Design Patterns

Barbara Russo

Definition of a Design Pattern

- "A **Pattern** describes a **problem** which occurs over and over again in our environment, and then describes the **core** of the **solution** to that problem, in such a way that you can **use this solution a million times** over, without ever doing it the same way twice"

(Alexander, Ishikawa, Silverstein, Jacobson, Fiksdhal-King, Angel "A Pattern Language", *1977*)

Reflecting …

- Here we identify
  - the problem: Building the tower
  - the solution: a suitable configuration of bricks
- The reuse of the solution ...
- The pattern is repeated … become robust

Design Patterns in Software Development

- We can translate the concept of design patterns to **software development**
- We have to define:
  - The "bricks"
  - The "configurations of the bricks"
- Object-Orientation provides a "natural way" to express design patterns

OO Design Patterns

- Design objects are our "bricks"
- Informally, a design pattern is a particular **configuration** of design objects
  - … that is, a **set of objects** and their **mutual relations** (inheritance, composition, aggregation, association, creation, …)

Design Patterns (cont'd)

- A pattern has four elements:
  1. The ***pattern name***. This is used to describe a problem, its solutions and consequences in one or two words.
  2. The ***problem***. This element describes a particular design problem and its context.
  3. The ***solution***. This describes the design elements, their relationships, their responsibilities, and collaborations.
  4. The ***consequences***. These elements are the results and trade-offs of applying design patterns.

Example

1. The ***pattern name***. Tower.
2. The ***problem***. Being visible at distance over 360º .
3. The ***solution***. Bricks are put side by side in a (circular) sequence. Circles of brick are put one over the other until to reach a predefined height
4. The ***consequences.*** *The height can be relevant if the surrounding ground is not flat*

Exercise

- Describe a pattern
  1. The ***pattern name***.
  1. The ***problem***.
  1. The ***solution***.
  1. The ***consequences***.

The reference book to read

- The "Gang of Four":
  - Erich Gamma,
  - Richard Helm,
  - Ralph Johnson,
  - John Vlissides
- Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley

The GoF Approach

- GoF distinguishes 3 kinds of patterns:
  - **Creational**: patterns dealing with object creation
  - **Structural**: patterns dealing with the composition of classes and objects
  - **Behavioral**: patterns dealing with objects interactions and sharing of responsibilities

Patterns

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibilities Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Singleton

- Goal:
  - a class has only one instance AND
  - any other instantiation **points** to the **same object**
- Use: very useful in context where we want to have only one object of a given class
  - If we want to have one and only one we can additionally make all the member data static or nesting a static class

The Singleton

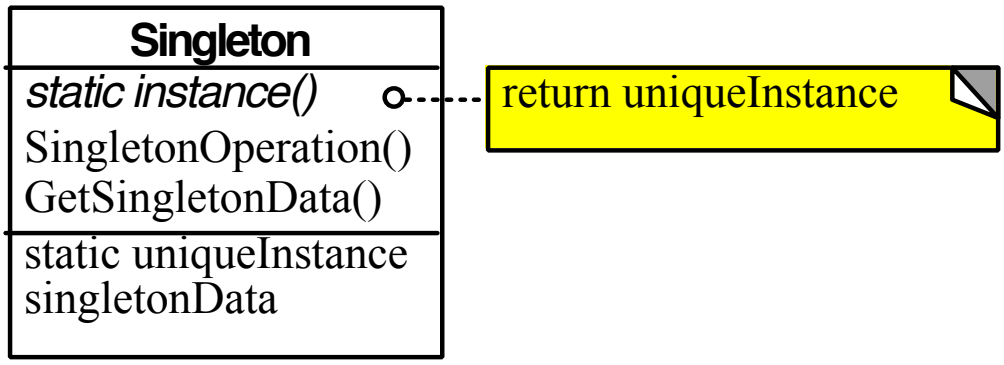
- Examples
  - Having only one file system
  - Having only one window manager
  - Having one accounting system per single company

