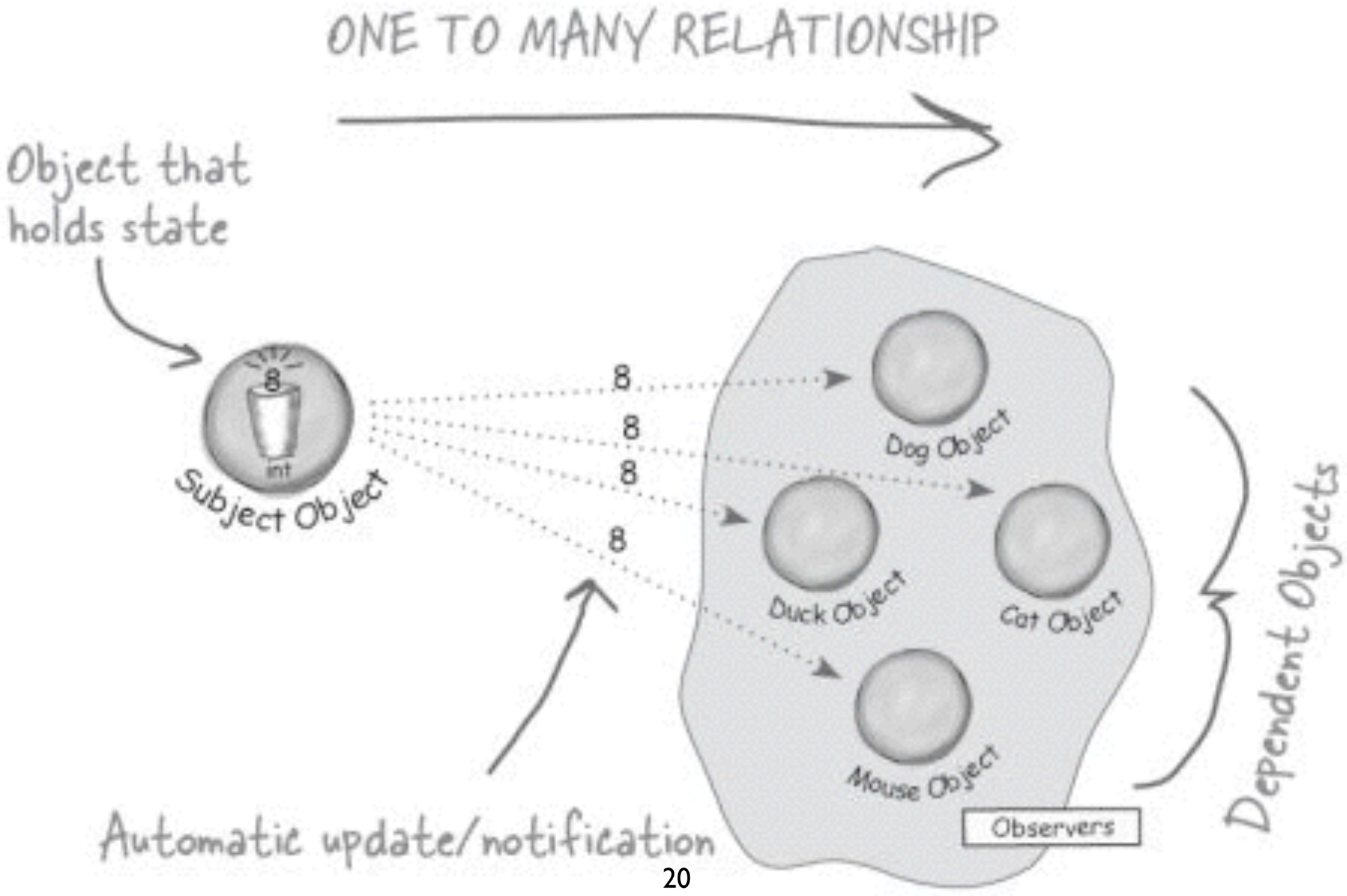
How it works


14
14
14
14
Subject Object
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

The definition

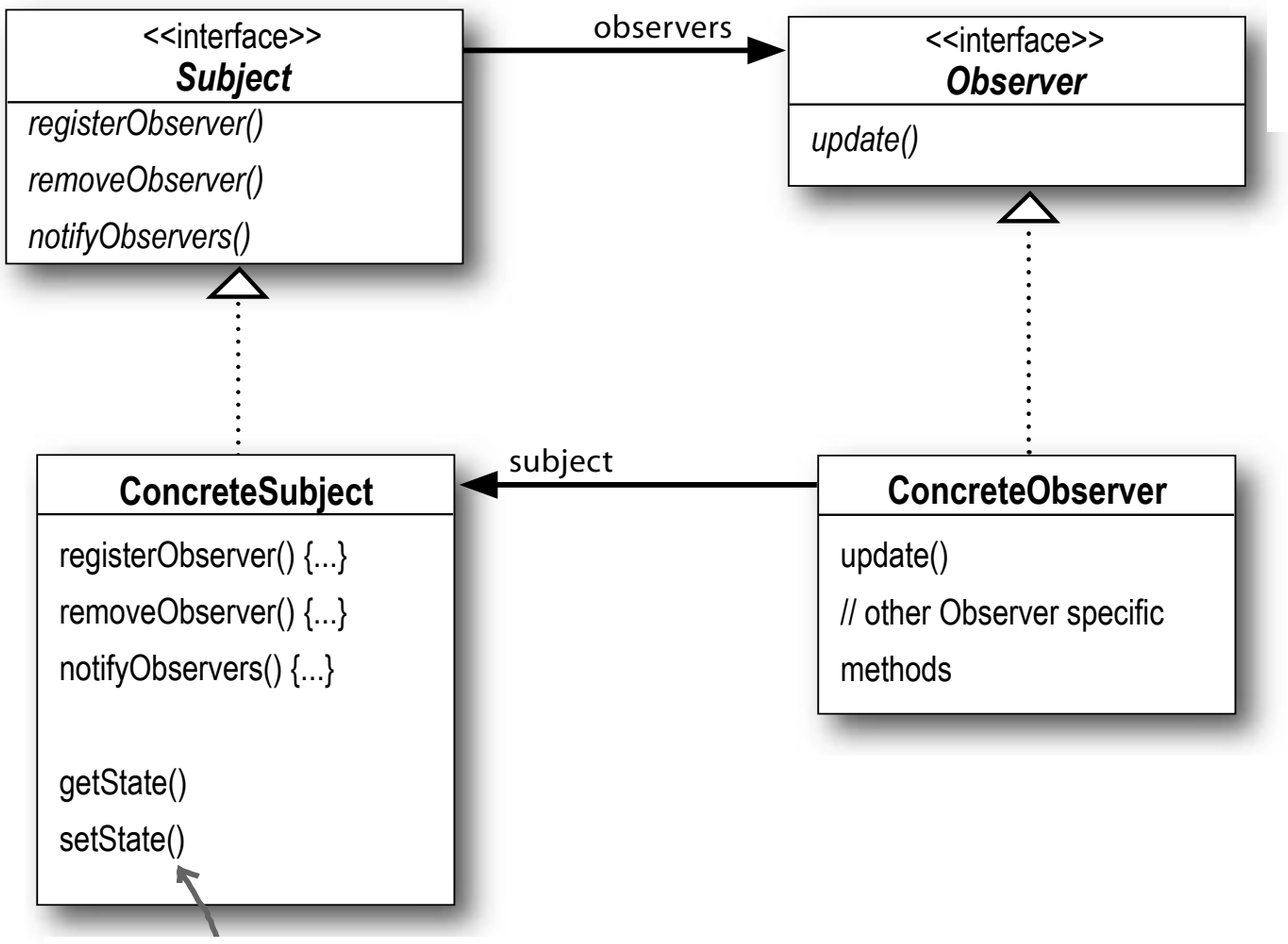
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The definition



The Observer Pattern defined: the class diagram

Here's the Subject interface.
Objects use this interface to register as observers and also to remove themselves from being observers.

