G52APR

Applications Programming

# Introduction to Patterns

*Colin Higgins*

# 0. Design Patterns

- What are design patterns?
  - Design patterns are language-independent strategies for solving common problems.
- Specifically here..
  - Design patterns are language-independent strategies for solving common *object-oriented design problems*.

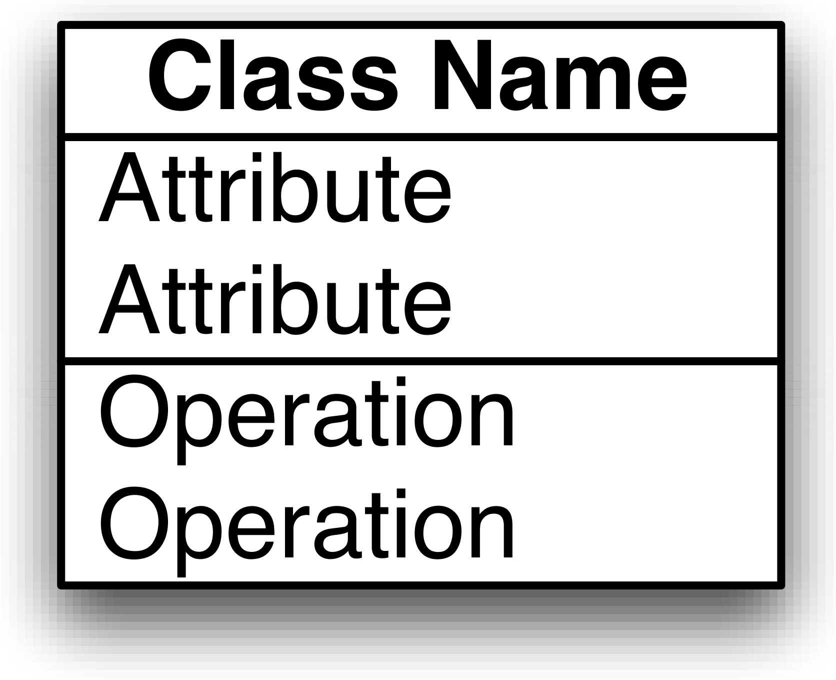
# Design Patterns

- How many design patterns?
  - Many.
- Why use design patterns?
  - Solutions to complex problems
  - Code reuse
  - To be a good Java developer
- Knowledge of them means:
  - Shared language
  - Solution at hand

# Unified Modeling Language

- UML
- Not going to cover it in detail in this module,
- except it provides a notation for us to draw diagrams of classes

# UML representation for a class



Class Name

Attribute

Attribute

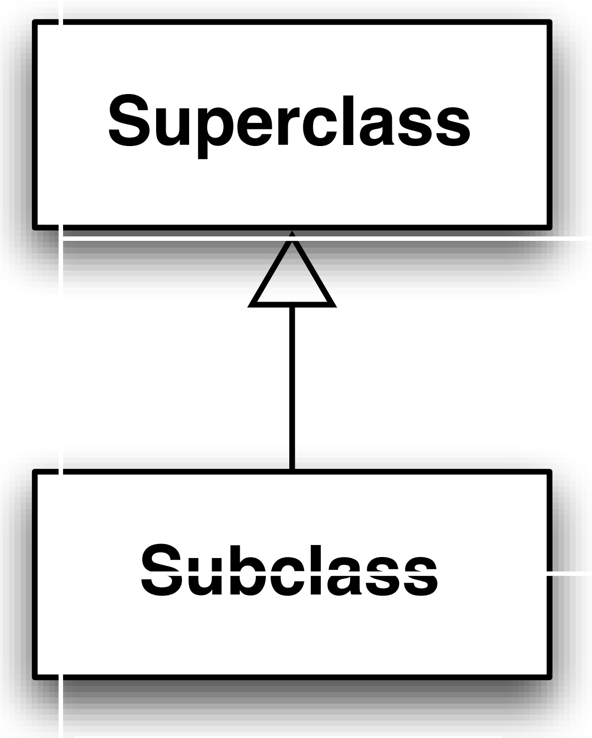
Operation

Operation

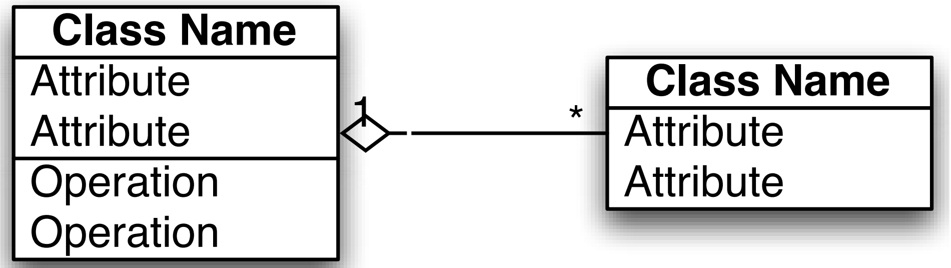
# UML Notation

- Association — line between two classes
- Aggregation — line with diamond
- Composition — line with filled diamond
- Inheritance — line with arrow
- Can add notes to say one-to-many etc

