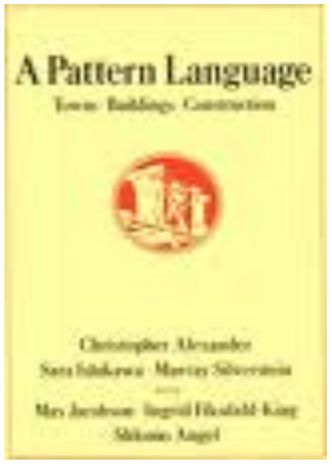
# COIS2240 Lecture 12

# Patterns originated in Architecture

- **Christopher Alexander's Philosophy:**
  - Buildings have been built for thousands of years by users who where not architects
  - Users know more about what they need from buildings and towns than an architect
  - Good buildings are based on a set of design principles that can be described with a pattern language




Christopher Alexander
* 1936 Vienna, Austria
- More 200 building projects
- Creator of the „Pattern language"
- Professor emeritus at UCB.

Although Alexanders patterns are about architecture and urban planning, they are applicable to many other disciplines, including software development.

# Design Patterns

- Design Patterns are the foundation for all SE patterns
  - Based on Christopher Alexander‘s patterns
- Book by John Vlissedes, Erich Gamma, Ralph Johnson and Richard Helm, also called the Gang of Four
  - Idea for the book at a BOF "Towards an Architecture Handbook“ (Bruce Anderson at OOPSLA’90)


John Vlissedes
•* 1961-2005
•Stanford
•IBM Watson Research Center

Erich Gamma
•* 1961
•ETH
•Taligent, IBM
• JUnit, Eclipse,
• Jazz

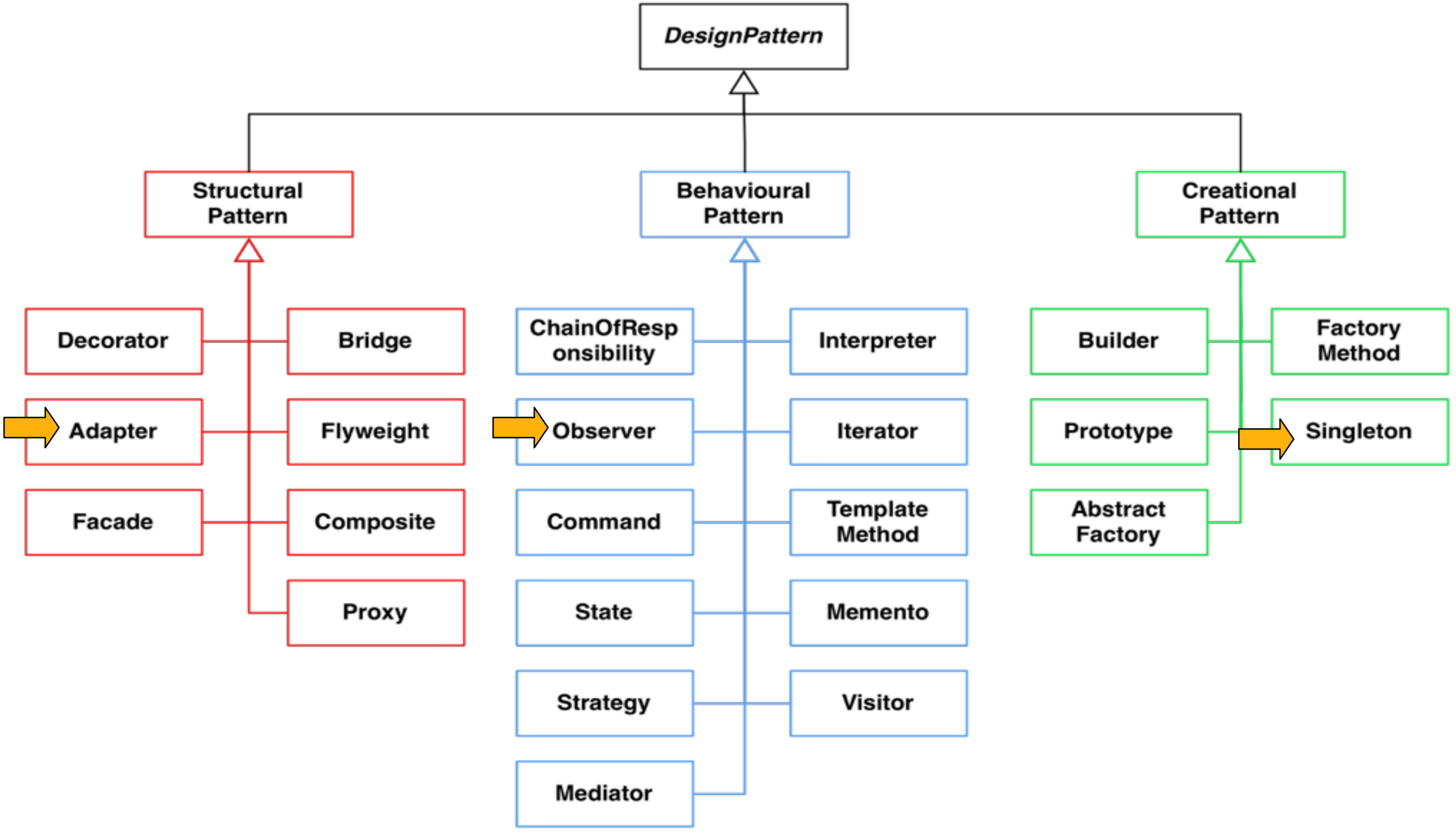
Ralph Johnson
•* 1955
•University of Illinois,
•Smalltalk, Design Patterns, Frameworks, OOPSLA veteran