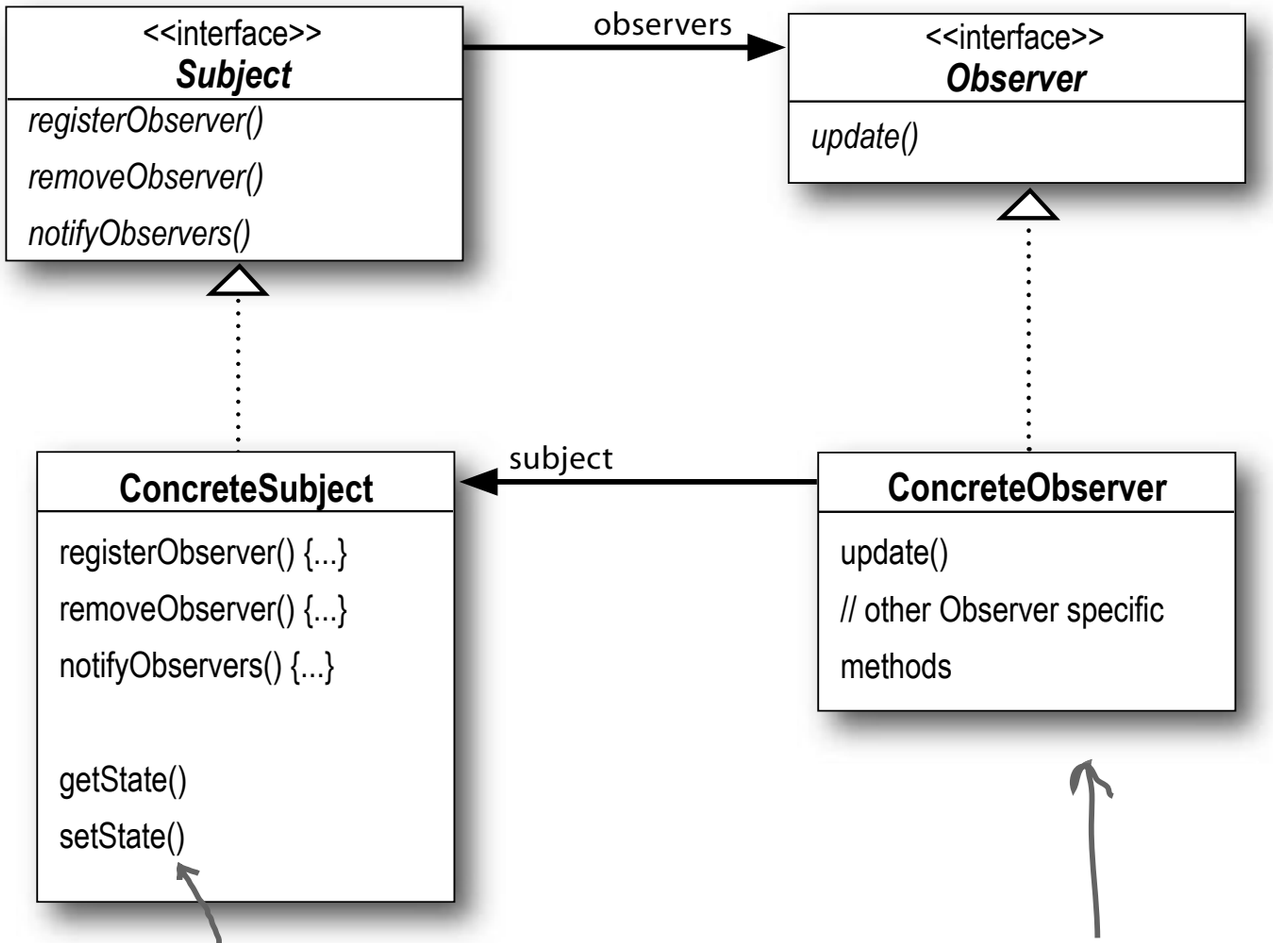


The Observer Pattern defined: the class diagram

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.



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.

The power of loose coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

The power of loose coupling

- Subjects know only that observers implement a certain interface
- We can add new observers at any time
  - and we never need to modify the subject for this
  - also we can easily remove observers
- Changes to either the subject or an observer will not affect each other

Another design principle

Design Principle

Strive for loosely coupled designs between objects that interact.

Loosely coupled designs minimize the interdependency between objects.

Similar Principles

- Hollywood principle
  - "Don't call us, we will call you!"
- Inversion of Control (IoC)

More about observer patterns

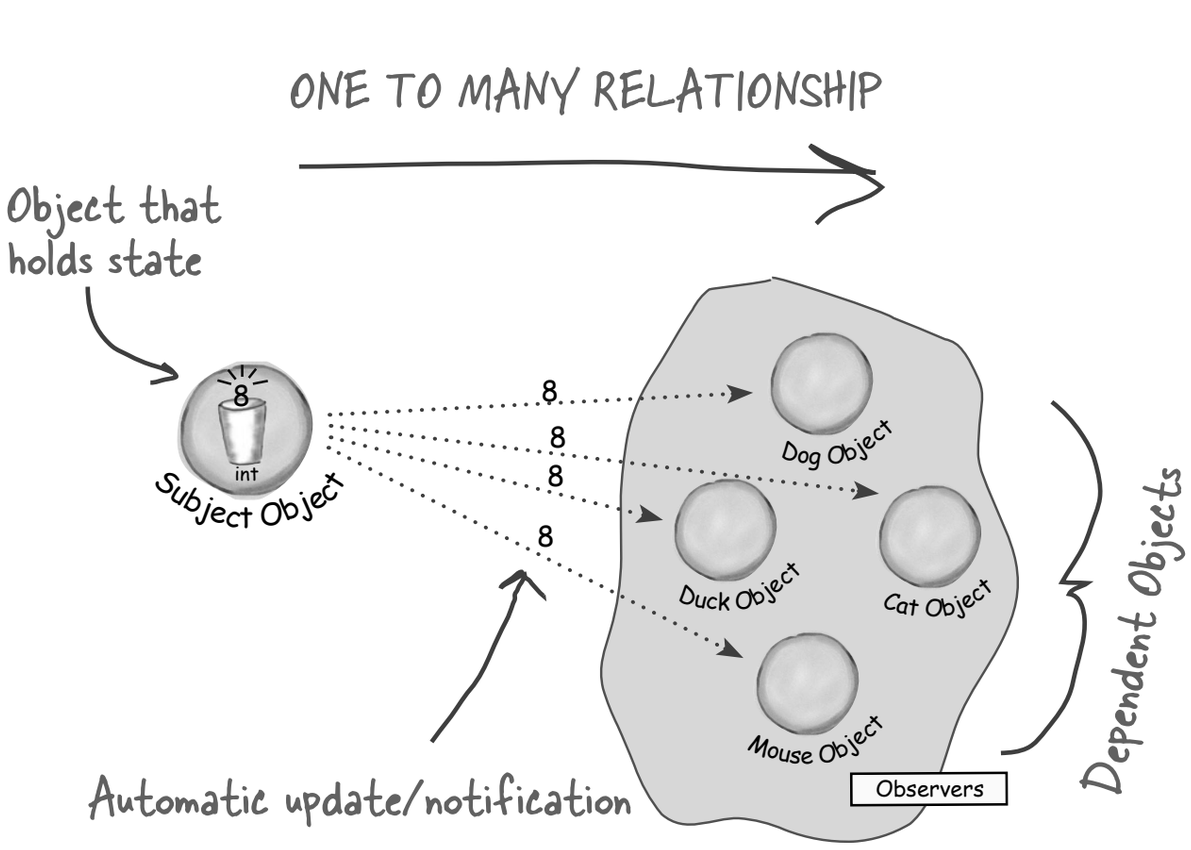
- Support for broadcast communication
  - no need for Concrete Subject to worry about number of observing objects
  - observing objects can change at runtime

