

CptS 322- Software Engineering Principles I

Design Patterns

Instructor: Sakire Arslan Ay
Fall 2023



World Class. Face to Face.

Outline

- Overview of design patterns
- Creational patterns
- Structural patterns
- Behavioral patterns

SOLID Design Principles

Software inevitably changes/evolves over time (maintenance, upgrade)

- **Single responsibility principle (SRP)**
 - Every class should have only one reason to be changed
 - If class "A" has two responsibilities, create new classes "B" and "C" to handle each responsibility in isolation, and then compose "A" out of "B" and "C"
- **Open/closed principle (OCP)**
 - Every class should be *open for extension* (derivative classes), but *closed for modification* (fixed interfaces)
 - Put the system parts that are likely to change into implementations (i.e. *concrete classes*) and define *interfaces* around the parts that are unlikely to change (e.g. *abstract base classes*)
- **Liskov substitution principle (LSP)**
 - Every implementation of an interface needs to fully comply with the requirements of this interface (requirements determined by its clients!)
 - Any algorithm that works on the interface, should continue to work for any substitute implementation
- **Interface segregation principle (ISP)**
 - Keep interfaces as small as possible, to avoid unnecessary dependencies
 - Ideally, it should be possible to understand any part of the code in isolation, without needing to look up the rest of the system code
- **Dependency inversion principle (DIP)**
 - High-level modules should not depend on low-level modules. *Decouple* them by formalizing their communication interface as an *abstract interface* based on the needs of the higher-level module.

Purpose of Design Principles

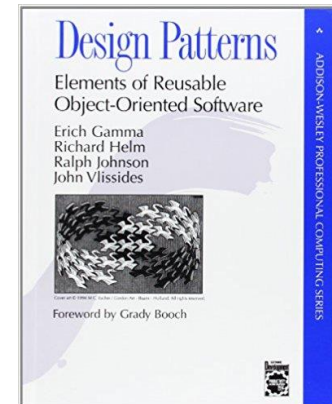
- **Principles** are used to *diagnose* problems with designs
- **Patterns** are used to *address* the problems

What is a design pattern?

- A standard solution to a common programming problem
 - a design or implementation structure that achieves a particular purpose
 - a high-level programming idiom
- A technique for making code more flexible or efficient
 - reduce coupling among program components
 - reduce memory overhead
- Shorthand for describing program design
 - a description of connections among program components
 - the shape of an object model

References:

- Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, 1995.
- Head First Design Patterns, by Eric Freeman and Elisabeth Robson.



Why we care about design patterns?

- You could come up with these solutions on your own ...
- But you shouldn't have to!
- A design pattern is a known solution to a known problem.

Types of Design Patterns

- Creational patterns
 - how objects are instantiated
- Structural patterns
 - how objects / classes can be combined
- Behavioral patterns
 - how objects communicate

Kinds of Creational Patterns

- Factory (method)
 - Abstract factory
 - Builder
 - Prototype
 - Flyweight
 - Singleton
- Creational patterns abstract the object instantiation process
 - They hide how objects are created and help make the overall system independent of how its objects are created and composed.

Kinds of Structural Patterns

- Composite
- Decorator
- Adapter
- Proxy
- ...
 - Structural patterns enable client code to:
 1. unify access to composite objects
 2. modify the interface
 3. extend behavior

Kinds of Behavioral Patterns

- Template method
- Iterator
- Strategy
- Null Object
- ...

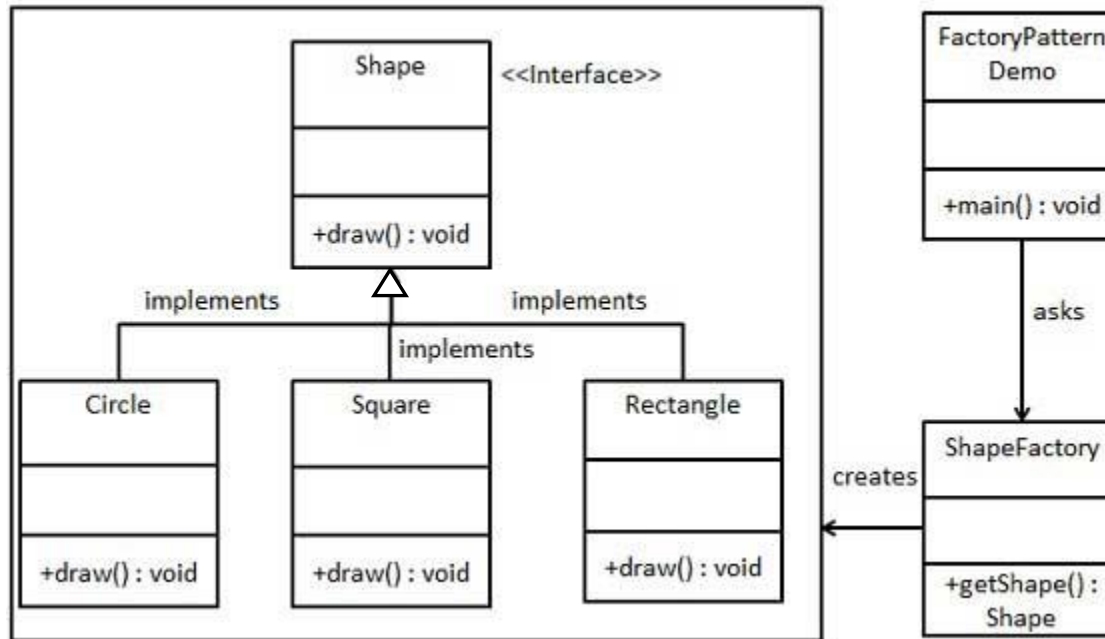
- Behavioral patterns identify and capture common patterns of communication between objects.

1. Factory Method Pattern (problem)

```
public interface Shape{
    public void draw();
}
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Rectangle::draw() method.");
    }
}
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square::draw() method.");
    }
}
```

- Clients use the supertype (Shape)
- But still need to use a Rectangle or Square constructor
- Must decide concrete implementation somewhere
- Don't want to change code to use a different constructor

1. Factory Method Pattern (solution)



- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

1. Factory Method Pattern (solution)

```
public class ShapeFactory {  
    //use getShape method to get object of type shape  
    public Shape getShape(String shapeType){  
        if (shapeType.equalsIgnoreCase("CIRCLE")) {  
            return new Circle();  
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {  
            return new Rectangle();  
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {  
            return new Square();  
        }  
        return null;  
    }  
}
```

- Clients call `getShape` instead of a particular constructor
- Advantages:
 - To switch the implementation, change only one place
 - `getShape` can do arbitrary computations to decide what kind of shape to create
- Frequently used in frameworks (e.g., Java swing)
 - `BorderFactory.createRaisedBevelBorder()`