Richard Helm
• University of Melbourne
•IBM Research, Boston Consulting Group (Australia)
•Design Patterns

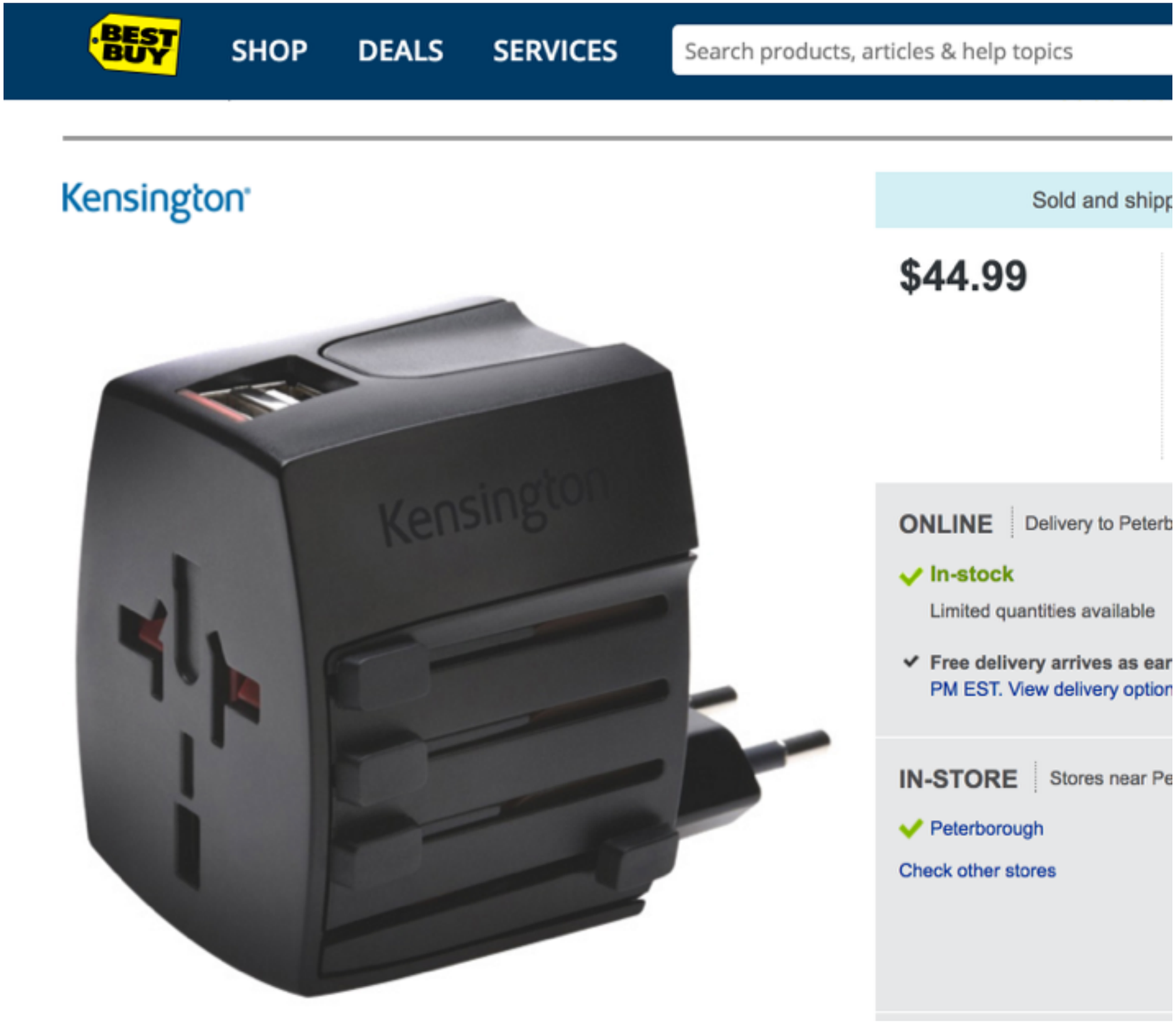
# 3 Types of Design Patterns (GoF Patterns)

- Structural Patterns
  - Reduce coupling between two or more classes
  - Introduce an abstract class to enable future extensions
  - Encapsulate complex structures
  - Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Behavioural Patterns
  - Characterize complex control flows that are difficult to follow at runtime.
  - Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Creational Patterns
  - They abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.
  - Make the system independent from the way its objects are created, composed and represented.