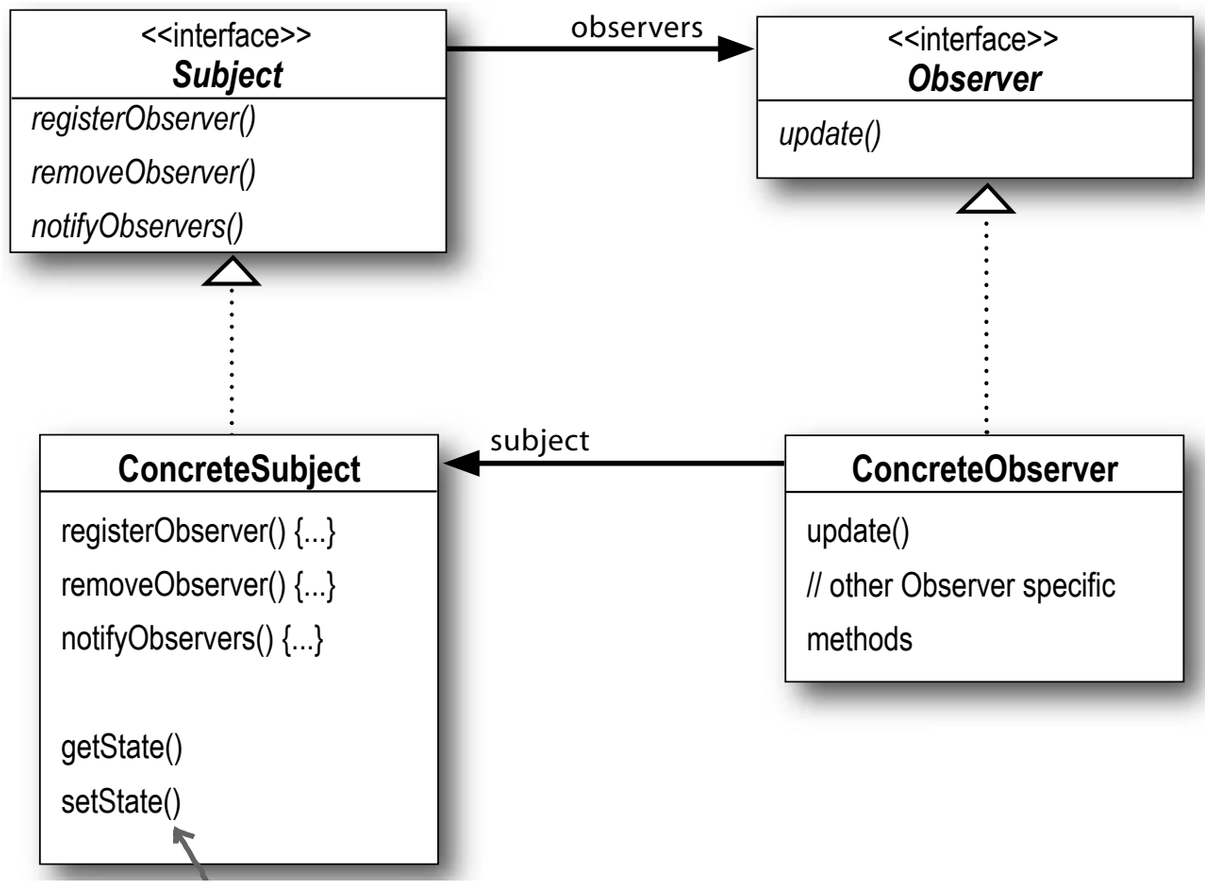
More about observer patterns

- Event-notification protocol needed
  - what events should the subject announce?
  - should every event be announced to every observer?
  - should the subject define different kinds of events and allow the observers to subscribe selectively?

Sharpen your pencil

Before moving on, try sketching out the classes you'll need to implement the Subject/Observers.



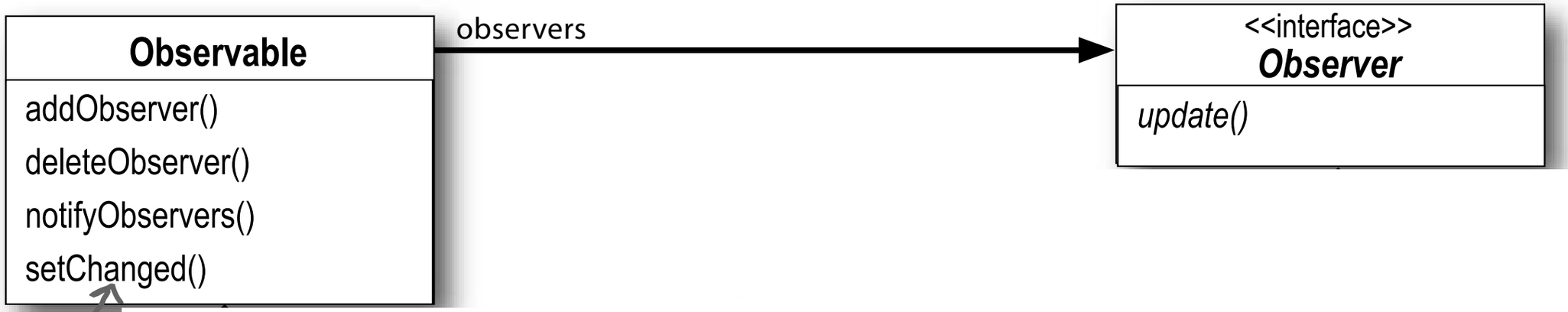


Hint: my answer would be different, but see http://java-x.blogspot.com/2007/01/implementing-observer-pattern-in-java.html

Java's built-in observer pattern

The Observable class keeps track of all your observers and notifies them for you.