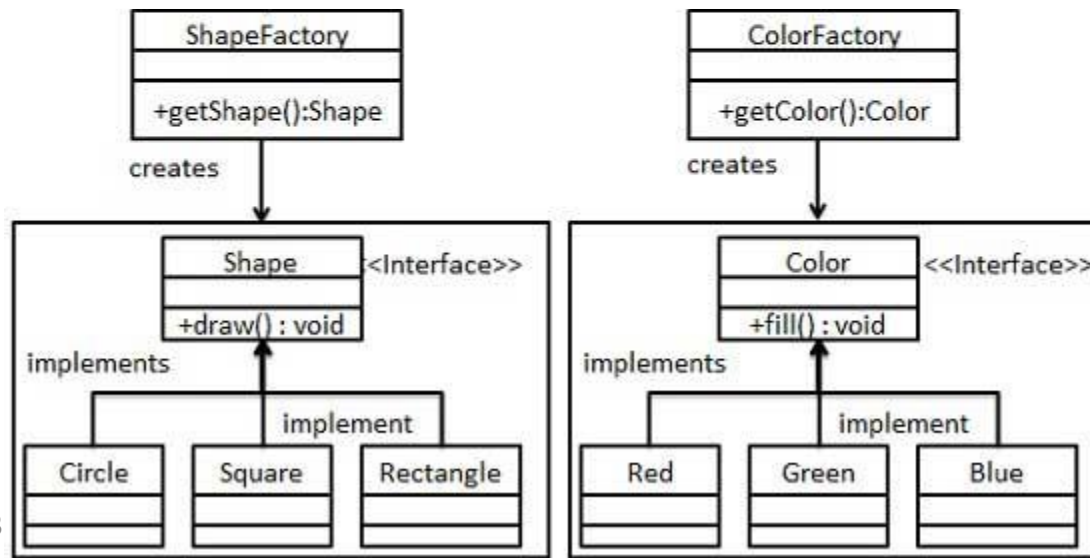
1. Factory Method Pattern - demo

```
public class FactoryPatternDemo {  
  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        //get an object of Rectangle and call its draw method.  
        Shape shape1 = shapeFactory.getShape(args[0]);  
  
        //call draw method of Rectangle  
        shape1.draw();  
  
        //get an object of Square and call its draw method.  
        Shape shape2 = shapeFactory.getShape(args[1]);  
  
        //call draw method of circle  
        shape2.draw();  
    }  
}  
  
java FactoryPatternDemo ["RECTANGLE","SQUARE","CIRCLE"]
```

- We can now use the Factory to get object of concrete class by passing an information such as type

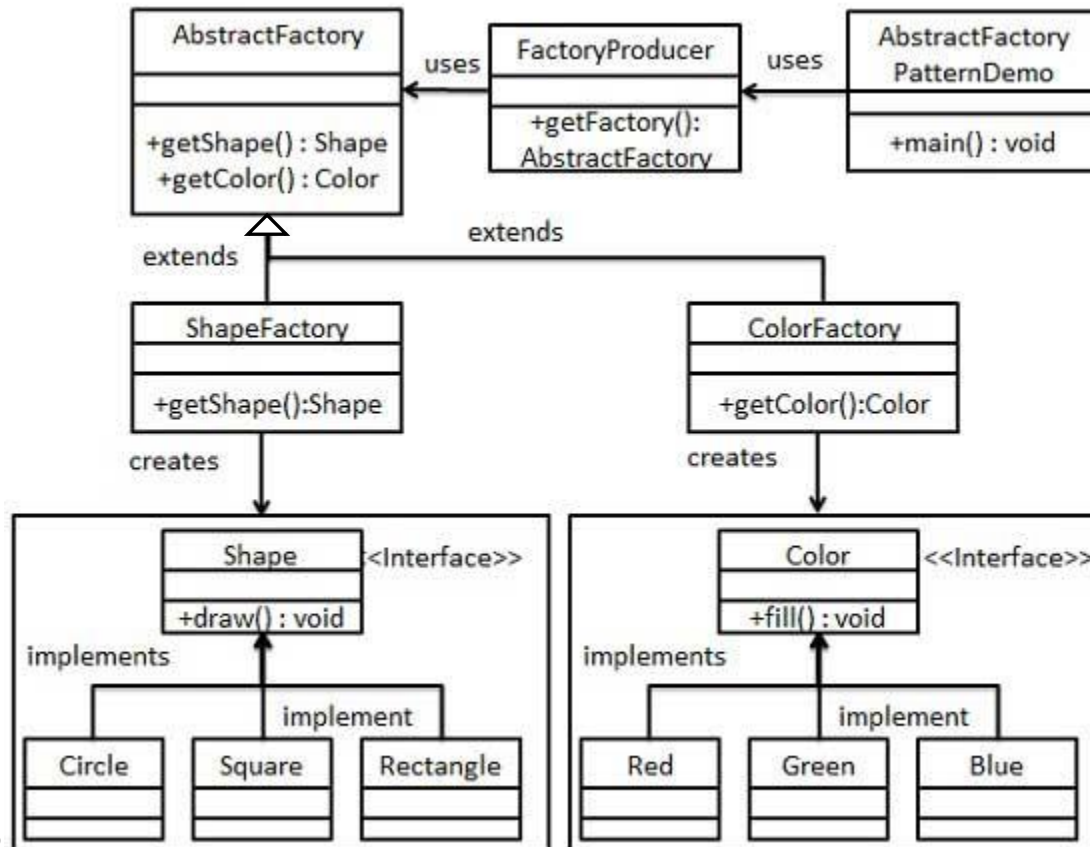
2. Abstract Factory Pattern

- Abstract Factory patterns involves a “super-factory” which creates other factories. This factory is also called as factory of factories.
- In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes.



Abstract Factory Pattern

- Abstract Factory patterns involves a “super-factory” which creates other factories. This factory is also called as factory of factories.
- In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes.



2. Abstract Factory Pattern Example (step1)

- Create an interface for Shapes.
- Create concrete classes implementing the Shapes interface.

```
public interface Shape{
    void draw();
}
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Rectangle::draw() method.");
    }
}
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Square::draw() method.");
    }
}
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Circle::draw() method.");
    }
}
```

2. Abstract Factory Pattern Example (step2)

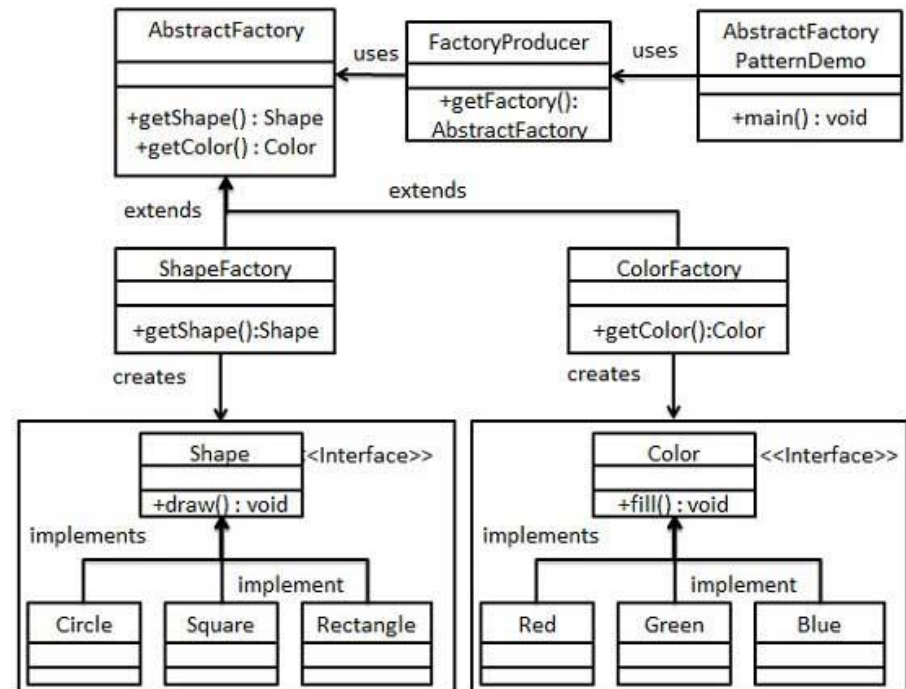
- Create an interface for Colors.
- Create concrete classes implementing the Colors interface.

```
public interface Color{
    void fill();
}
public class Red implements Color{
    @Override
    public void fill() {
        System.out.println("Red::fill() method.");
    }
}
public class Green implements Color{
    @Override
    public void fill() {
        System.out.println("Green ::fill() method.");
    }
}
public class Blue implements Color{
    @Override
    public void fill() {
        System.out.println("Blue ::fill() method.");
    }
}
```

2. Abstract Factory Pattern Example (step3)

- Create an Abstract class to create factories for Color and Shape objects.