# Taxonomy of Design Patterns



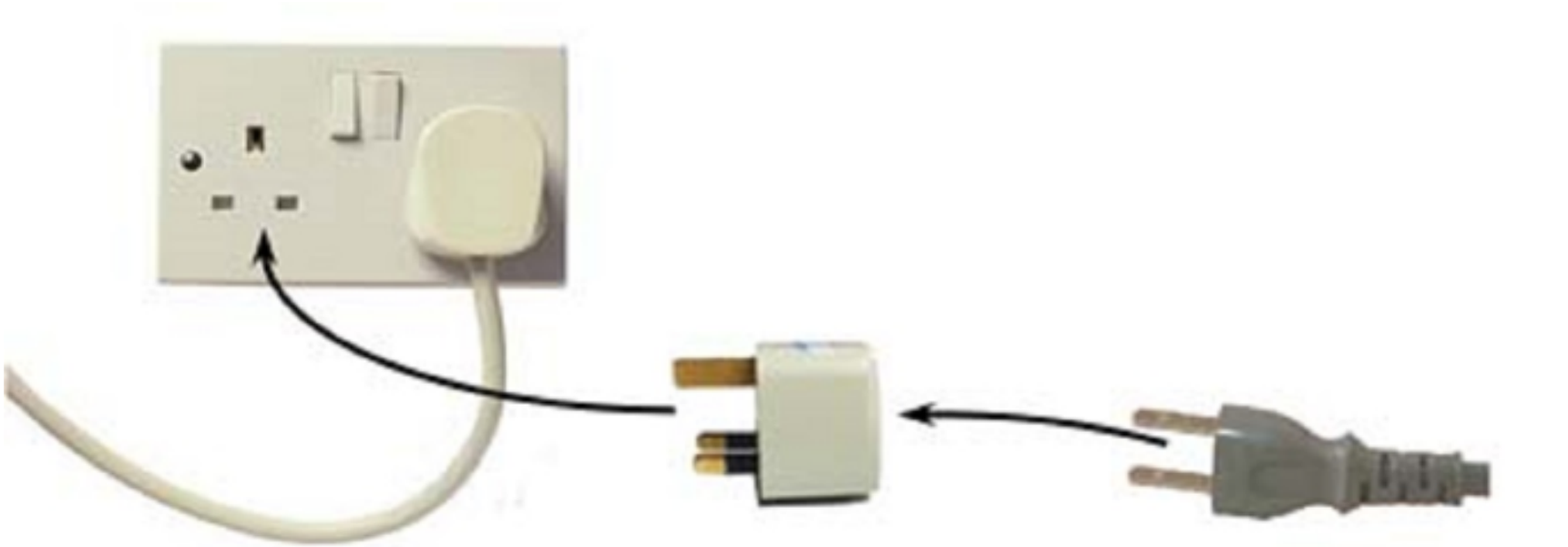
# Adapter Pattern .



# Adapter Pattern .

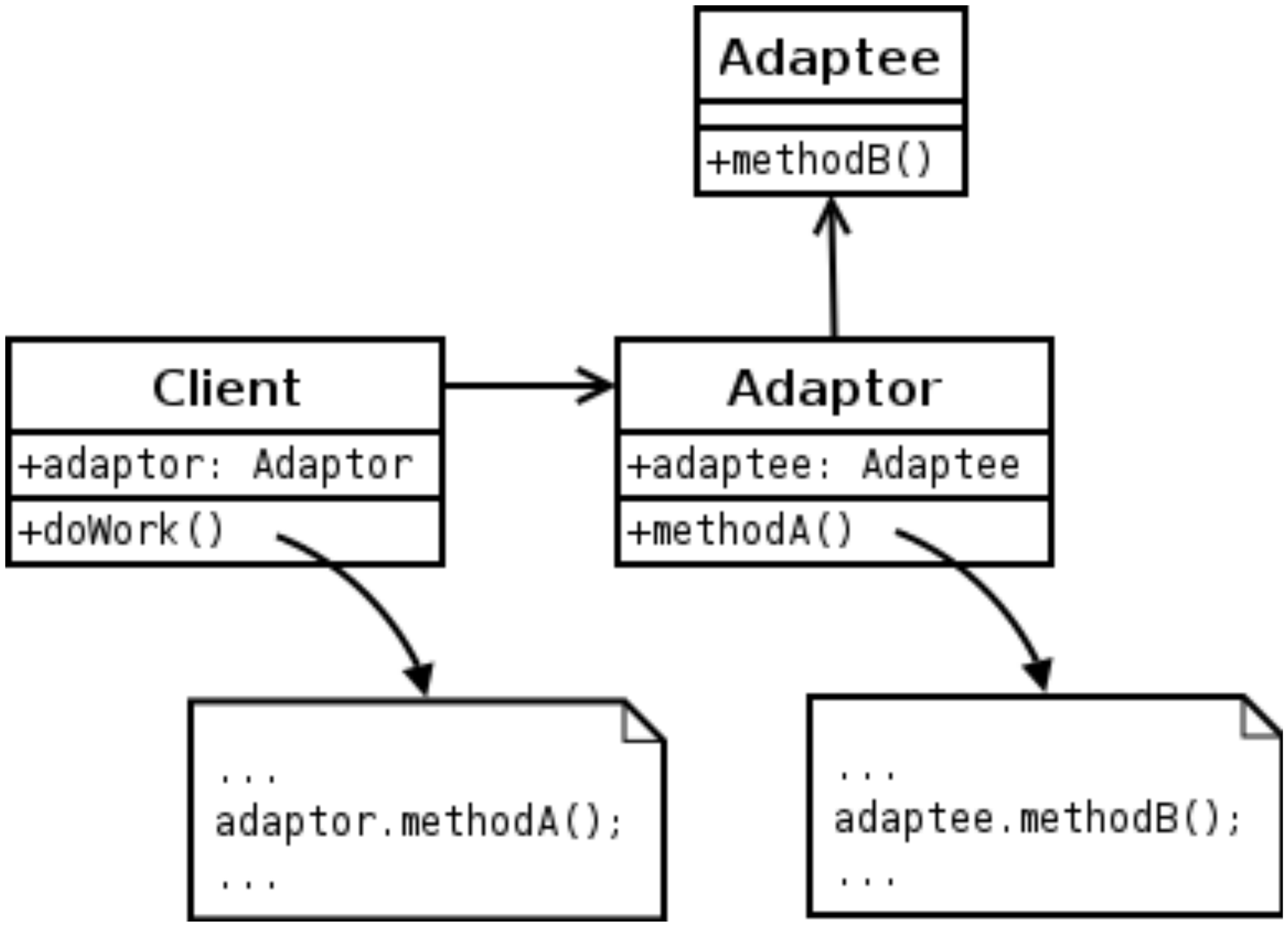
- Adapter Pattern: Connects incompatible components
  - It converts the interface of one component into another interface expected by the other (calling) component
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering)
- Also known as a wrapper.