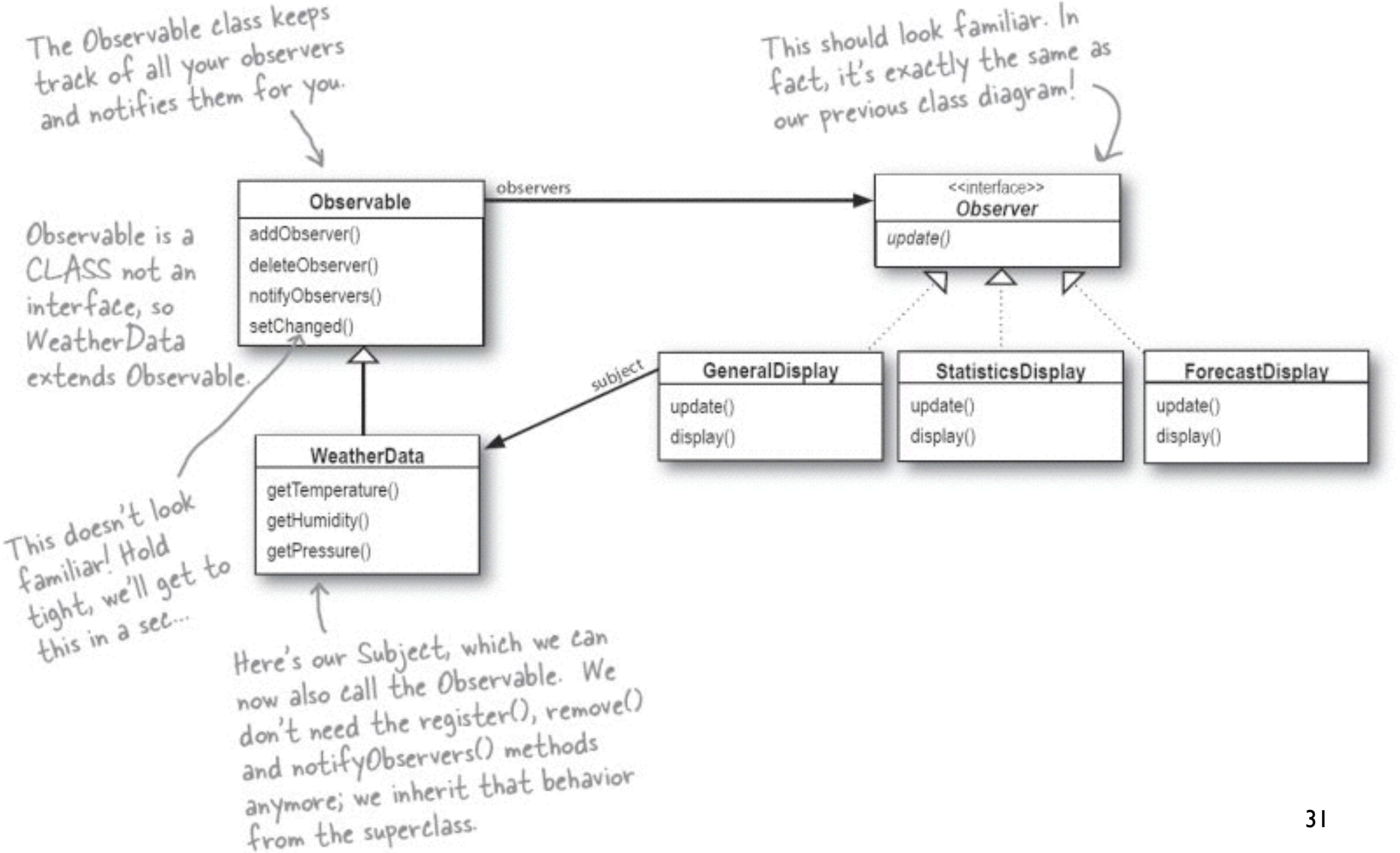
This should look familiar. In fact, it's exactly the same as our previous class diagram!



Observable is a CLASS not an interface, so WeatherData extends Observable.

This doesn't look familiar! Hold tight, we'll get to this in a sec...

Java's built-in observer pattern



How Java's built-in observer pattern works

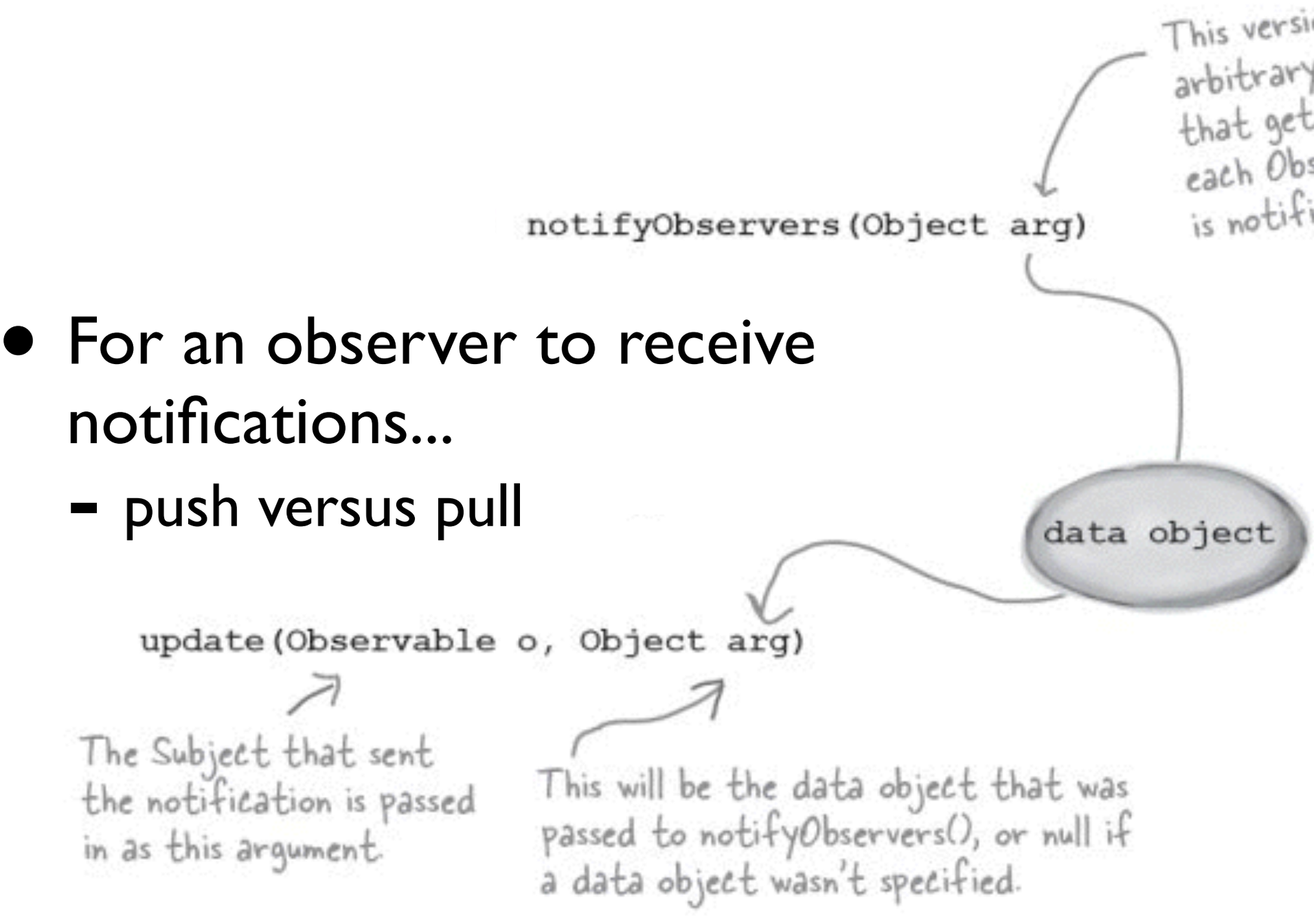
- For an object to become observer...
  - implement the Observable interface
  - to add call addObserver(),
    to remove call deleteObserver() on the subject