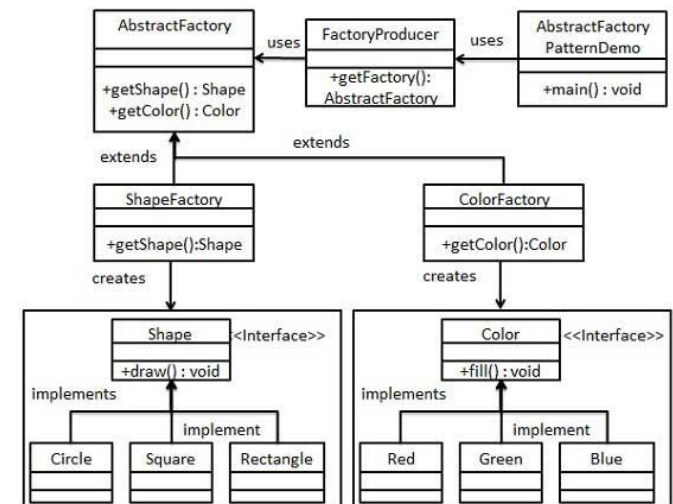
```
public abstract class AbstractFactory {  
    abstract Color getColor(String color);  
    abstract Shape getShape(String shape) ;  
}
```



2. Abstract Factory Pattern Example (step4)

- Create Factory classes extending `AbstractFactory` to generate object of concrete class based on given information.

```
public class ShapeFactory extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType) {  
        if (shapeType.equalsIgnoreCase("CIRCLE")) {  
            return new Circle();  
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {  
            return new Rectangle();  
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {  
            return new Square();  
        }  
        return null;  
    }  
  
    @Override  
    Color getColor(String color) {  
        return null;  
    }  
}
```



2. Abstract Factory Pattern Example (step4) – cont.

```
public class ColorFactory extends AbstractFactory {
```

```
    @Override
```

```
    public Shape getShape(String shapeType) {  
        return null;  
    }
```

```
    @Override
```

```
    Color getColor(String color) {
```

```
        if (color.equalsIgnoreCase("RED")) {  
            return new Red();
```

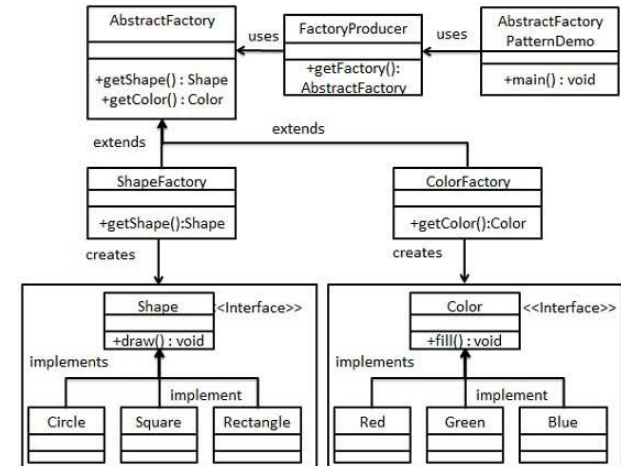
```
        } else if (color.equalsIgnoreCase("GREEN")) {  
            return new Green();
```

```
        } else if (color.equalsIgnoreCase("BLUE")) {  
            return new Blue();  
        }
```

```
        return null;
```

```
    }
```

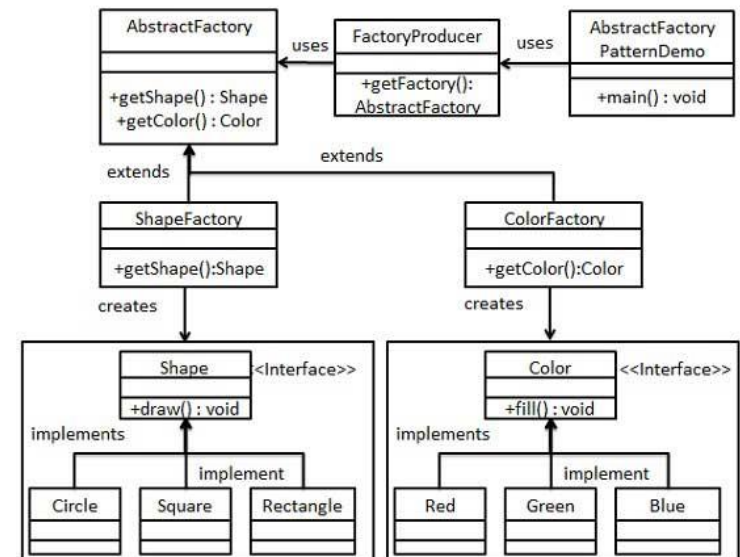
```
}
```



2. Abstract Factory Pattern Example (step5)

- Create a Factory generator/producer class to create factories by passing an information such as Shape or Color.

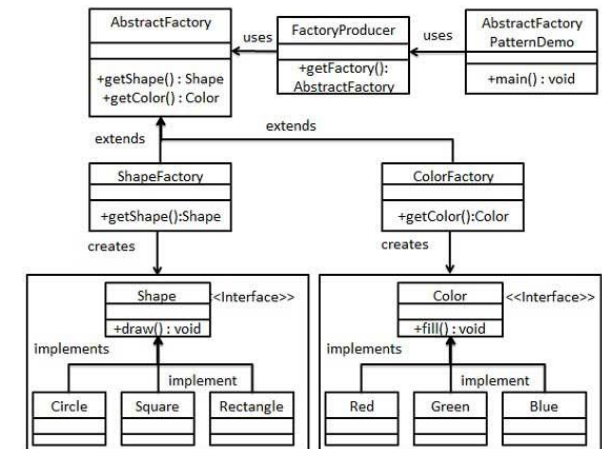
```
public class FactoryProducer {  
  
    public static AbstractFactory getFactory(String choice) {  
  
        if (choice.equalsIgnoreCase("SHAPE")) {  
            return new ShapeFactory();  
  
        } else if (choice.equalsIgnoreCase("COLOR")) {  
            return new ColorFactory();  
        }  
  
        return null;  
    }  
}
```