# Adapter Pattern .



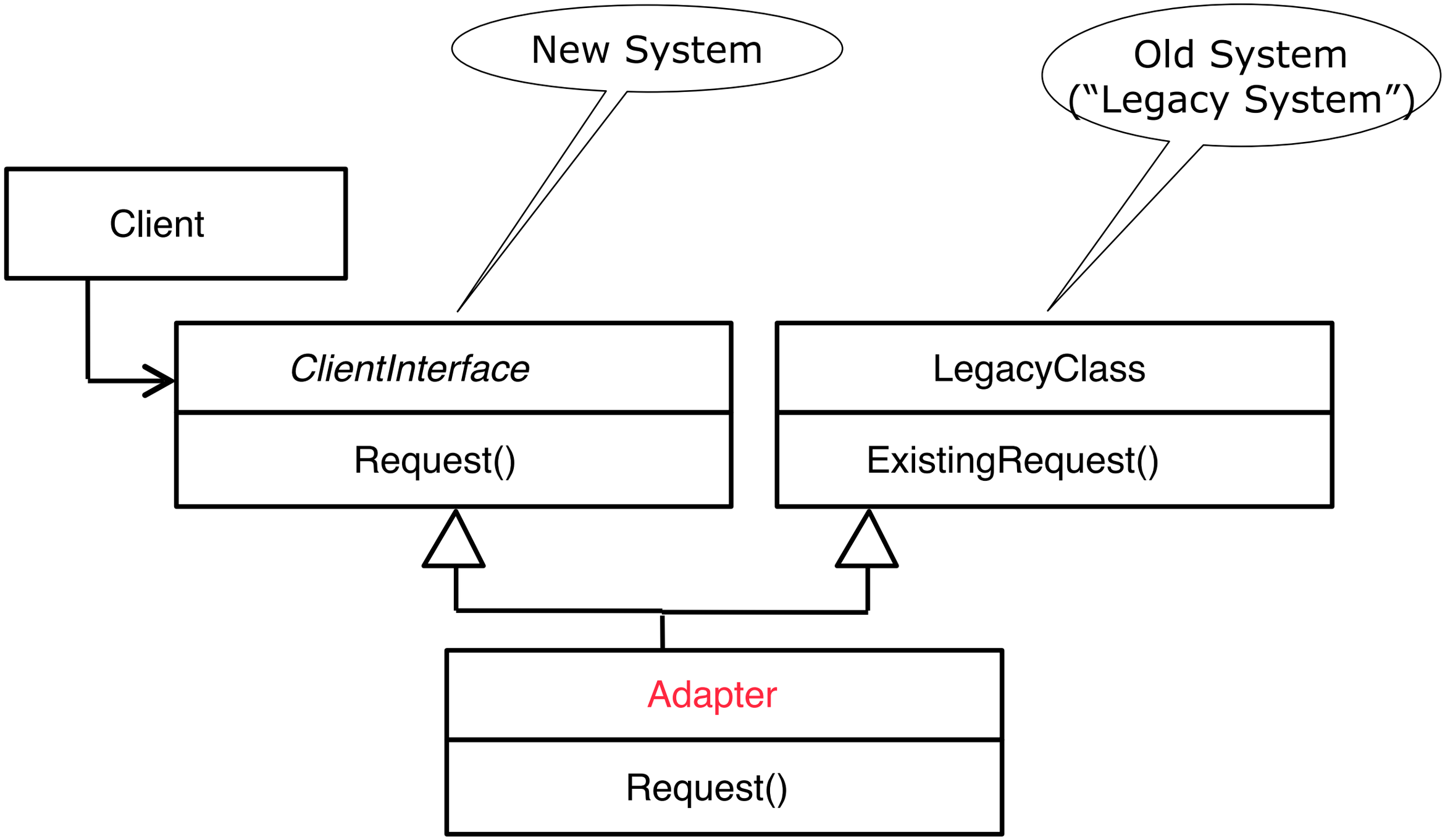

source: https://1.bp.blogspot.com/-CGxalcUP5bg/V8WRA5WCMsI/AAAAAAAAG5o/mxCloKk3jM4t5mTNU9LLo8GVa3RpQiikQCLcB/s1600/Adapter%2Bdesign%2Bpattern%2Bin%2Bwork%2BJava.jpg