How Java's built-in observer pattern works

- For the observable to send notifications...
  - you must call setChanged() to signal that your state has changed
  - then, call either notifyObservers() or notifyObservers(Object arg)

How Java's built-in observer pattern works

- For an observer to receive notifications...
  - push versus pull



About setChanged()

Behind the Scenes

```
setChanged() {
   changed = true
}

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

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

The setChanged() method sets a changed flag to true.

Pseudocode for the Observable Class.

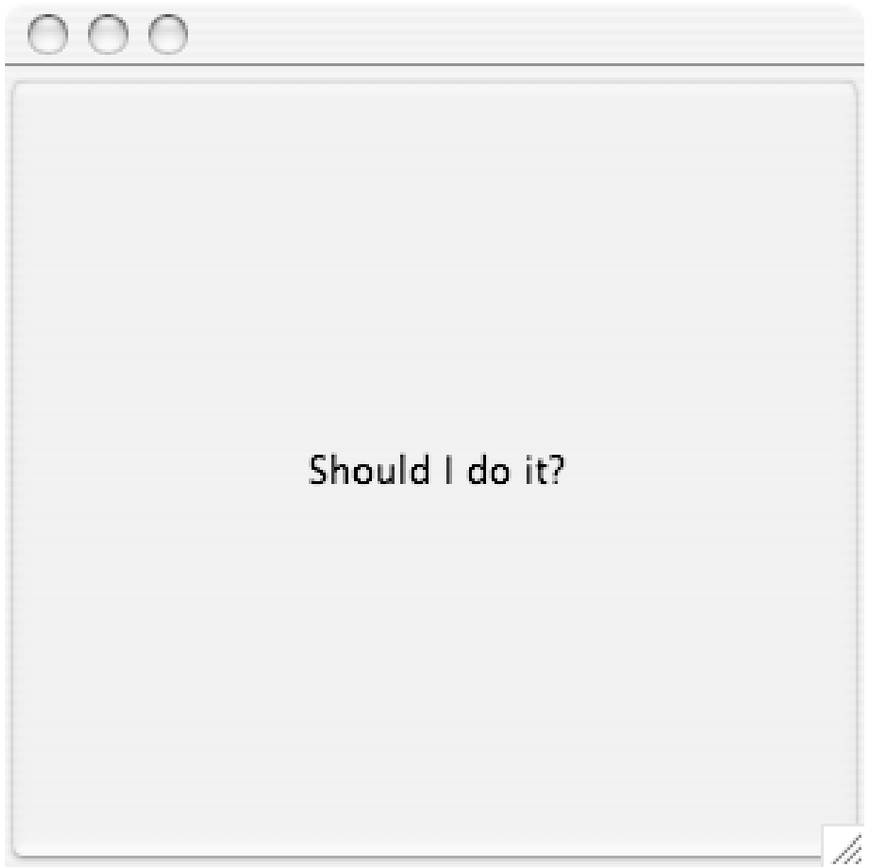
notifyObservers() only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.

Other places you'll find the Observer Pattern in the JDK

A little life-changing application

Here's our fancy interface.



And here's the output when we click on the button.