Singleton pattern

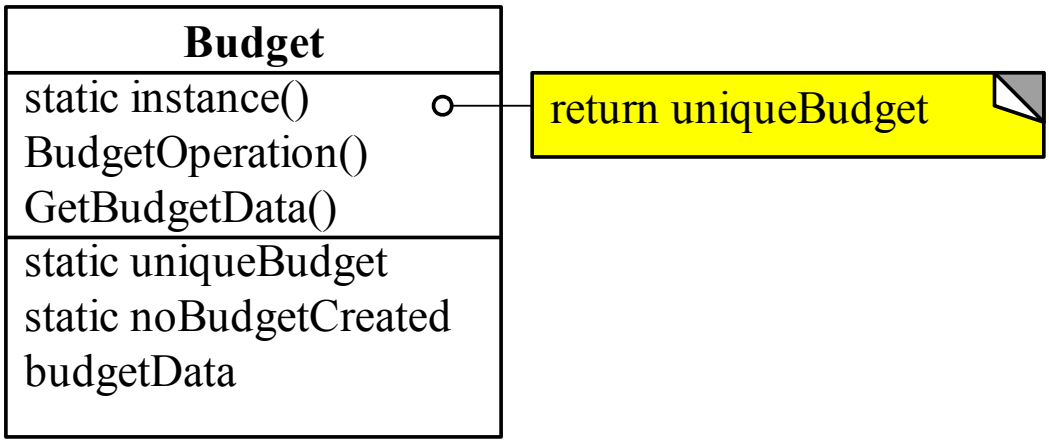
- Hiding the creation of the class instance in a **static function**
- Clients can **access** to the instance only through the function
- The constructor is either **private or protected**

The Singleton Pattern - Creational

- in Java, constructor must be private.



Uniqueness of the Budget



Java Skeleton

```
public class Budget {
  private static Budget uniqueBudget = null;
  public static Budget instance() {
    if (uniqueBudget == null)
      uniqueBudget=new Budget();
     return uniqueBudget;
    }
  }
  private Budget() { ... }
  ...
}
...

Budget townshipBudget = Budget.instance();

Budget wrongBudget = new Budget(); WRONG!!!
```


Patterns

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibilities Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

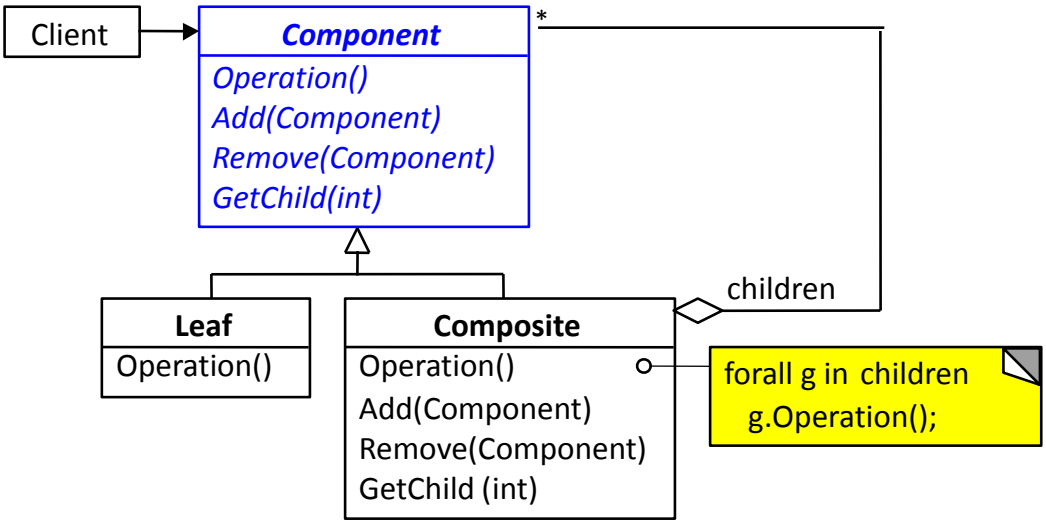
Structural Patterns

- These patterns are concerned with how **structures** are formed by the **composition** of classes and objects.
- Two types of structural patterns:
  - Structural **class** pattern which uses **inheritance** to compose interfaces or implementations.
  - Structural **object** pattern, which describes the **ways** to compose objects to realize new functionality.