2. Abstract Factory Pattern Example (demo)

- Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

```
public class AbstractFactoryPatternDemo {  
    public static void main(String[] args) {  
  
        //get shape factory  
        AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");  
  
        //get an object of Shape Circle  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
        shape1.draw(); //call draw method of Shape Circle  
  
        //get an object of Shape Rectangle  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
        shape2.draw(); //call draw method of Shape Rectangle  
  
        //get color factory  
        AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");  
  
        //get an object of Color Red  
        Color color1 = colorFactory.getColor("RED");  
        color1.fill(); //call fill method of Red  
        //get an object of Color Blue  
        Color color3 = colorFactory.getColor("BLUE");  
        color3.fill(); //call fill method of Color Blue  
    }  
}
```



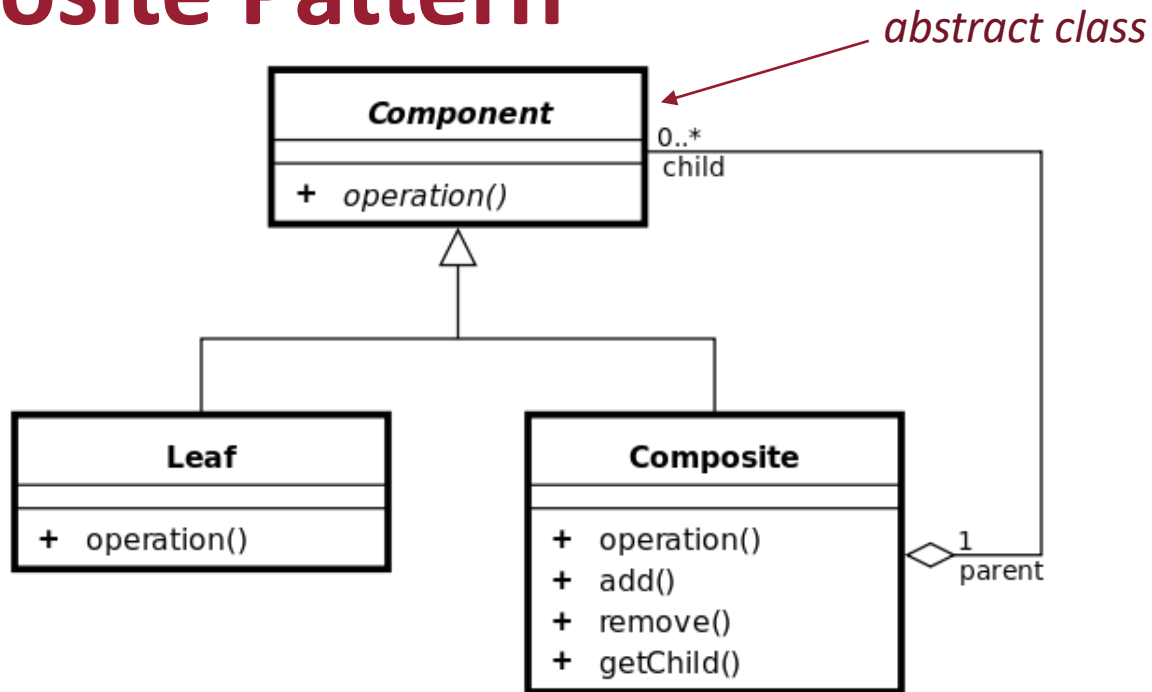
2. Abstract Factory Pattern

- Use the Factory pattern and Abstract Factory pattern in any of the following situations:
 - A system should be independent of how the objects are created, composed, and represented
 - A class can't anticipate the class of objects it must create.

Kinds of Structural Patterns

- Composite
 - Decorator
 - Adapter
 - Proxy
 - ...
- Structural patterns enable client code to:
 1. unify access of composite objects
 2. modify the interface
 3. extend behavior

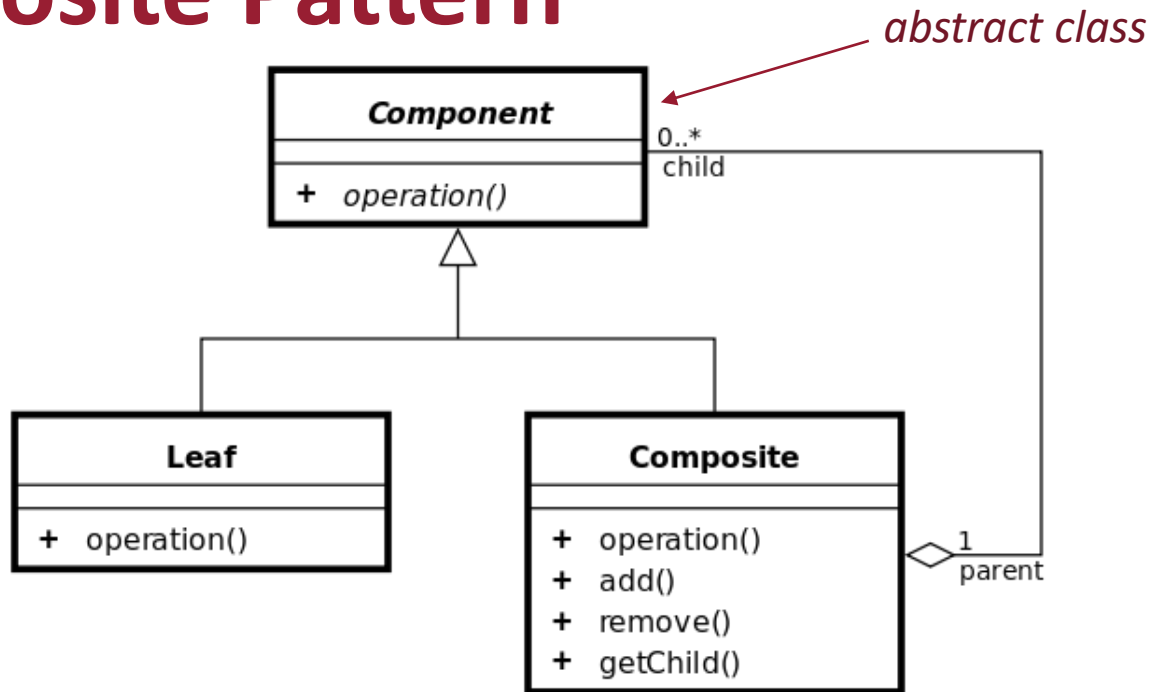
3. Composite Pattern



Composite pattern:

- Allows you to treat individual objects and composition of objects uniformly
- Allows you to represent part-whole hierarchies uniformly
 - Components can be further divided into smaller components