```
File  Edit  Window  Help  HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

```
public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();
        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // Set frame properties here
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Simple Swing app that just creates a frame and throws a button in it.

Makes the devil and angel objects listeners (observers) of the button

Here are the class definitions for the observers, defined as inner classes (but they don't have to be)

Rather than update(), the

Design principle challenge

How does the observer pattern uses these principles?

> ***Design Principle***
>
> *Identify the aspects of your application that vary and separate them from what stays the same.*

number and types of observers vary; observers are separated from the state of the subject

> ***Design Principle***
>
> *Program to an interface, not an implementation.*

subjects and observers know each other via interfaces

> ***Design Principle***
>
> *Favor composition over inheritance.*

observers are composed with their subject; no inheritance is used

Challenges

- Priority Observers?
- Stop propagation?

Behavioral Design Patterns

- Problem: Manage differences of behaviors in subclasses
- Approach: Avoid defining the different behaviours directly in subclasses.
- Solution: Define somewhere else and create a middleman -- Delegation

What's the key technique used by design patterns?

- Programming to the interface!!

Strategy

- Differences of subclasses: Functional capability
- Avoid defining as part of the subclass: Cannot predict the behavior of new classes.
- Use an agent instead of handling it directly: the strategy interface

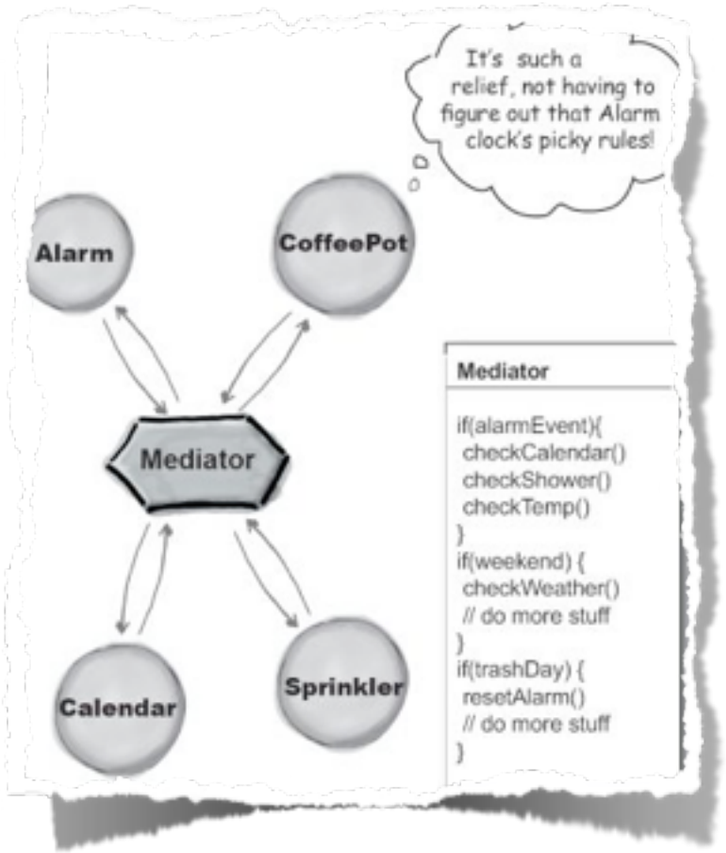