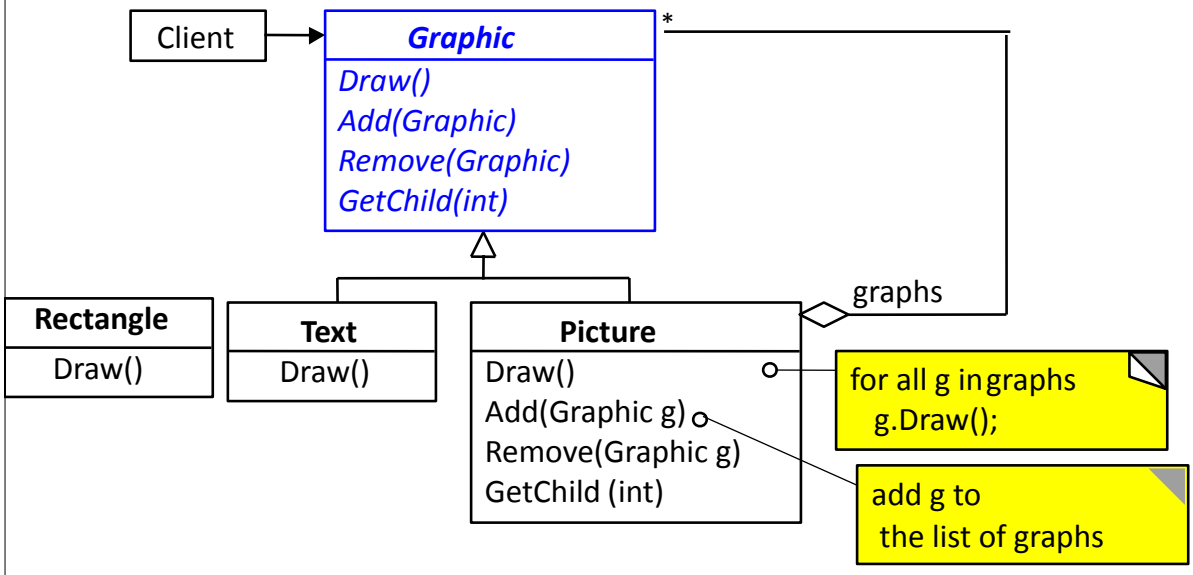
Goal and use

- Goal:
  - It structures objects and compositions in a tree structure
  - It lets clients treat individual objects and compositions in a uniform way
- Use: Very useful when we deal with objects that can be located at different levels of abstractions

The Composite Pattern

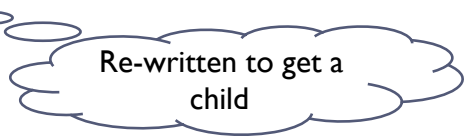


Example



Example of Component

```
class Directory {
    Directory dir;
    File[] f;
    ...
    boolean isDirectory() { return f == null; }
    boolean isFile() { return f != null; }
    File getFile(int i) {
         if (isFile()) return f[i];
         return null
    }
    Directory getDirectory() {
        if (isDirectory()) return dir;
        return null; }
.... }
```



Benefits

- Composite objects can in turn be compounded to created iteratively new complex object
- The client has a easier work as they can deal with composite and primitive objects at the same way.
- Clients do not know whether they operate with primitive or composite
- It is easier to add new objects. No existing code needs to be modified with the addition of a new object

Shortcomings

- It can make a project too generic
- It is hard to limit the expansion of the tree
- Anyone can add a new leaf or composite

Adapter (Wrapper)

- Let different classes work together when they have incompatible interfaces
- Also called Wrapper

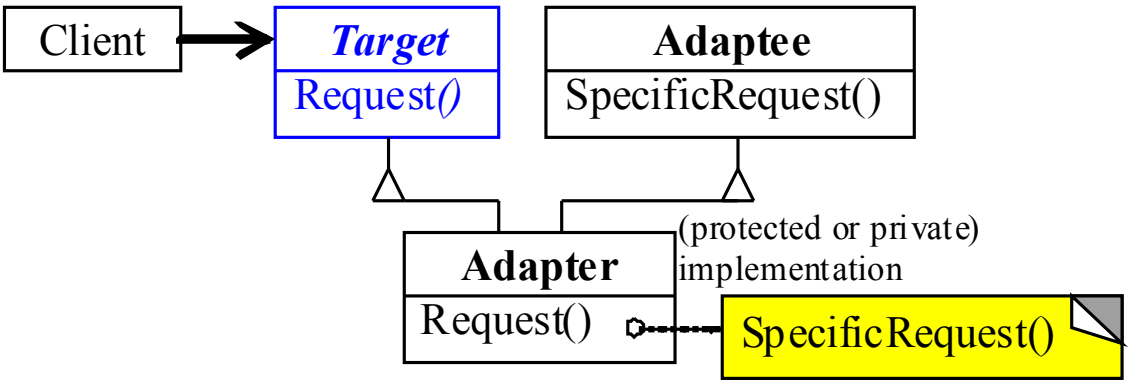
Example - Subclasses of Shape

- LineShape and Polygon shape are strightforward to implement as they are composite of few drawing objects and rules for re-shaping.
- TextShape is harder it may need complex updating of the text and memory management
- A TestView class is used to visualize TextShape,
  - The problem is that is not built for visualizing other format shapes.