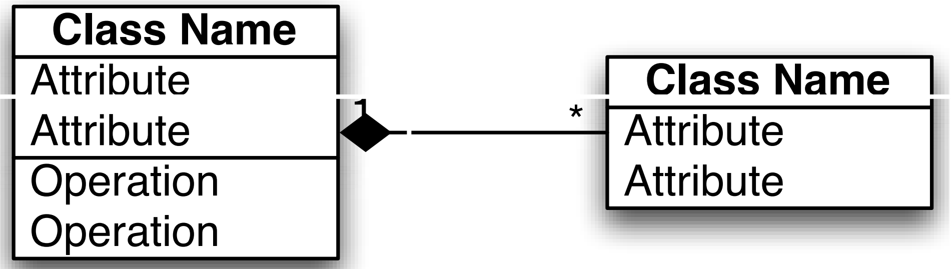
# UML Class Diagram



Inheritance



Aggregation



Composition

# 1. Singleton Pattern

- Sometimes we only want a single occurrence of an object.
- We could politely ask programmers not to create more than one.
- Better is to enforce this (defensive programming?)
- How do we enforce this?

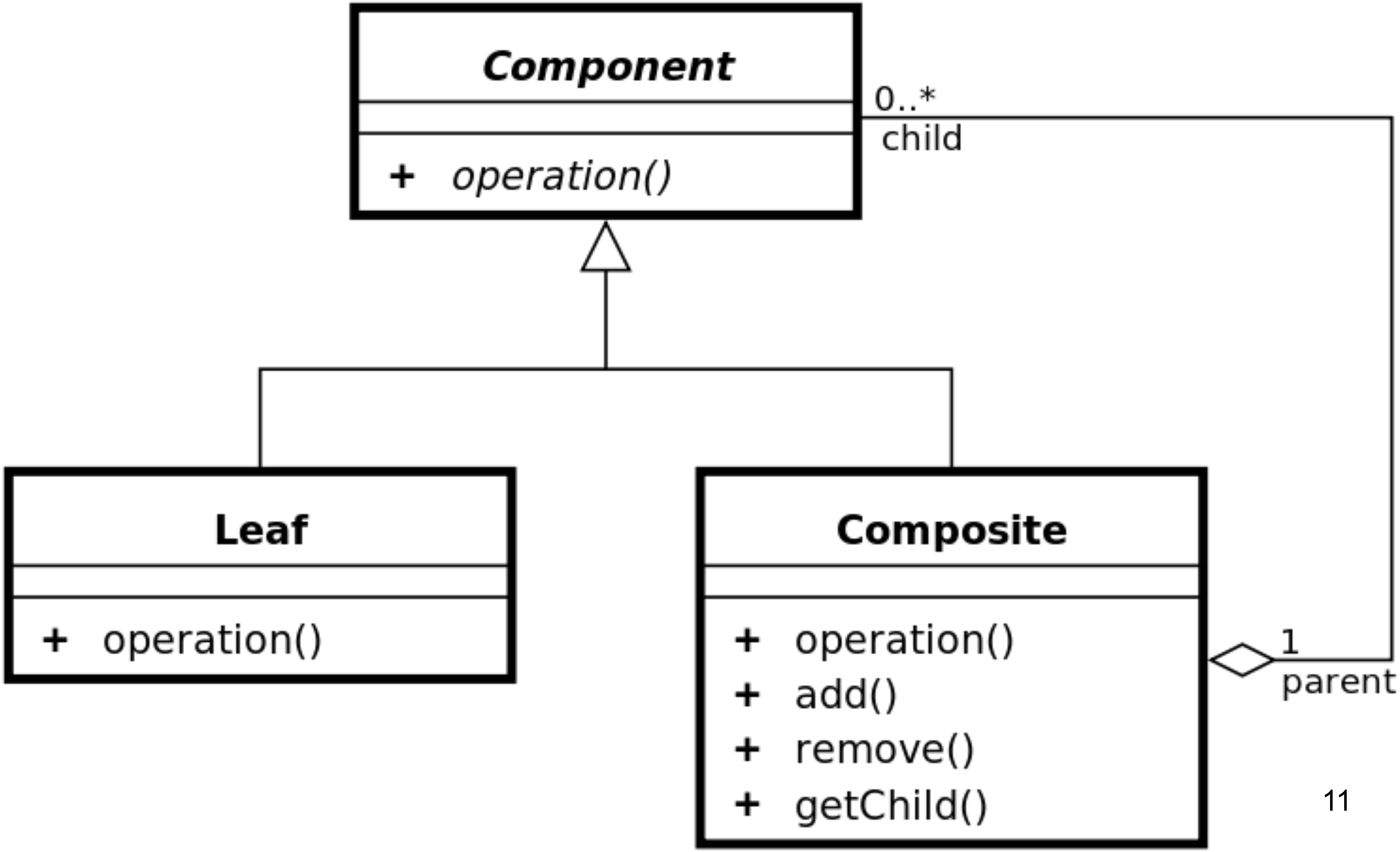
# 2. Composite Pattern

- ***The Composite Pattern*** *allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual and collections of objects uniformly.*