Observer

- Differences: the details of the coupling relationship vary in subclasses in a one-to-many mapping
- Avoid defining the one-to-many mapping explicitly in one place.
- Use the subject/observer interface to hide the variations of observer types.
- Use composition to manage the relationship

Half done!

Observer

vs.



Mediator

Bob's HouseOfTheFuture

(Java-enabled)

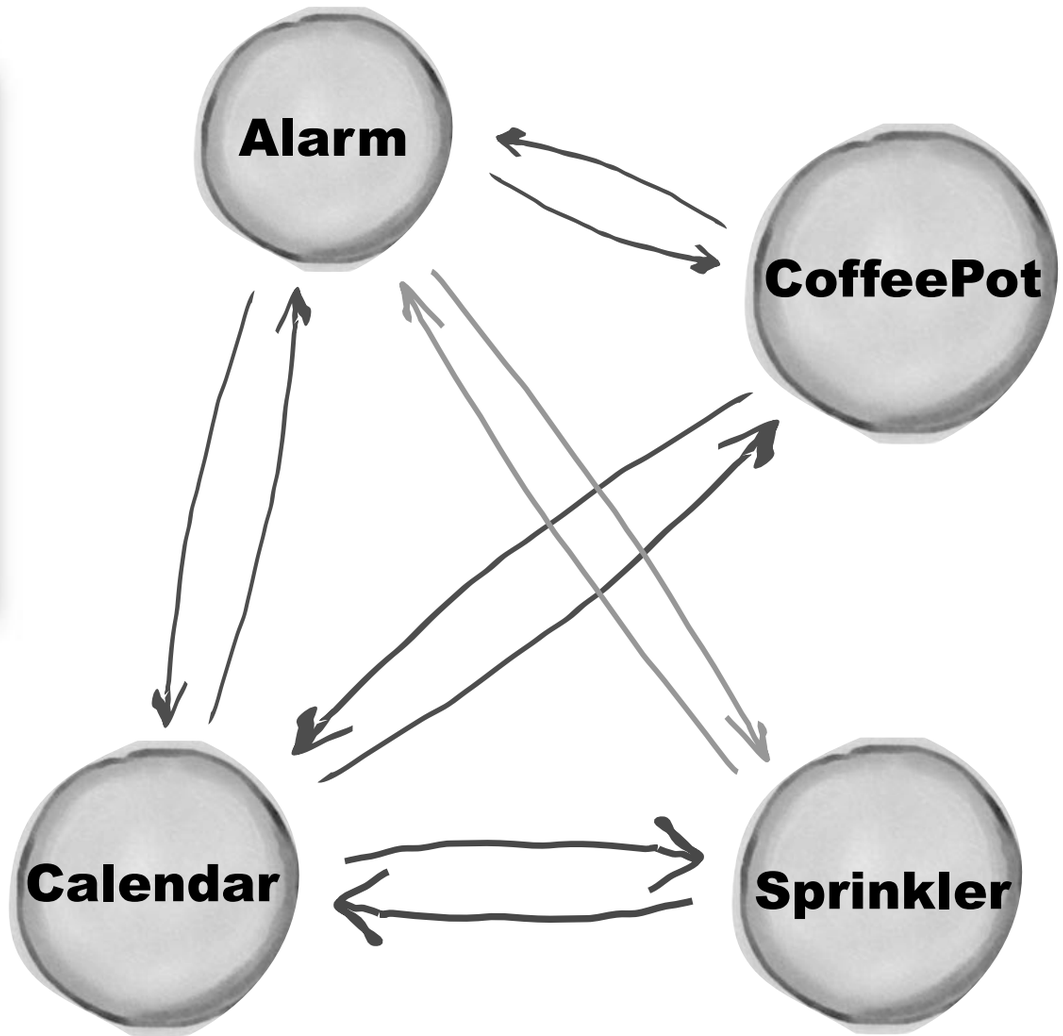
- Bob has a Java-enabled auto-house, thanks to the good folks at **HouseOfTheFuture**.
- All of his appliances are designed to make his life easier.
- When Bob stops hitting the snooze button, his alarm clock tells the coffee maker to start brewing.
- Even though life is good for Bob, he and other clients are always asking for lots of new features:
  - No coffee on the weekends...
  - Turn off the sprinkler 15 minutes before a shower is scheduled...
  - Set the alarm early on trash days...

Bob's HouseOfTheFuture

(Java-enabled)

Alarm

```
onEvent() {
  checkCalendar()
  checkSprinkler()
  startCoffee()
  // do more stuff
}
```



CoffeePot

```
onEvent() {
  checkCalendar()
  checkAlarm()
  // do more stuff
}
```

Calendar

```
onEvent() {
  checkDayOfWeek()
  doSprinkler()
  doCoffee()
  doAlarm()
  // do more stuff
}
```

Sprinkler

```
onEvent() {
  checkCalendar()
  checkShower()
  checkTemp()
  checkWeather()
  // do more stuff
}
```

Bob's HouseOfTheFuture

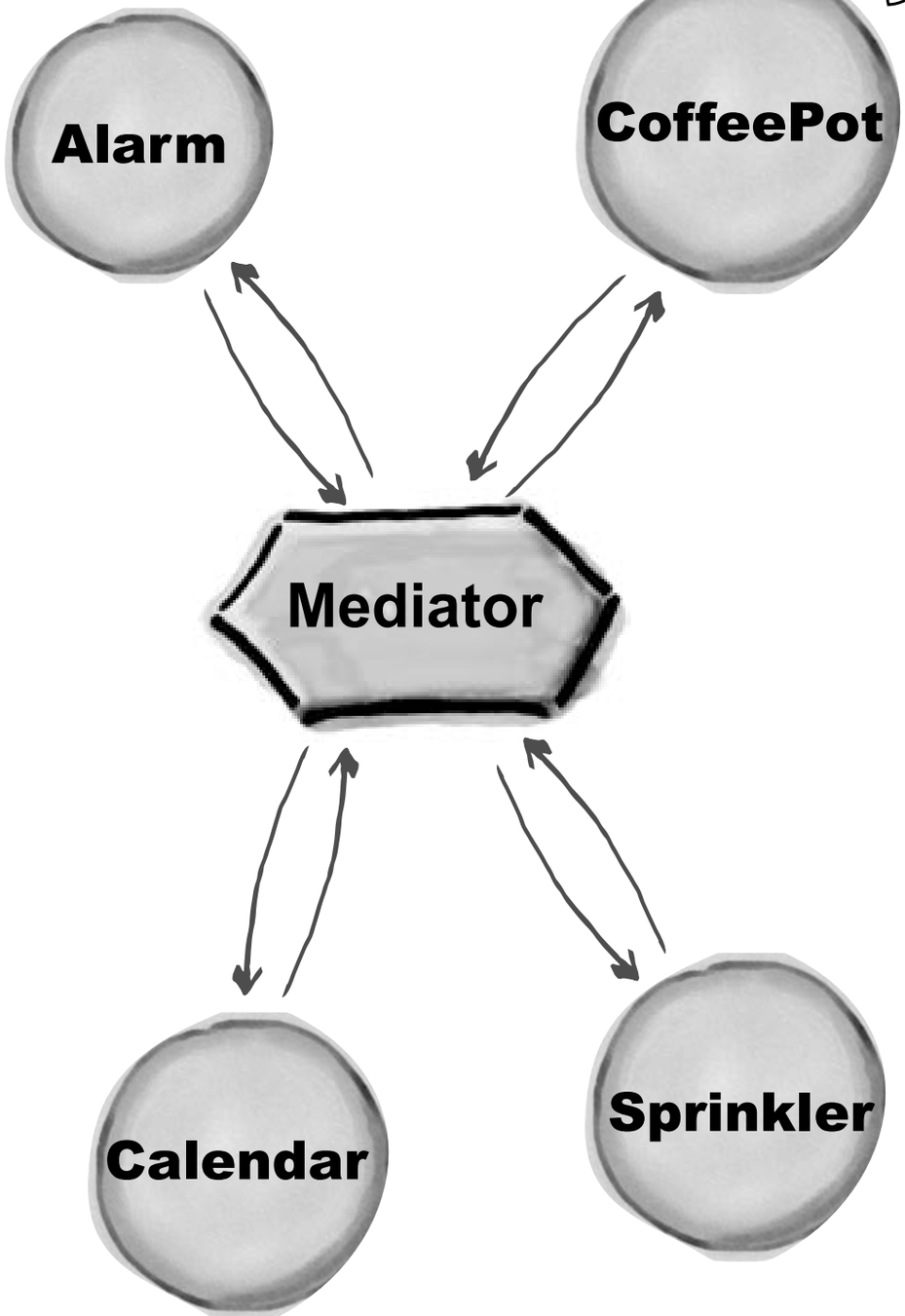
(Java-enabled)

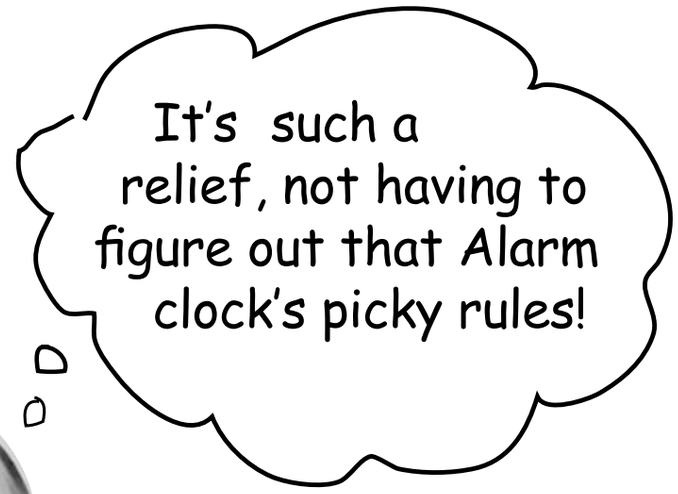
HouseOfTheFuture's dilemma

Many to Many relationships!!

- It's getting really hard to keep track of which rules reside in which objects
- how the various objects should relate to each other
- Adding new objects is challenging

Mediator pattern in action...





Mediator