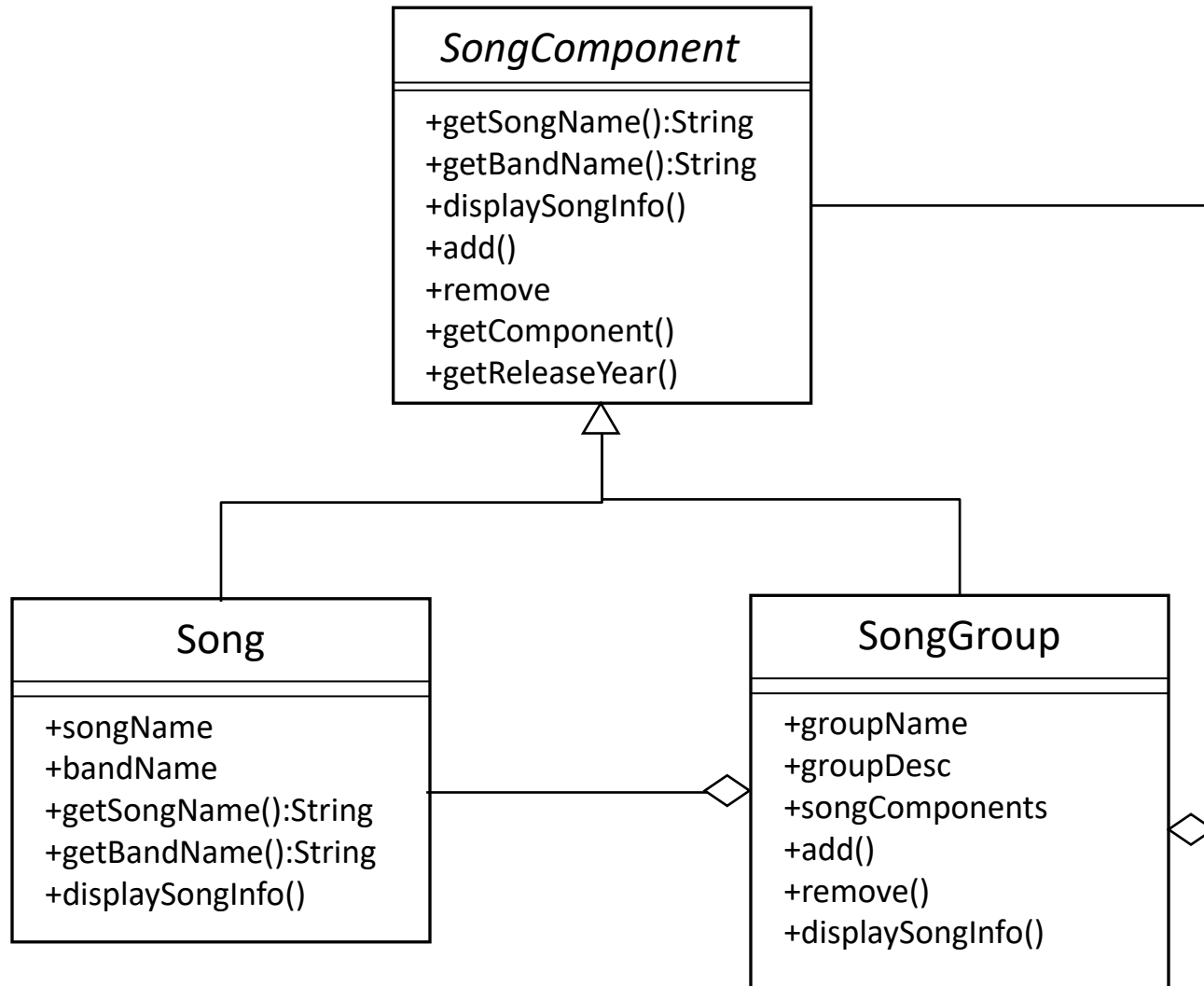
3. Composite Pattern



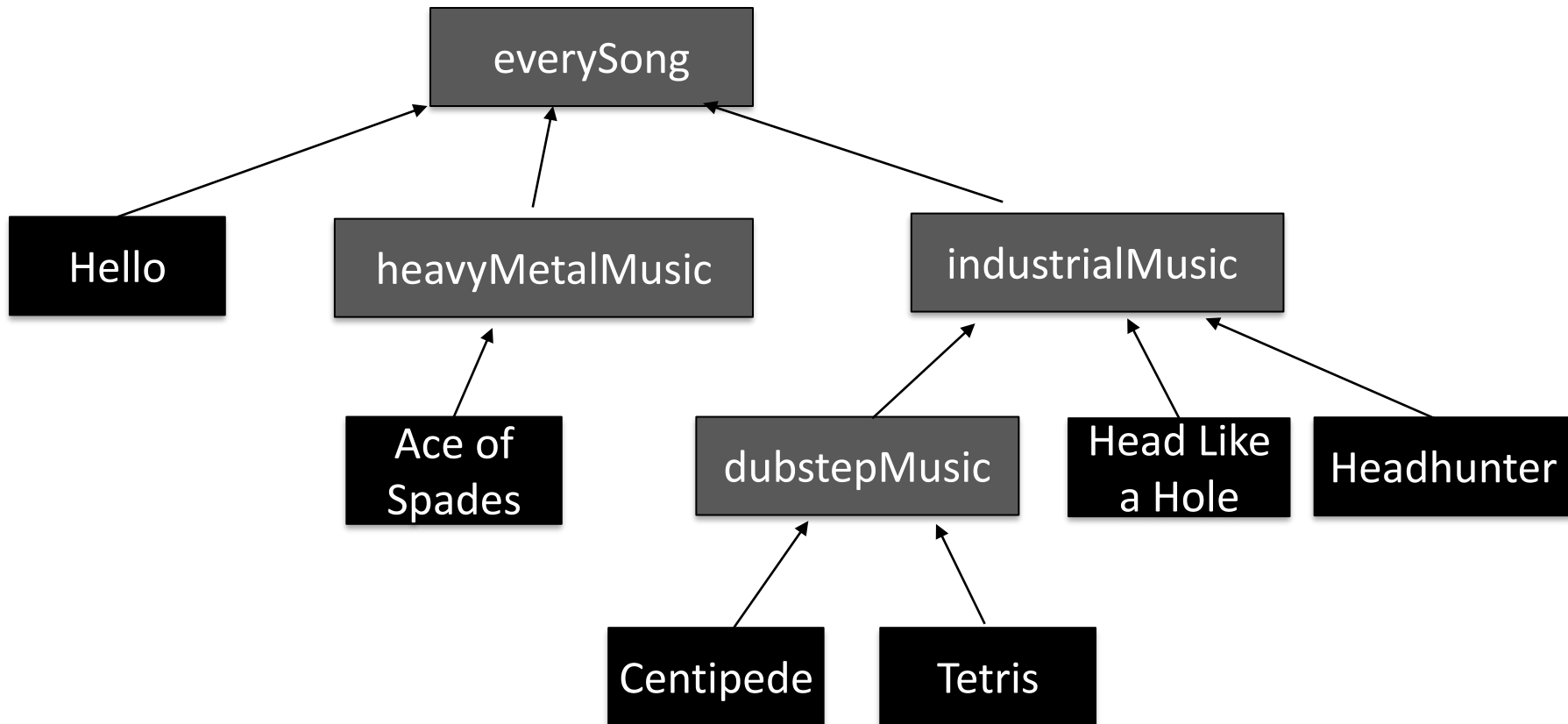
Composite pattern:

- Allows you to treat individual objects and composition of objects uniformly
- Allows you to represent part-whole hierarchies uniformly
 - Components can be further divided into smaller components