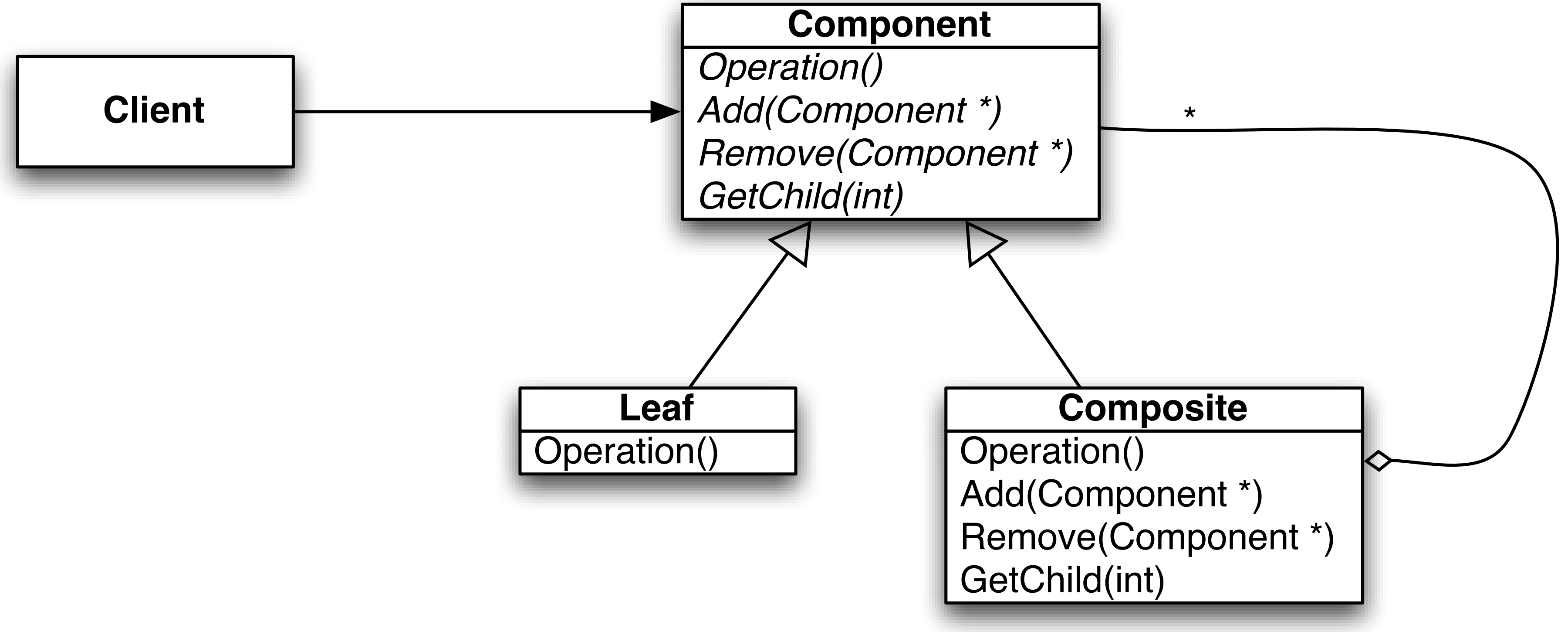
# Composite Terminology

- Client — Code that manipulates the object in composition uses the…
- Component — interface for all objects in the composition
- Composite — component with children
- Leaf — primitive component

# Composite Pattern



# Alternative Composite UML diagram



# Composite Notes

# Composite Example

```
/** Client */
public class Program {

   public static void main(String[] args) {
      //Initialize four ellipses
      Ellipse ellipse1 = new Ellipse();
      Ellipse ellipse2 = new Ellipse();
      Ellipse ellipse3 = new Ellipse();
      Ellipse ellipse4 = new Ellipse();

      //Initialize three composite graphics
      CompositeGraphic graphic = new CompositeGraphic();
      CompositeGraphic graphic1 = new CompositeGraphic();
      CompositeGraphic graphic2 = new CompositeGraphic();

      //Composes the graphics
      graphic1.add(ellipse1);
      graphic1.add(ellipse2);
      graphic1.add(ellipse3);

      graphic2.add(ellipse4);

      graphic.add(graphic1);
      graphic.add(graphic2);

      //Prints the complete graphic (four times the string "Ellipse").
      graphic.print();
   }
}
```

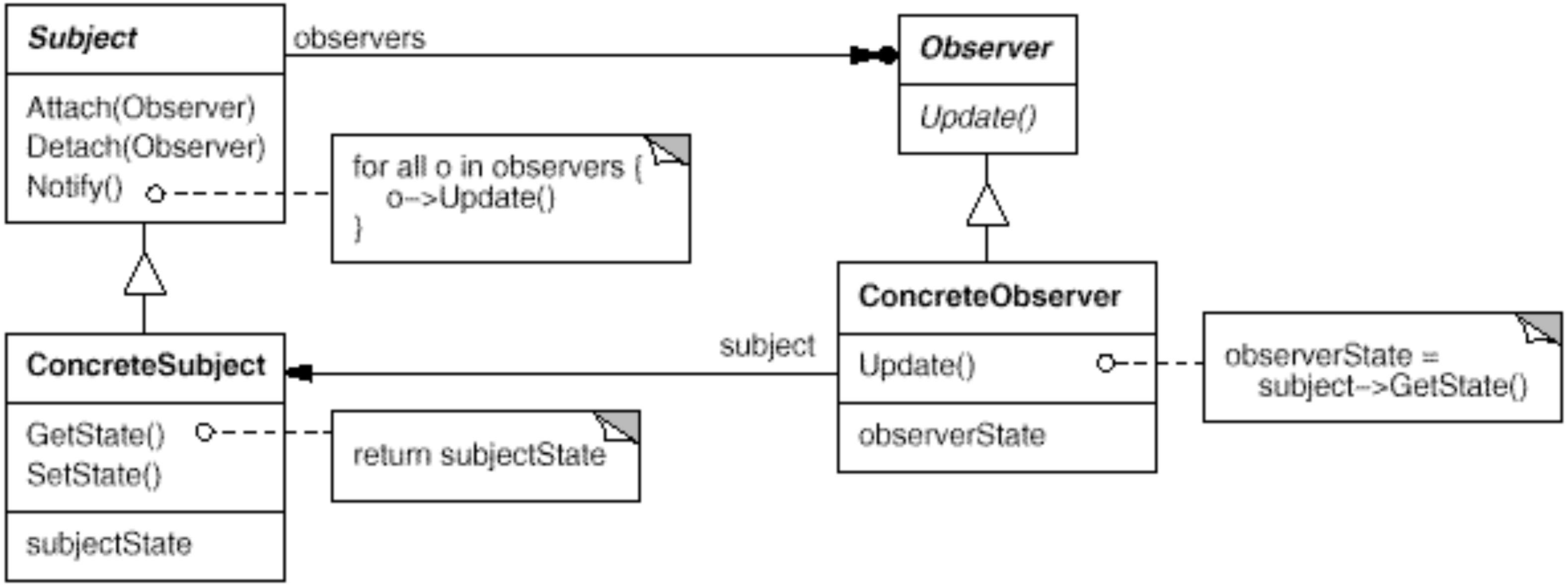
```
/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}

/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
```

```
/** "Composite" */
import java.util.List;
import java.util.ArrayList;
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}
```