```
if(alarmEvent){
 checkCalendar()
 checkShower()
 checkTemp()
}
if(weekend) {
 checkWeather()
 // do more stuff
}
if(trashDay) {
 resetAlarm()
 // do more stuff
}
```

Mediator pattern

- Intent
  - To define an object that encapsulates how a set of objects interacts,
  - avoiding having those objects explicitly know about each other

Mediator pattern

- Applicability (use when...)
  - set of objects communicate in well-defined but complex ways
  - reuse of an object is difficult because it refers to and communicates with many other objects
  - behavior is distributed between several classes and should be customizable without subclassing
- Mediator is commonly used to coordinate related GUI components

Mediator pattern

- Advantages
  - increases the reusability of objects supported by the Mediator by decoupling them
  - simplifies maintenance (centralized control logic)
  - simplifies and reduces the variety of messages sent between objects
  - many-to-many interactions become one-to-many
- Disadvantages
  - Mediator object can become overly complex

Questions!

- What parts can change easily?
  - What parts cannot?
- Observer and Mediator are different but related. How?

Summary

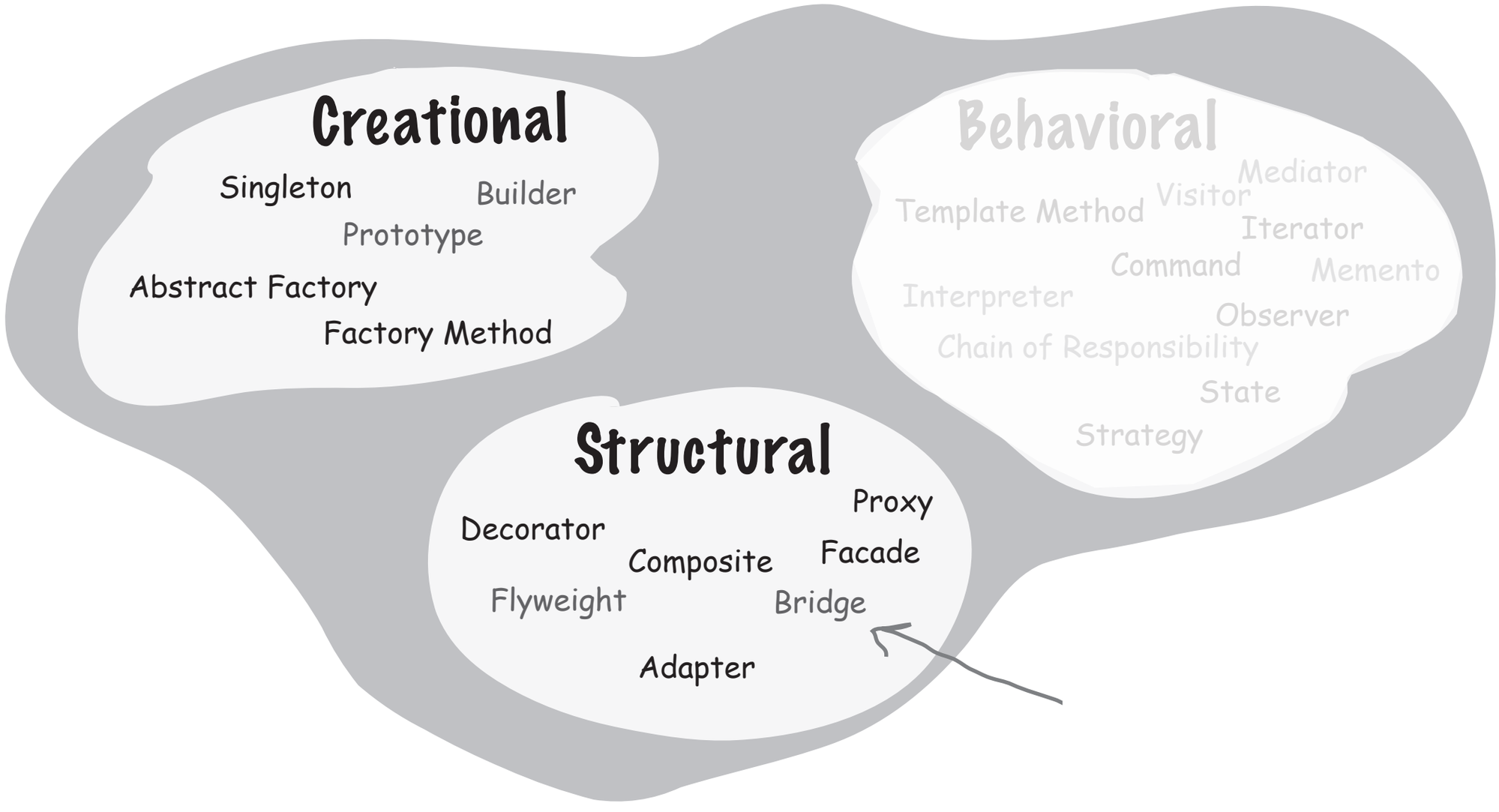
- Similarities
  - Decouple the knowledge between data source and data sinks
  - Provide the Hollywood style communication
- Differences
  - Observer pattern provides one-to-many publish/subscribe communication. Number of observers can see an event
  - Mediator pattern encapsulates communication of many-to-many in a centralized class
- Can Observer/Mediator be useful for calendar? How about time machine?

Real-World Mediators

- Databases
- Messaging services
- Content-based networks

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



Structural patterns let you compose classes or objects into larger structures.

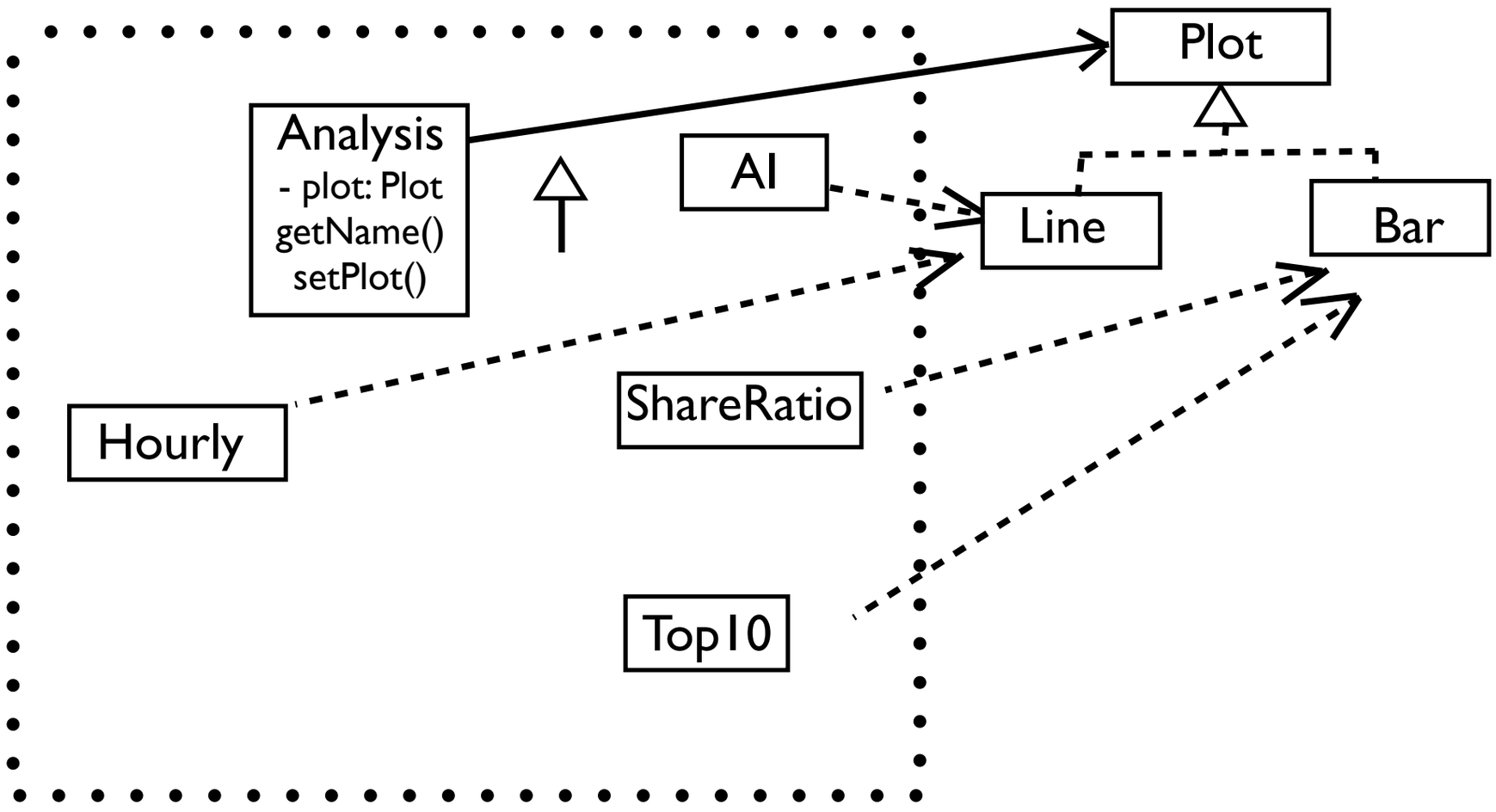
Exercise with Design Patterns

- Criticize Joe's design
- Fix Joe's design
- Give a new design for a new problem

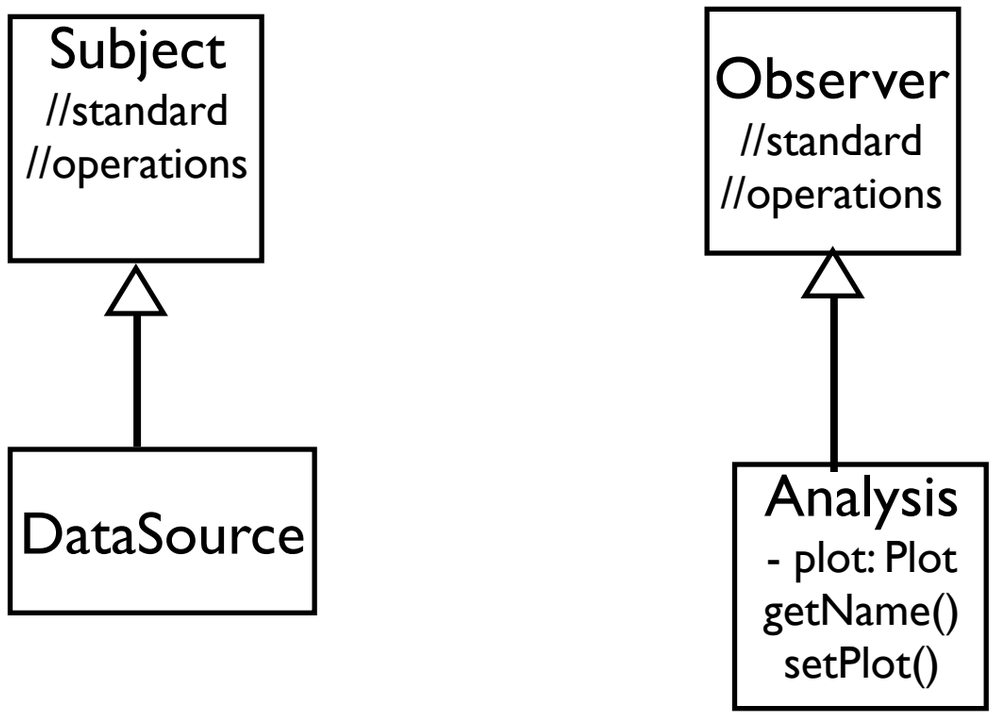
What's wrong with Joe's design?

- "plot" function is useless in some analyses.
- Bar charts and line charts plotting are duplicated
- Cannot dynamically change plots.

The Fix using Strategy



New solution



Next
Structural Patterns