Composite Pattern - Example

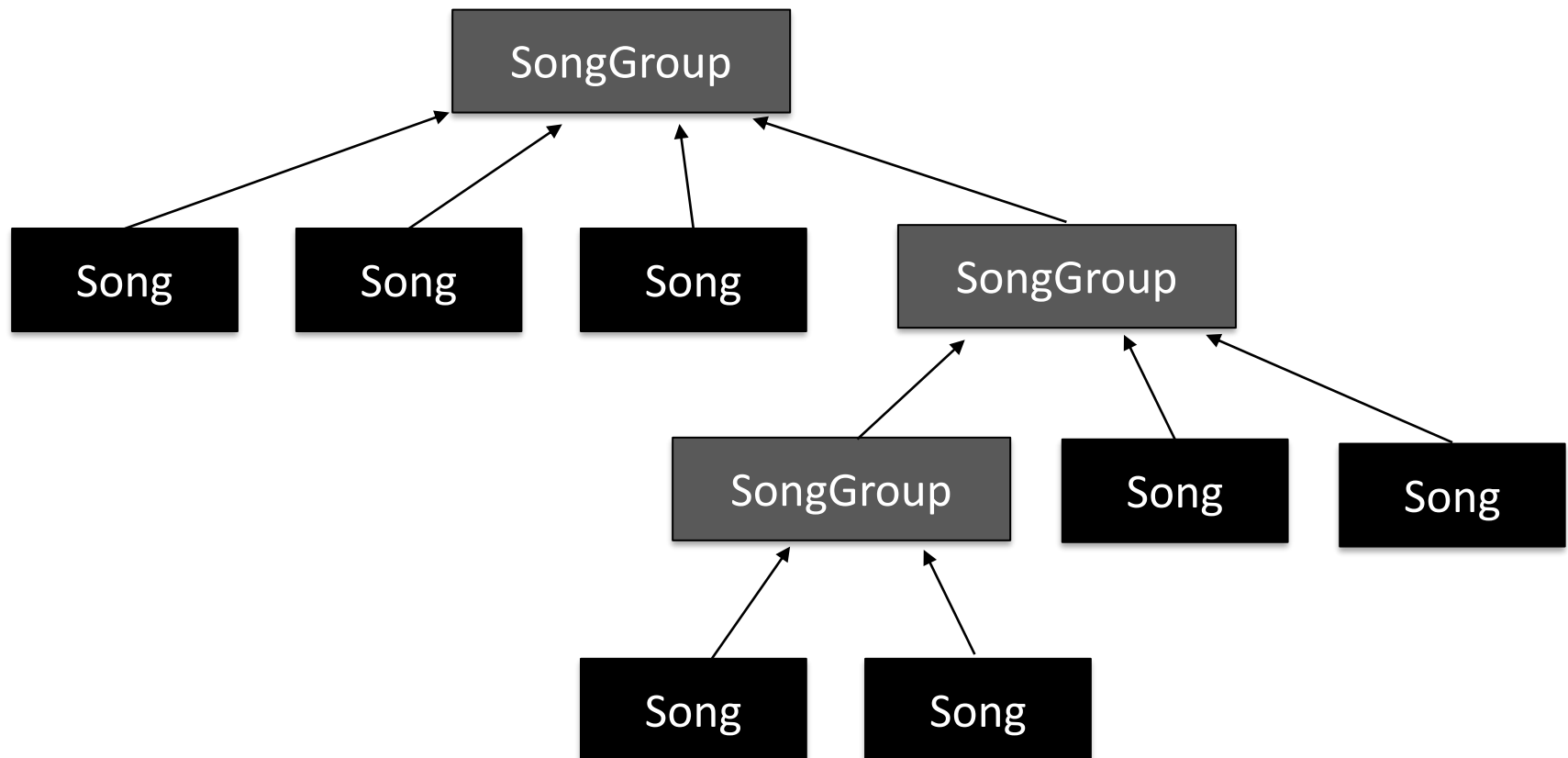


Composite Pattern - Example



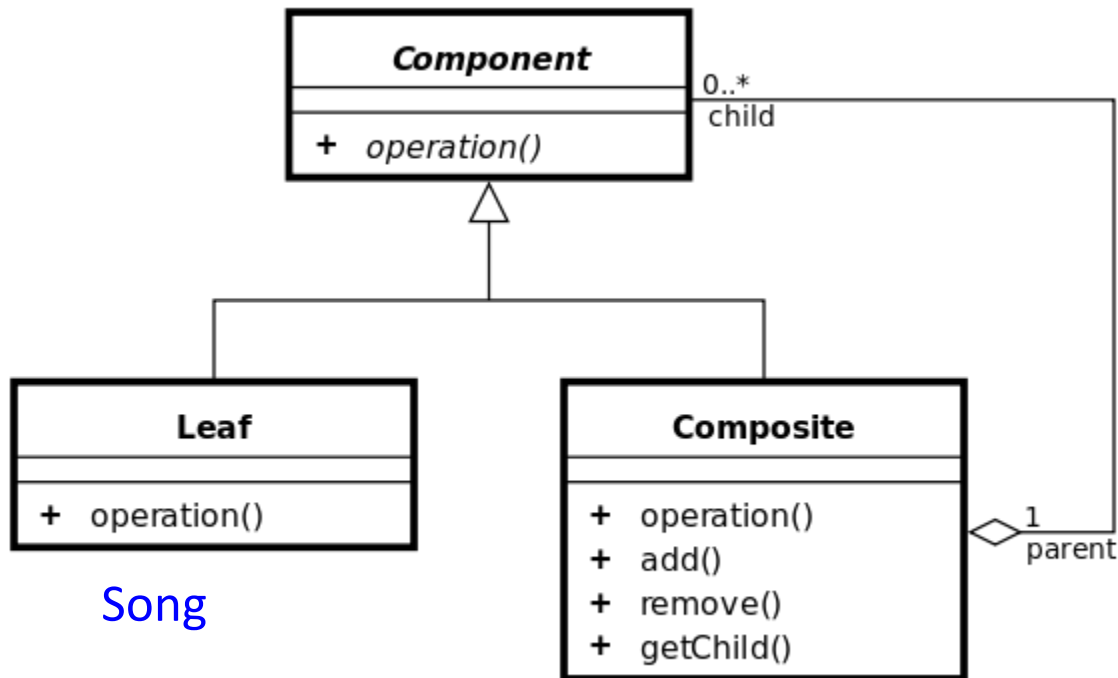
Composite Pattern - Example

- A composite can contain never ending groupings

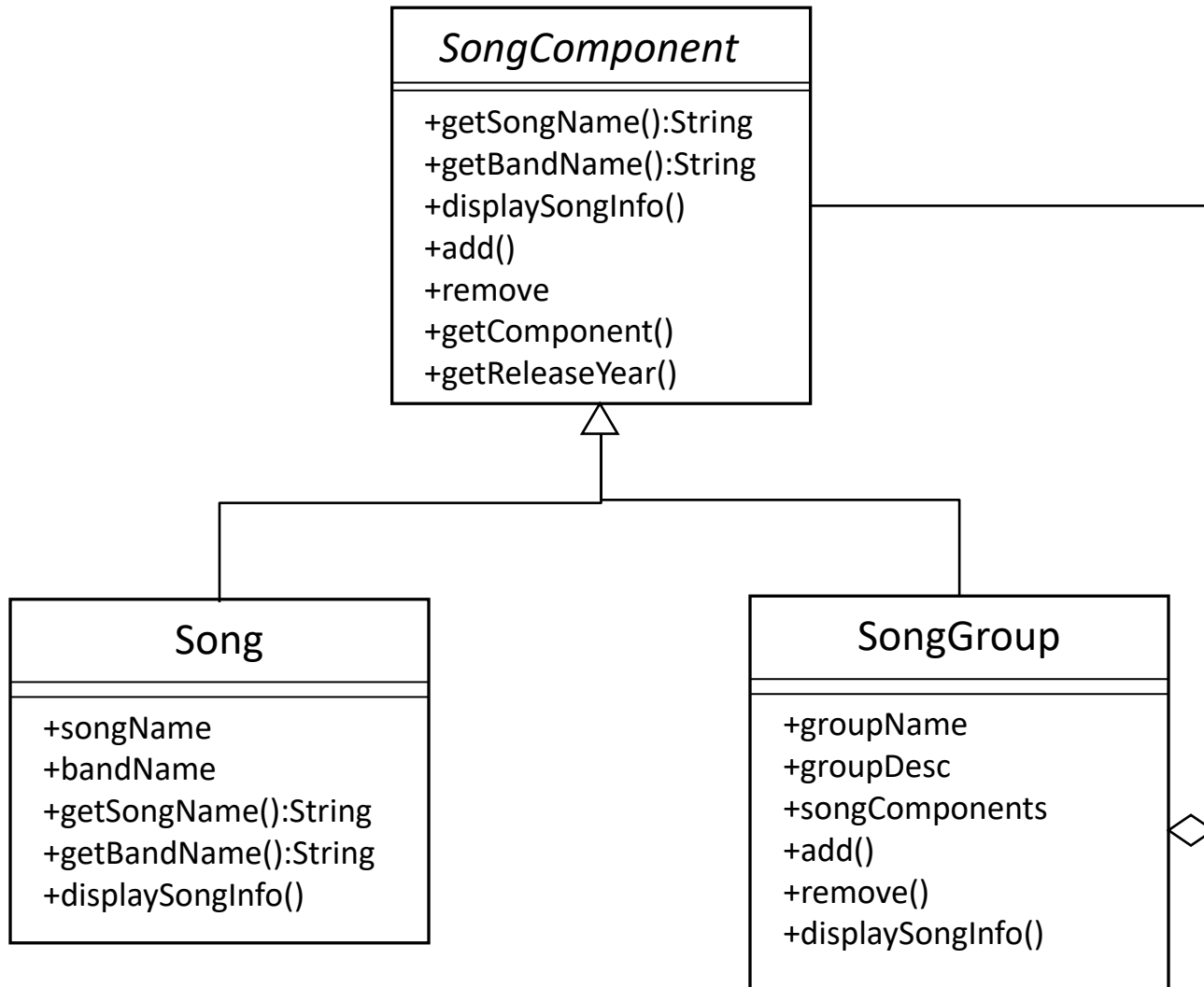


Composite Pattern - Example

SongComponent



Composite Pattern - Example

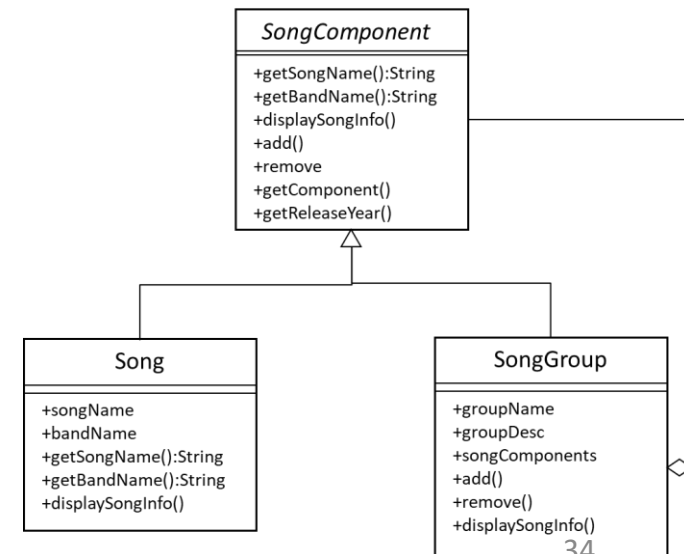


Composite Pattern – Example (step1)

SongComponent abstract class (Component):

- This acts as an interface for every Song (Leaf) and SongGroup (Composite) we create

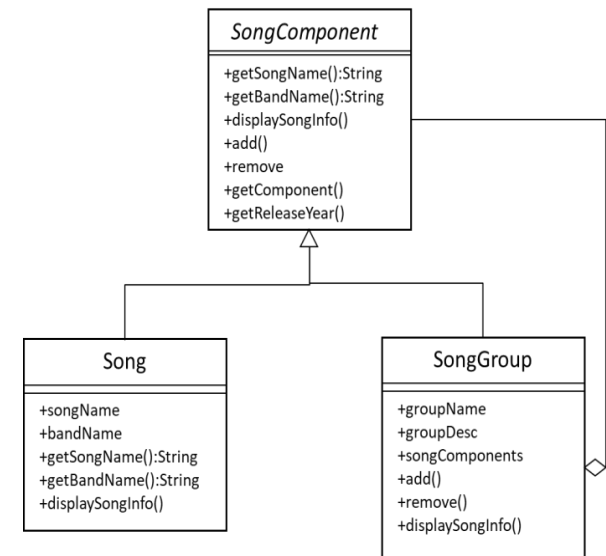
```
public abstract class SongComponent {  
    // Add components  
    public void add(SongComponent newSongComponent) { }  
    // Remove components  
    public void remove(SongComponent newSongComponent) { }  
    // Get component  
    public SongComponent getComponent(int componentIndex) { }  
    // Retrieve song names  
    public String getSongName() { }  
    // Retrieve band names  
    public String getBandName() { }  
    // Retrieve release year  
    public int getReleaseYear() { }  
    // When this is called by a class object  