# 3 Observer Pattern

- Motivation:

# Observer Pattern UML diagram



# Observer Pattern Example

- This is the Observer and Singleton Pattern from class
  - (Note code is not optimally formatted so it fits on one slide per class)

# Application.java

```
public class Application {

        public static void main(String[] args) {
                DataSource ds = DataSource.getDataSource();
                View v = new View("alpha");

                ds.addObserver(v);
                ds.setData("first change");
                ds.addObserver(new View("beta"));
                ds.setData("second change");

                ds.deleteObserver(v);

                ds.setData("third change");
        }
}
```

# DataSource.java

```
import java.util.Observable;

public class DataSource extends Observable {
        private static DataSource dataSource;                    // Singleton Pattern
        private String data;

        private DataSource() { this.data = "initialised”; }
        public String getData() { return data; }

        public void setData(String data) {
                this.data = data;
                setChanged();
                notifyObservers(data);                           // Observer Pattern
        }

        public static synchronized DataSource getDataSource() { // Singleton Pattern
                if (dataSource == null) { dataSource = new DataSource(); }
                return dataSource;
        }
}
```

# View.java

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

public class View implements Observer {
        private String name;

        public View(String name) {
                this.name = name;
        }

        public void update(Observable arg0, Object arg1) {
                System.out.println("View " + name +
                        ": observed value is '" + arg1 + "'");
        }
}
```

# 4. Decorator Pattern

- Used to deal with the situation when you’d end up with a large number of similar classes
- See also a good explanation using Java I/O as an example at
  http://stackoverflow.com/questions/6366385/decorator-pattern-for-io
- Think about if we were to model the software system in a cafe
  - Each drink would have a different class

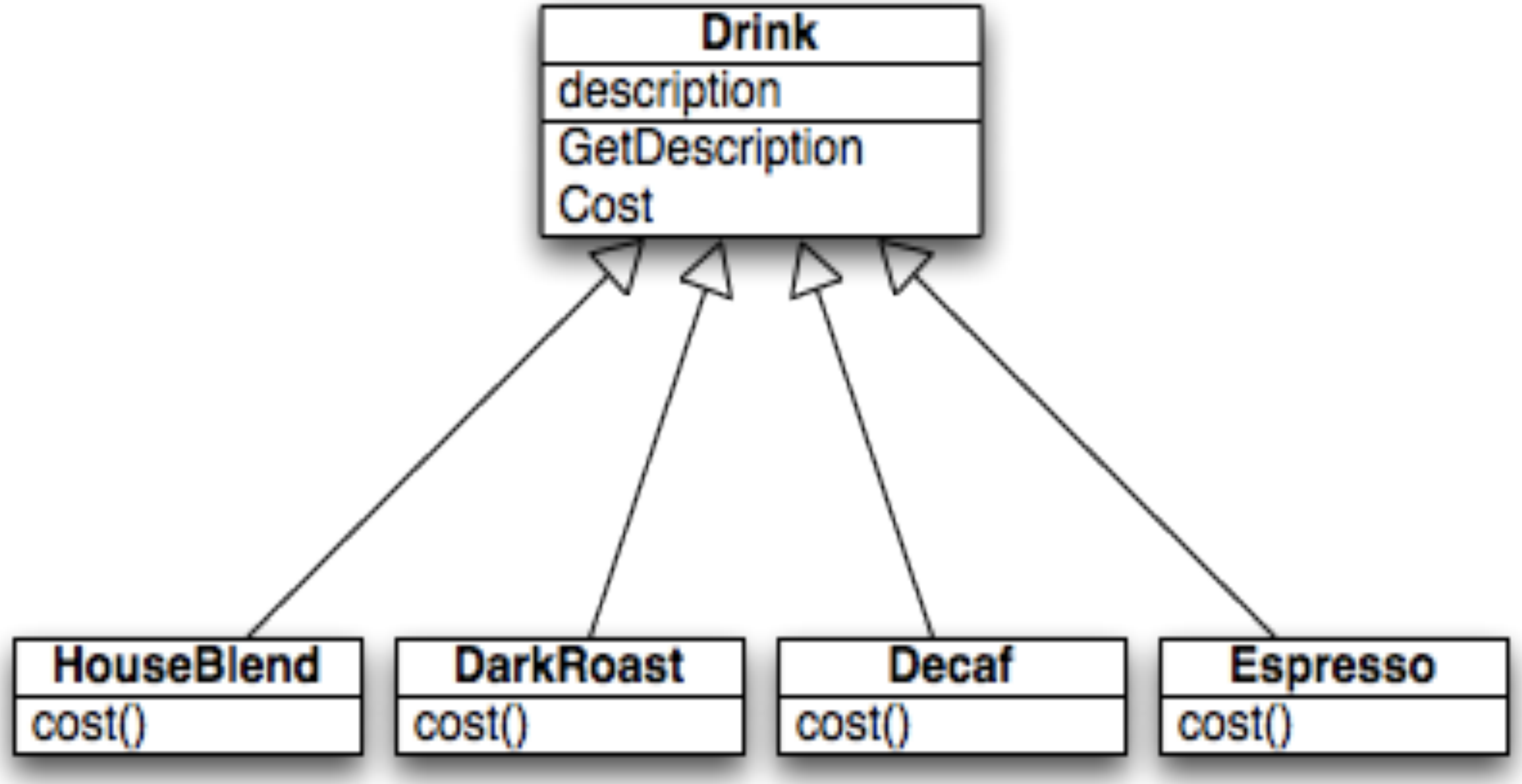
# Design principle

- “Favour composition over inheritance”
- **Composition**
  - Inheritance is fixed at compile time
  - Composed objects can be changed

# Drinks

- Coffees
  - House Blend
  - Dark Roast
  - Decaf
  - Espresso

- Toppings
  - Mocha
  - Steamed Milk
  - Soy
  - Whip

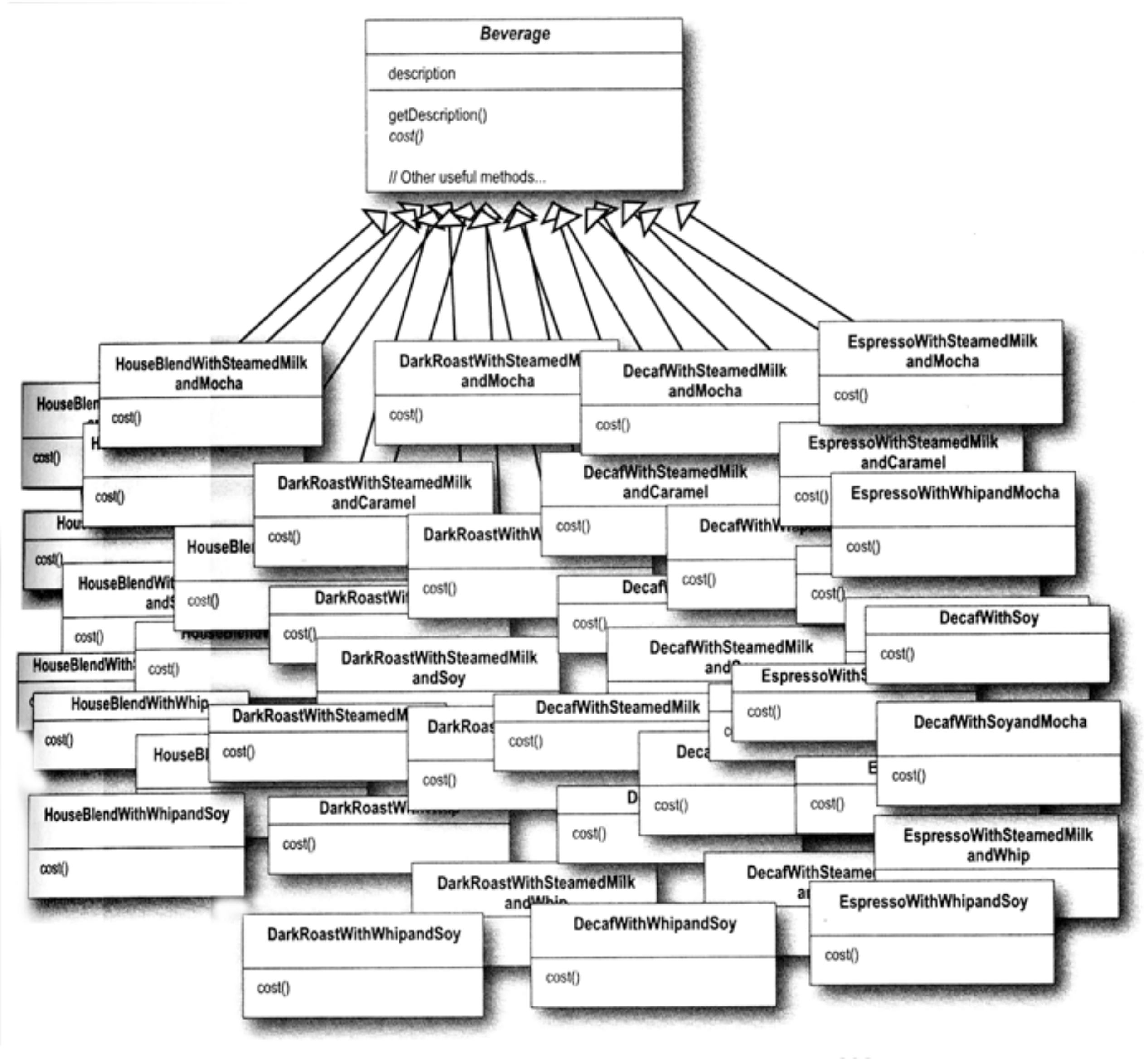

Drink
description
GetDescription
Cost
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()