# Adapter Pattern

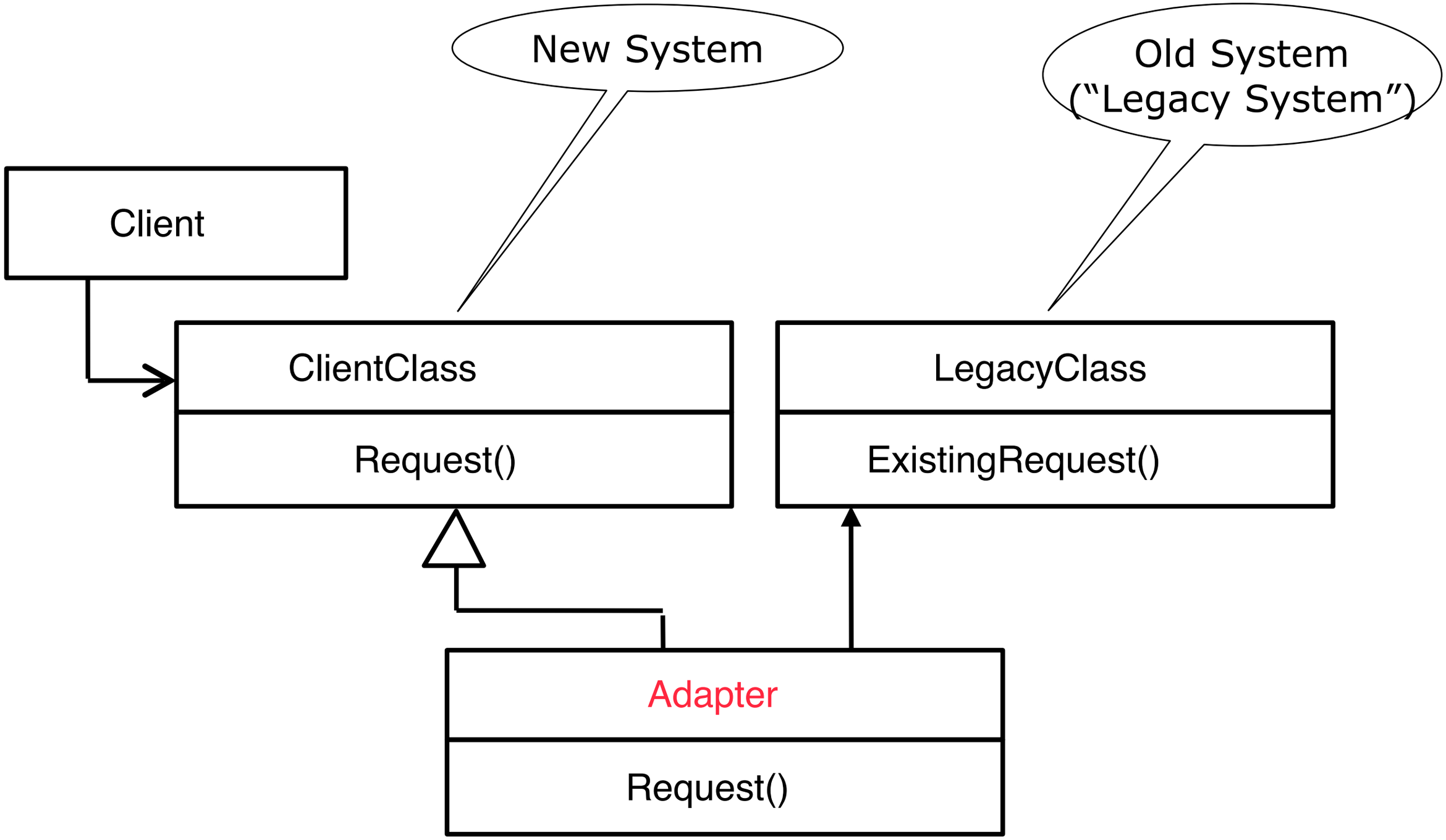



source: https://upload.wikimedia.org/wikipedia/commons/d/d7/ObjectAdapter.png

# Class Adapter Pattern



# Object Adapter Pattern



# How does it look like in code? hmm.. (Class Adapter)

```
class LegacyRectangle {
    public double drawRectangle(int x, int y, int height, int width) {
            ……
    }
}
```

```
interface ClientInterface {
    void drawRec(int xTopLeft, int yTopLeft, xBottomRight, yBottomRight);
}

class MyNewClassAdapter extends LegacyRectangle implements ClientInterface {
    void drawRec(int xTopLeft, int yTopLeft, xBottomRight, yBottomRight) {
        // do stuff to calculate the height and the width
        drawRectangle(int x, int y, int height, int width);
    }
}
```

# How does it look like in code? hmm.. (Object Adapter)

```
class LegacyRectangle {
    public double drawRectangle(int x, int y, int height, int width) {
            ……
    }
}
```

```
abstract class Client {
    void drawRec(int xTopLeft, int yTopLeft, xBottomRight, yBottomRight);
}

class MyNewClassAdapter extends Client{
   LegacyRectangle legrec;
    void drawRec(int xTopLeft, int yTopLeft, xBottomRight, yBottomRight) {
        // do stuff to calculate the height and the width
        legrec.drawRectangle(int x, int y, int height, int width);
    }
}
```