    // that extends SongComponent it will  
    // print out information specific to the  
    // Song or SongGroup  
    public void displaySongInfo() { }  
}
```



Composite Pattern – Example (step2)

SongGroup class (Composite):

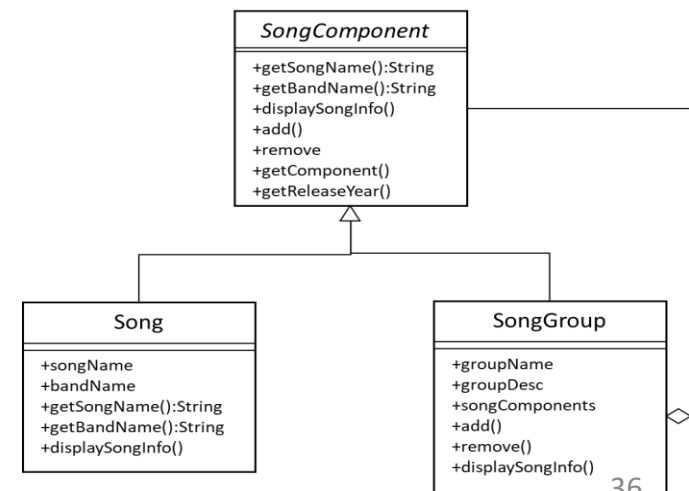
```
public class SongGroup extends SongComponent {  
    // Contains any Songs or SongGroups that are added to this ArrayList  
    ArrayList <songComponent> songComponents = new <songComponent> ArrayList();  
  
    String groupName;  
    String groupDescription;  
  
    public SongGroup(String newGroupName, String newGroupDescription){  
        groupName = newGroupName;  
        groupDescription = newGroupDescription;  
    }  
  
    public String getGroupName() { return groupName; }  
  
    public String getGroupDescription() {  
        return groupDescription;  
    }  
  
    public void add(SongComponent newSongComponent) {  
        songComponents.add(newSongComponent);  
    }  
  
    public void remove(SongComponent newSongComponent) {  
        songComponents.remove(newSongComponent);  
    }  
    .....  
}
```



Composite Pattern – Example (step2-cont.)