```
public class Drink
{
    protected String description;

    public string getDescription()
    {
        return description;
    }

    public abstract double cost();
}
```

```
public class HouseBlend extends Drink {
    private float cost = 0.89;

    public HouseBlend()  {
            description = “House Blend”;
    }

    public double cost() {
            return cost;
    }
}
```

# Toppings

- What about the Toppings?
- Inherit the various types?
  - EspressoWithMocha
  - DarkRoastWithSteamedMilk
  - …
- Implementation as before


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
HouseBlendWithWhip
DecafWithSoyandMocha
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()

# Class Explosion

- Where did all these classes come from?
  - 16 combination of toppings
  - 4 drinks
  - 64 different classes to implement
- Maintenance nightmare!
  - What if the cost of Mocha topping goes up?
- Expanding difficulty.
  - A new coffee means 16 new classes
  - A new topping means ≥64 new classes

# Favour Composition

- Inheritance 64 — Composition 0
- Nothing is encapsulated
  - DarkRoastWithMocha
  - HouseBlendWithMocha
  - HouseBlendWithSteamedMilk
- No Code Reuse…

# Inheritance

- Inheritance is powerful
- Doesn’t always lead to flexible designs
- Can ‘inherit’ behaviour at runtime via composition

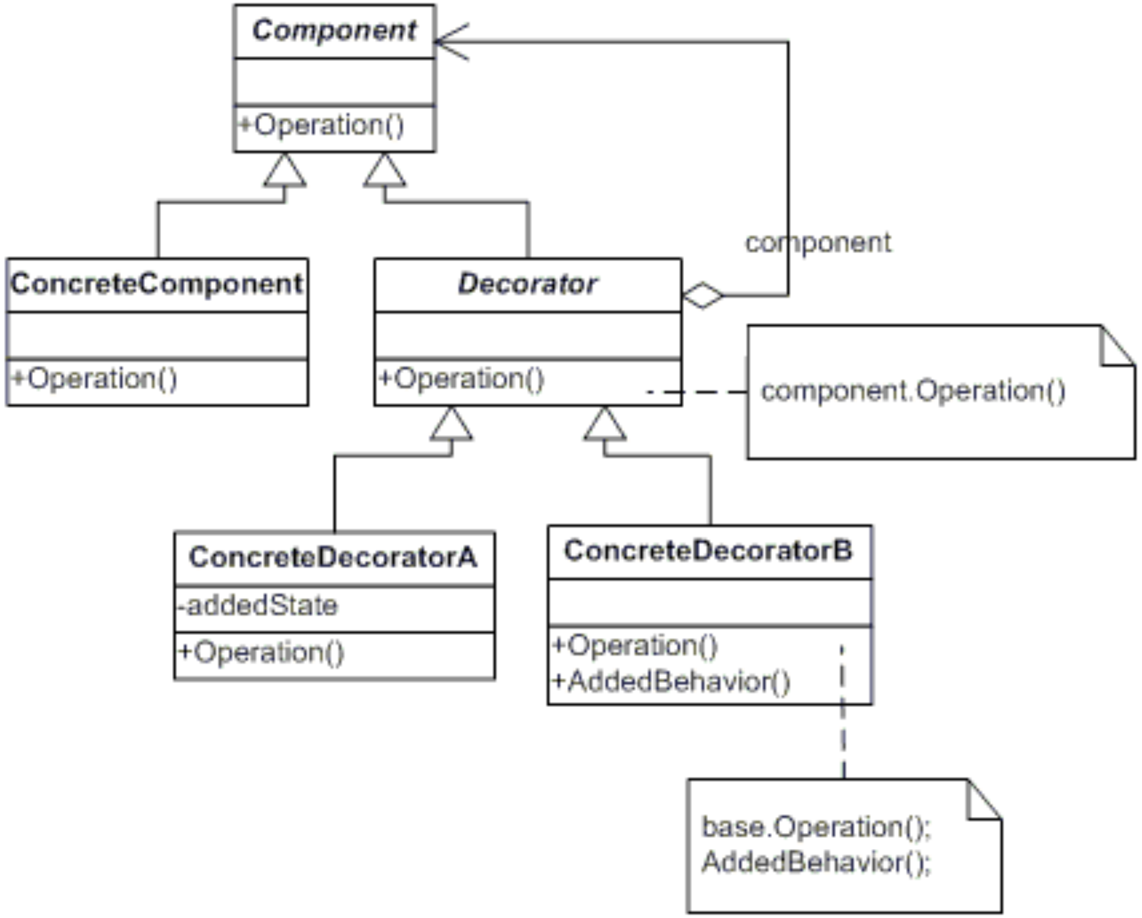