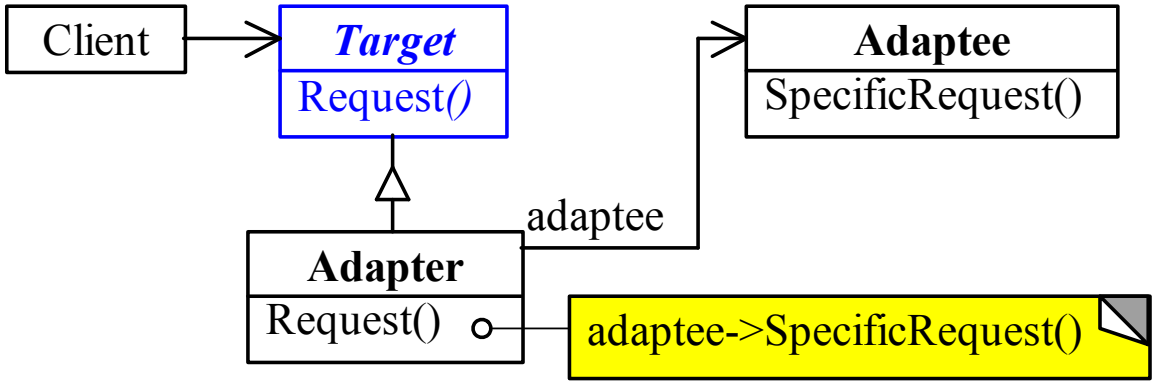
Two solutions

- Inheriting the interface of Shape and implementing TextView (**class** adapter)
- Compounding an instance of TextView in a TextShape and implementing TextShape through the interface Shape (**object** adapter)

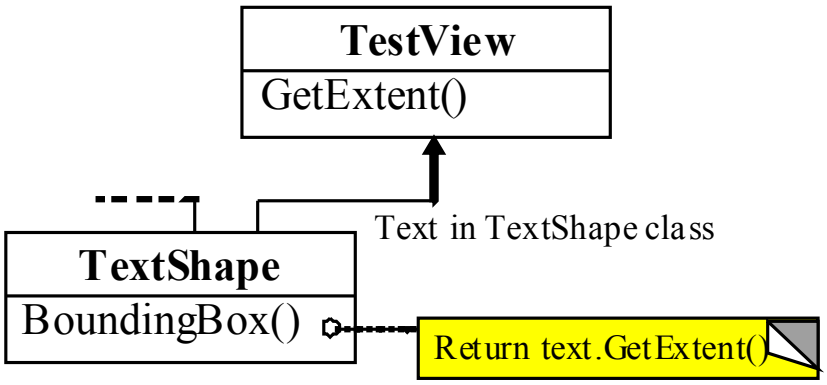
Class Adapter (with multiple inheritance)



Object Adapter (without multiple inheritance)



Example

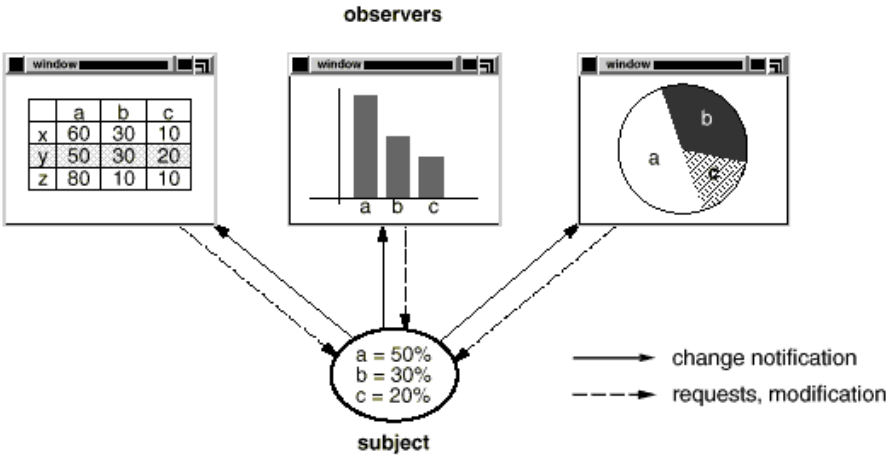


0		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibilities Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