# Singleton

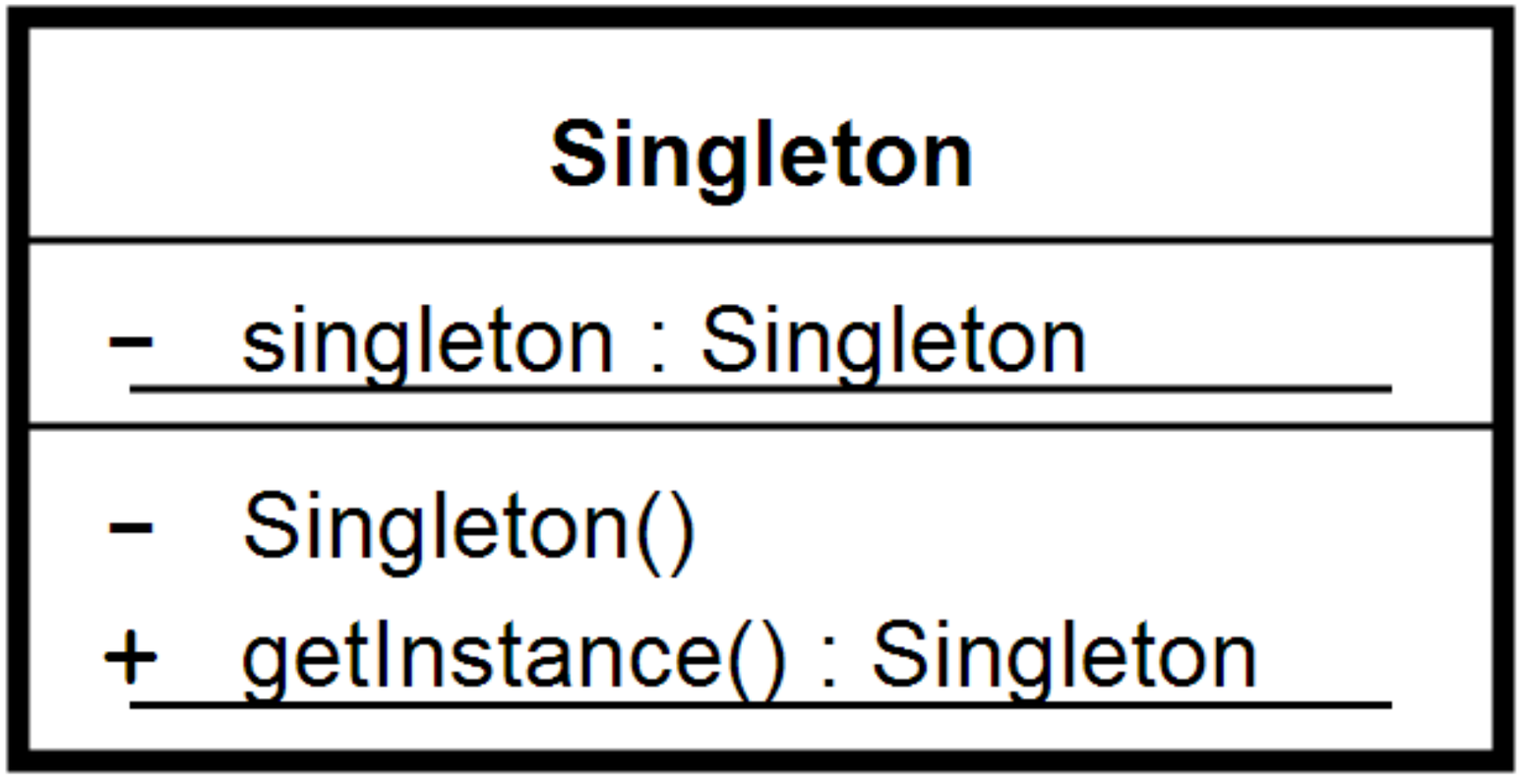
- It's important for some classes to have exactly one instance.
- More than one instance will result in incorrect program behaviour
- More than one instance will result in the overuse of resources
- More than one instance will result in inconsistent results
- There is a need for a global point of access

# Singleton

- Example: There must be one instance of the printer spooler to accessed by all clients.
- This usually happens when you want to share a global resource.
- The singleton pattern ensures that there is only one point of entry and only one instance is created.

# Singleton

- How to do that?



# Singleton

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

# Singleton —Eager initialization

static private data element

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

private constructor

public static getter

# Singleton—Lazy instantiation

initialize with null

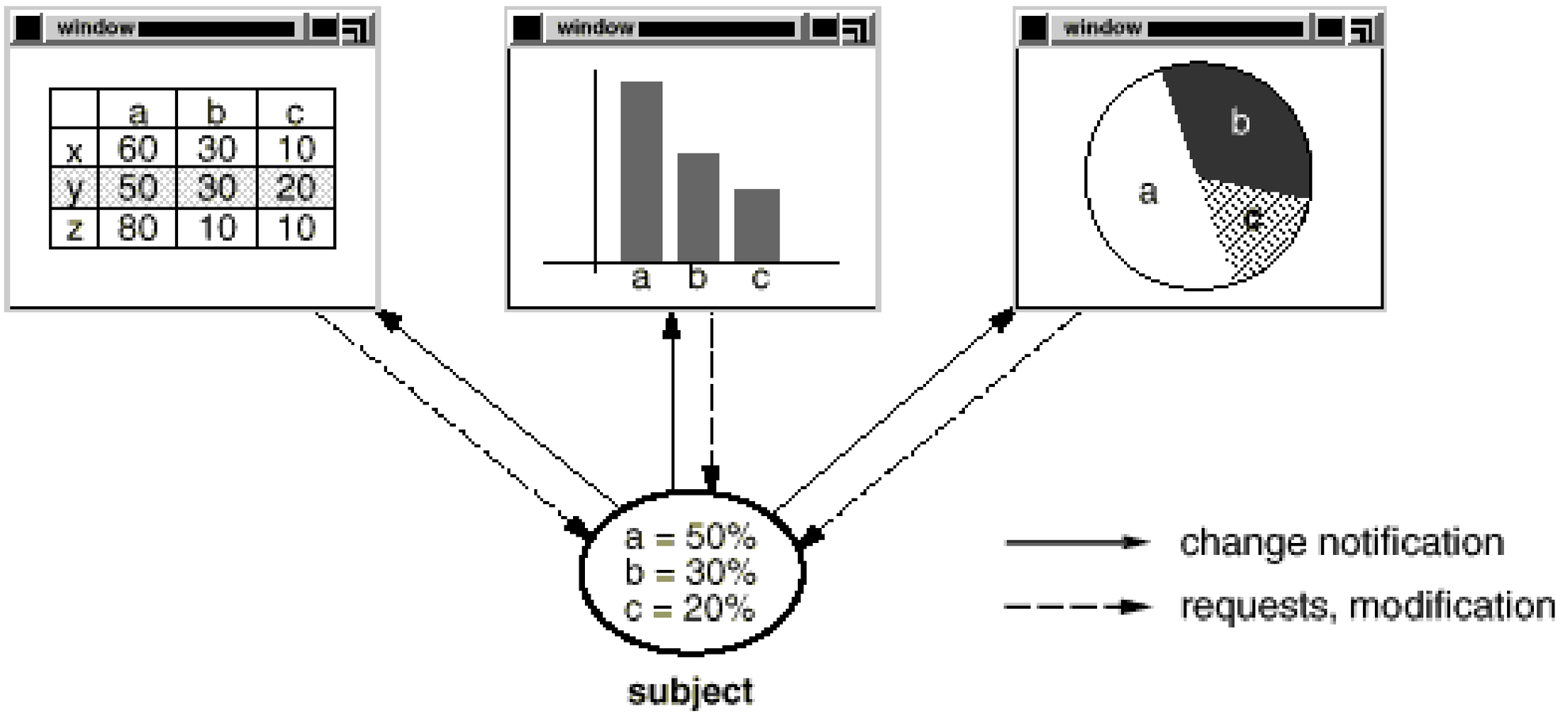
```
public final class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
                instance = new Singleton();
        }
        return instance;
    }
}
```