# Inherit or Compose

- Behaviour is fixed statically at compile time
- Composition can extend at runtime
- Composition allows us to add new responsibilities to objects without touching the superclass
- New functionality by writing new code, not editing old (and working) code

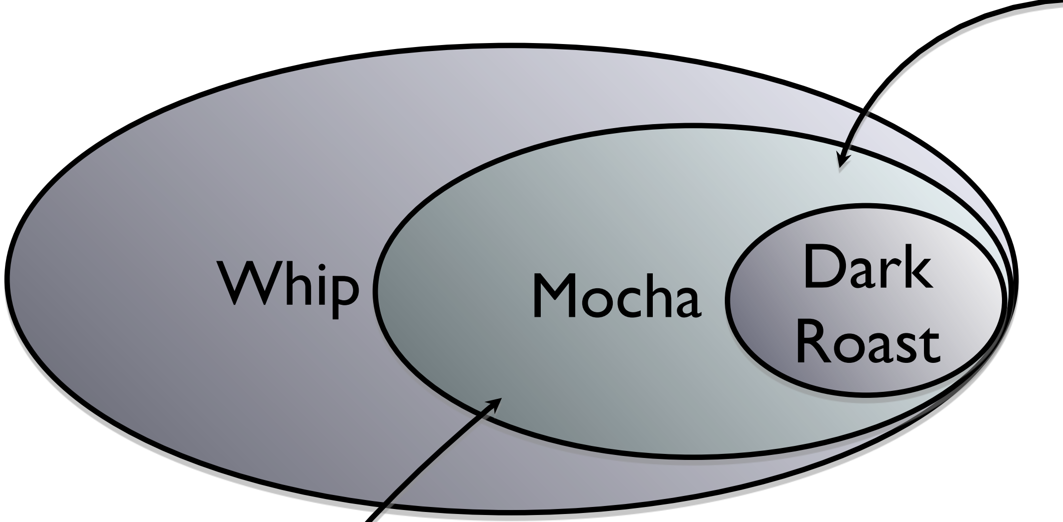
# Decorator Pattern

- Attach additional responsibilities or functions to an object dynamically or statically. Also known as Wrapper.
- Can use the **Decorator Pattern** to solve this design
- Uses composition rather than inheritance

# Decorator Pattern UML Diagram



All Objects are of the same type, Drink



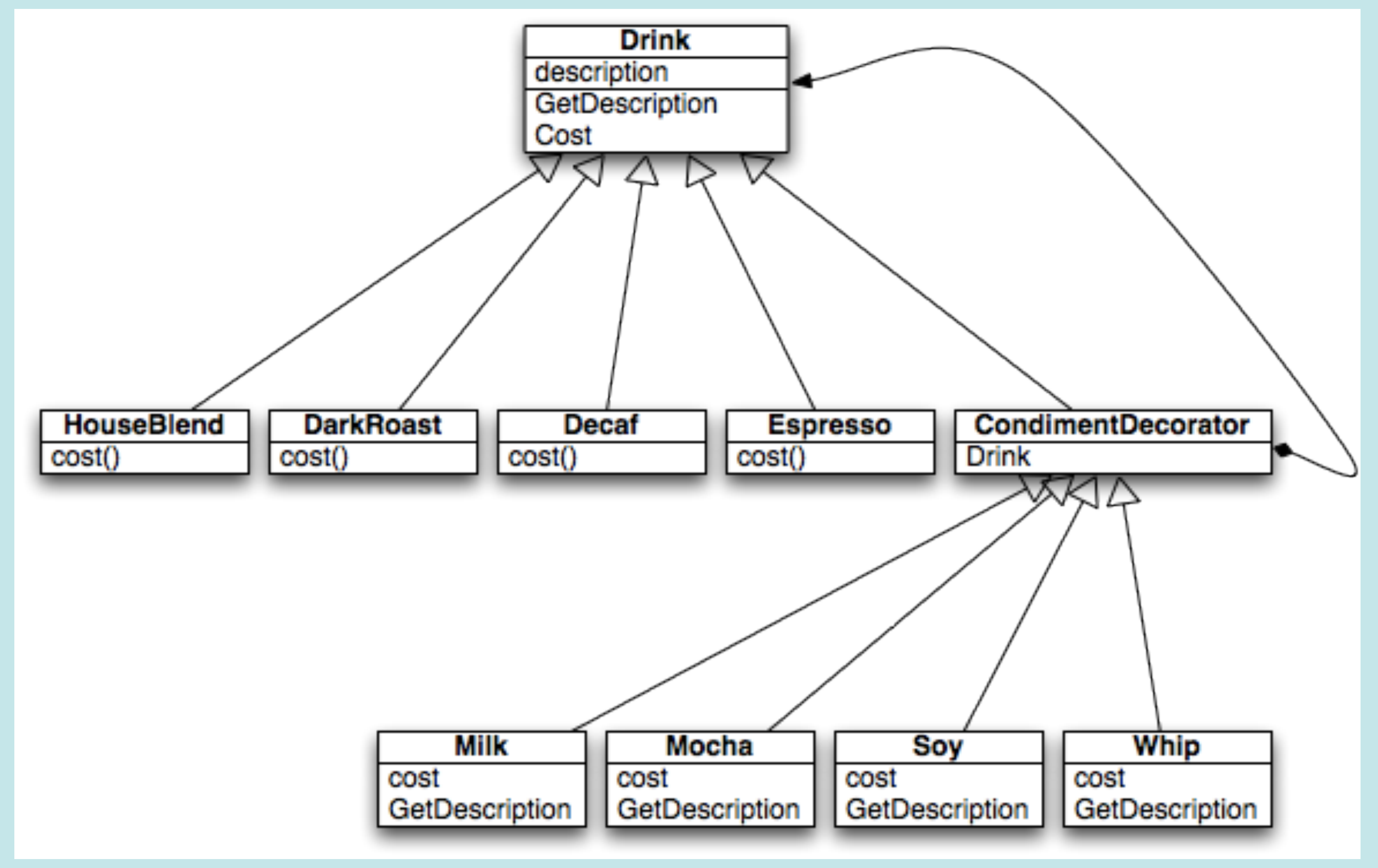
Mocha and Whip are 'Decorators', they take an object of type Drink and wrap it up

# Decorator

- Start with a DarkRoast object
- Customer wants Mocha, wrap a Mocha object around DarkRoast
- Also want Whip, wrap a Whip object around Mocha
- Both decorators and concrete classes share same type, Drink

# Coffee Decorators

- How it works
  - Call cost() on the decorated object
  - Decorator calls cost() on the object it decorates and adjusts the price
- Decorators can decorate Decorators


Drink
description
GetDescription
Cost
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
CondimentDecorator
Drink
Milk
cost
GetDescription
Mocha
cost
GetDescription
Soy
cost
GetDescription
Whip
cost
GetDescription