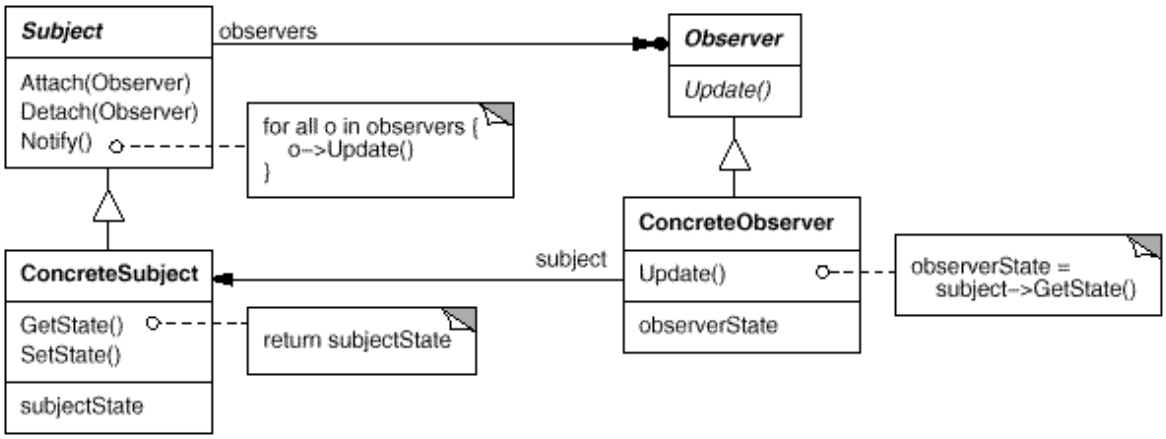
Observer (1/2)

- Intent
  - Define a 1:n dependency between objects so that when one object changes state, all its dependents **are notified** and **updated** automatically
- Also Known As
  - Dependents, Publish-Subscribe, Model-View Controller
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

Observer (2/2)



Structure



Participants (1/2)

- Subject
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- Observer
  - Defines an interface for update notification

Participants (2/2)

- ConcreteSubject
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes
- ConcreteObserver
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

Consequences (1/3)

- Benefits
  - Observers can be added without modifying the subject
  - All subjects know its list of observers
  - Subject does not need to know the concrete class of an observer

Implementation Issues (1/4)

- How does the subject keep track of its observers?
  - Array, linked list
- What if an observer wants to observe more than one subject?
  - Have the subject tell the observer who it is via the update interface

Implementation Issues (2/4)

- Make sure the subject updates its state **before** sending out notifications
- How much info about the change should the subject send to the observers?
  - Push Model – Lots
  - Pull Model - Very Little

Implementation Issues (3/4)

- Can the observers subscribe to specific events of interest?
  - If so, it's publish-subscribe
- Can an observer also be a subject?
  - Yes

Implementation Issues (4/4)

- What if an observer wants to be notified only after several subjects have changed state?
  - Use an intermediary object which acts as a mediator, ChangeManager
  - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

MVC - Model View Controller

- **Model/View/Controller** user interface framework
  - Model = Subject
  - View = Observer
  - Controller is whatever object changes the state of the subject
  - Since Java 1.1 AWT/Swing.event package

Java implementation of Observer

- Java provides the **Observable/Observer** classes as built-in support for the Observer pattern
- The java.util.Observable class is the base **Subject class**.
  - Provides methods to add/delete observers
  - Provides methods to notify all observers
  - Uses a Vector for storing the observer references
- The java.util.Observer interface is the **Observer interface**. It must be implemented by any observer class

Observable/Observer Example (1/6)

```
/**
 * A subject to observe!
 */
public class ConcreteSubject extends Observable {
  private String name;
  private float price;

  public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name
                       + " at " + price);
  }
```


Observable/Observer Example (2/6)

```
  public String getName() {
    return name;
  }
  public float getPrice() {
    return price;
  }
  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }
  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }
}
```

Observable/Observer Example (3/6)

```
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;

  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }

  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
       name = (String)arg;
       System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
  subject!");
    }
  }
}
```


Observable/Observer Example (4/6)

```
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;

  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }

  public void update(Observable obj, Object arg) {
     if (arg instanceof Float) {
       price = ((Float)arg).floatValue();
       System.out.println("PriceObserver: Price changed to " + price);
     } else {
      System.out.println(”PriceObserver: Some other change to
  subject!");
     }
  }
}
```

Observable/Observer Example (5/6)

```
// Test program for ConcreteSubject, NameObserver and
// PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops",  1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```


A Problem With Observable/Observer (1/2)

- Problem
  - Suppose the class which we want to be an observable is already part of an inheritance hierarchy:
    - `class SpecialSubject extends ParentClass`
- Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass?

A Problem With Observable/Observer (2/2)

- Solution
  - Use Delegation
    - We will have SpecialSubject contain an Observable object
    - We will delegate the observable behavior that SpecialSubject needs to this contained Observable object