lazy instantiation

# Observer

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

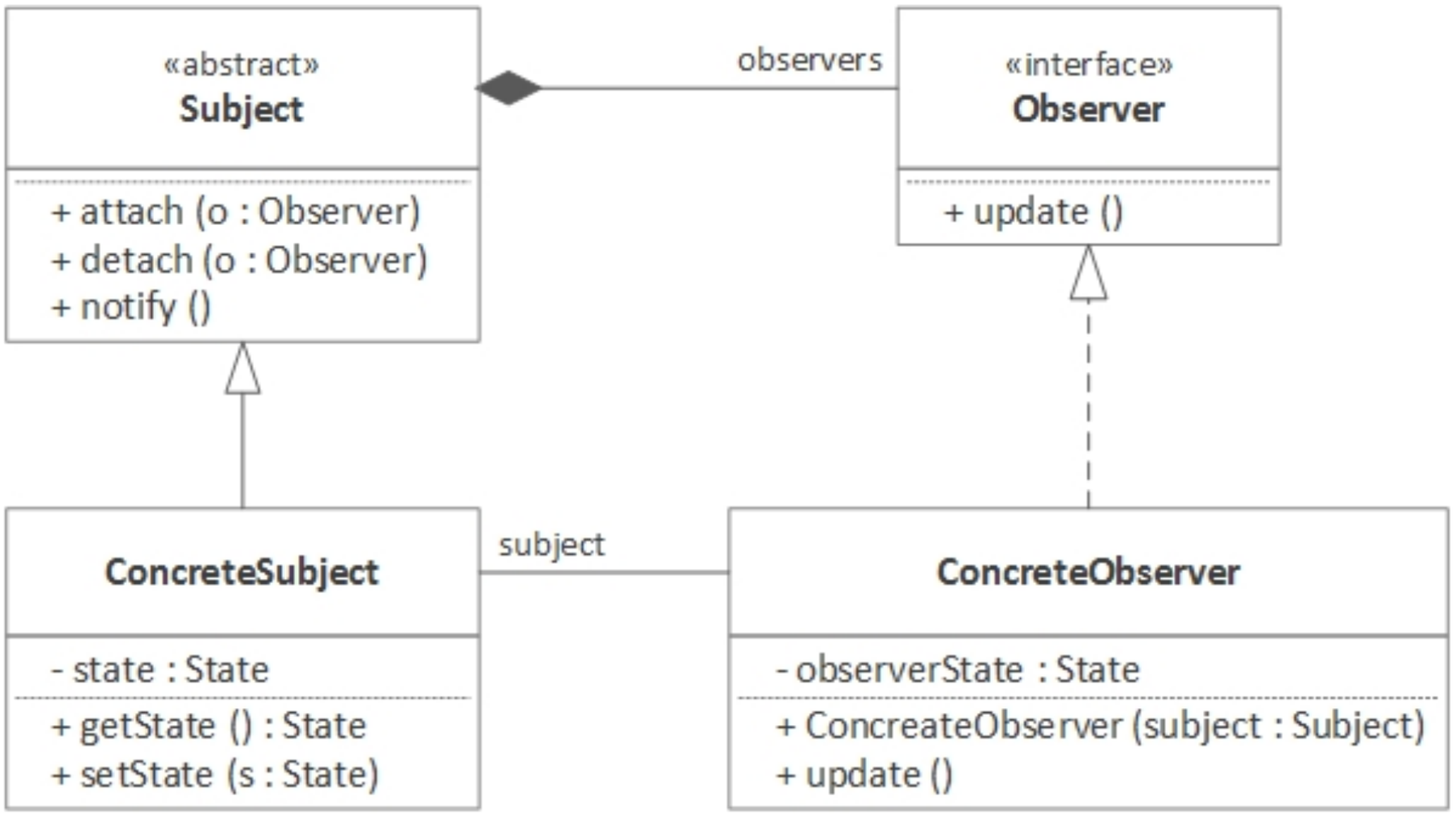
# Observer


window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

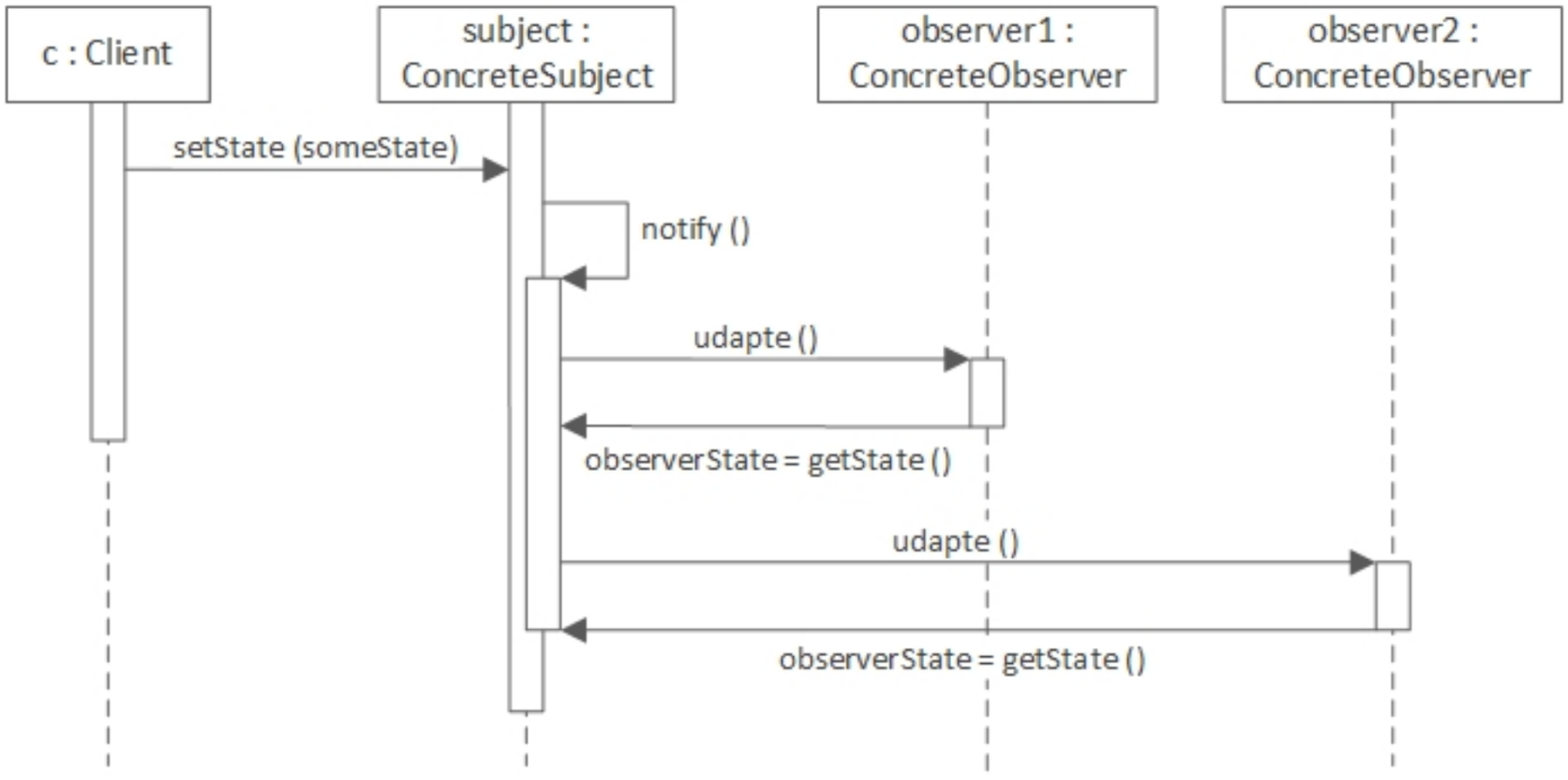
# Observer

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

# Observer


«abstract»
Subject
+ attach (o : Observer)
+ detach (o : Observer)
+ notify ()
observers
«interface»
Observer
+ update ()
ConcreteSubject
- state : State
+ getState () : State
+ setState (s : State)
subject
ConcreteObserver
- observerState : State
+ ConcreateObserver (subject : Subject)
+ update ()

# Observer


c : Client
subject : ConcreteSubject
observer1 : ConcreteObserver
observer2 : ConcreteObserver
setState (someState)
notify ()
udapte ()
observerState = getState ()
udapte ()
observerState = getState ()

# Observer

```
abstract class Observer {
protected Subject subj;
  public abstract void update();
}

class HexObserver extends Observer {
  public HexObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }
  public void update() {
  System.out.print( " " +
Integer.toHexString( subj.getState() ) );
  }
}
```

# Observer

```
class BinObserver extends Observer {
  public BinObserver( Subject s ) {
    subj = s;
    subj.attach( this ); } // Observers register
themselves
    public void update() {
    System.out.print( " " +
Integer.toBinaryString( subj.getState() ) );
  }
}
```

# Observer

```
class Subject {
  private Observer[] observers = new Observer[9];
  private int totalObs = 0;
  private int state;
  public void attach( Observer o ) {
    observers[totalObs++] = o;
  }

  public int getState() {
    return state;
  }

  public void setState( int in ) {
    state = in;
    notify();
  }

  private void notify() {
    for (int i=0; i < totalObs; i++) {
      observers[i].update();
    }
  }
}
```

# Observer

```
public class ObserverDemo {
  public static void main( String[] args ) {
    Subject sub = new Subject();
    // Client configures the number and type of Observers
    new HexObserver( sub );
    new BinObserver( sub );
    Scanner scan = new Scanner();
    while (true) {
      System.out.print( "\nEnter a number: " );
      sub.setState( scan.nextInt() );
    }
  }
}
```