```
public class Drink {
    protected String description;

    public string getDescription()
    {
          return description;
    }

    public abstract double cost();
}
```

```
public class HouseBlend extends Drink {
    private float cost = 0.89;

    public HouseBlend()  {
          description = “House Blend”;
    }

    public double cost() {
          return cost;
    }
}
```

```
public class CondimentDecorator extends Drink {
    protected Drink drink;

    public abstract String getDescription();
    public abstract double cost();
}
```

```
public class Mocha extends CondimentDecorator {
    private float cost = 0.20;

    public Mocha(Drink drink) {
            this.drink = drink;
    }

    public String getDescription() {
            return drink.getDescription() + “, Mocha”;
    }
    public double cost() {
            return drink.cost() + cost;
    }
}
```

```
Drink myHouseBlendMocha  = new Mocha(new HouseBlend());
```

# Decorator Pattern

- The *Decorator Pattern* attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- Black-box reuse
- Classes implementation unchanged

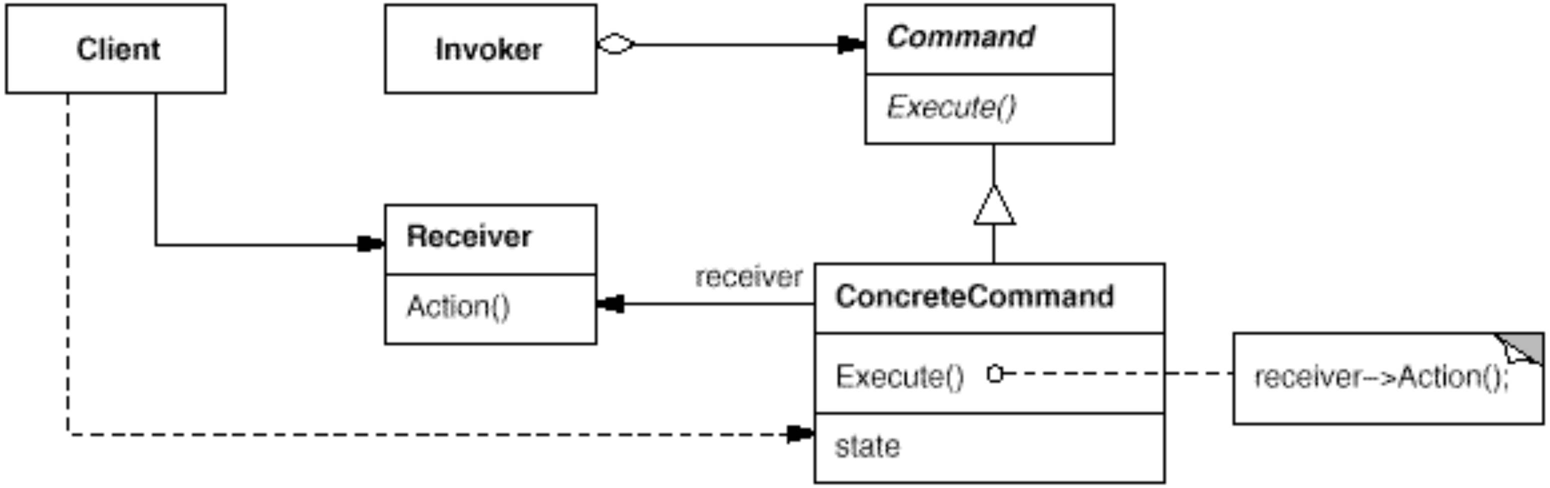
# Design Principle

- “Classes should be open for extension, but closed for modification”

# Open-Closed Principle

- *Classes should be open for extension*
- Feel free to extend our classes with any new behaviour you like, but…
- *closed for modification*
- Sorry, but our code is fixed and bug free you can’t change it

# 5. Command Pattern UML



# Command pattern motivation

The Command Pattern is useful when:

- A history of requests is needed
- You need callback functionality
- Requests need to be handled at variant times or in variant orders
- The invoker should be decoupled from the object handling the invocation.
- Allows “undo” operation (ie *un-execute*)

# Command pattern notes

- Downside…
- Ends up forcing a lot of Command classes
  - makes your design look cluttered
- Intelligence required of which Command to use and when
  - leads to possible maintenance issues for the central controller.

# Command Pattern Example