SongGroup class (Composite)- cont.:

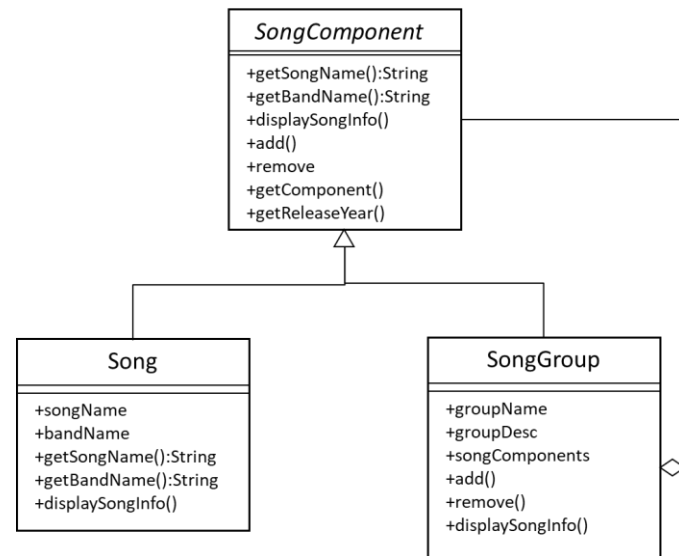
```
public class SongGroup extends SongComponent {
    .....
    public SongComponent getComponent(int componentIndex) {
        return (SongComponent) songComponents.get(componentIndex);
    }
    public void displaySongInfo() {
        System.out.println(getGroupName()+" "+getGroupDescription()+"\n");
        // Cycles through and prints any Songs or SongGroups added
        // to this SongGroups ArrayList songComponents
        Iterator songIterator = songComponents.iterator();
        while(songIterator.hasNext()) {
            SongComponent songInfo = (SongComponent) songIterator.next();
            songInfo.displaySongInfo();
        }
    }
}
```



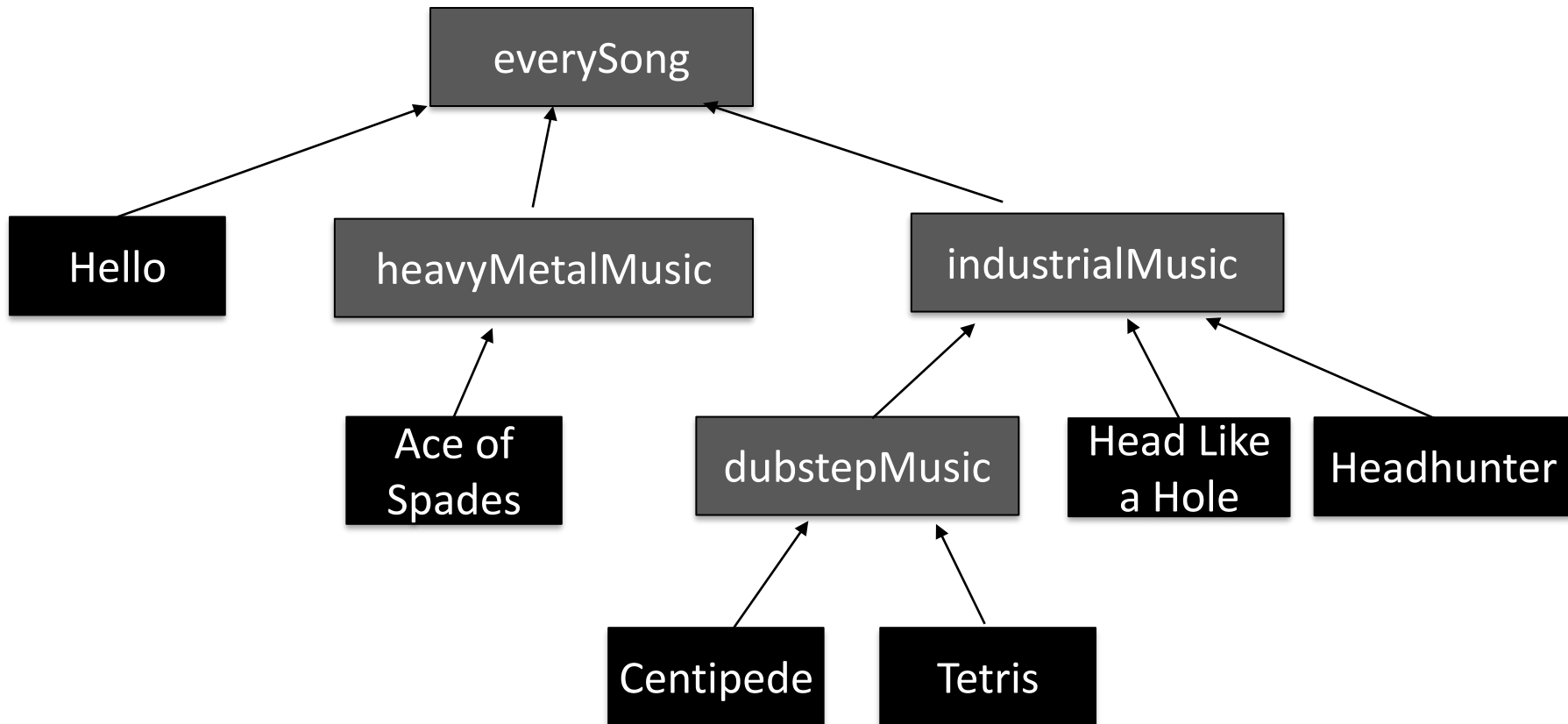
Composite Pattern – Example (step3)

Song class (Leaf):

```
public class Song extends SongComponent {  
    String songName;  
    String bandName;  
    int releaseYear;  
  
    public Song(String newSongName,  
                String newBandName,  
                int newReleaseYear) {  
        songName = newSongName;  
        bandName = newBandName;  
        releaseYear = newReleaseYear;  
    }  
  
    public String getSongName() { return songName; }  
    public String getBandName() { return bandName; }  
    public int getReleaseYear() { return releaseYear; }  
  
    public void displaySongInfo() {  
        System.out.println(getSongName() + " was recorded by " +  
                             getBandName() + " in " + getReleaseYear());  
    }  
}
```



Composite Pattern - Example



Composite Pattern – Example (step4 - cont.)

Client program:

```
public class SongListGenerator {

    public static void main(String[] args){

        SongComponent industrialMusic = new SongGroup("Industrial",
            "is a style of experimental music with provocative themes");
        SongComponent heavyMetalMusic = new SongGroup("\nHeavy Metal",
            "is a genre of rock that developed in the late 1960s");
        SongComponent dubstepMusic = new SongGroup("\nDubstep",
            "is a genre of electronic dance music");

        // Top level component that holds everything
        SongComponent everySong = new SongGroup("Song List", "All Songs");

        // Composite holding individual groups of songs and
        // a SongGroup with Songs
        everySong.add(industrialMusic);

        industrialMusic.add(new Song("Head Like a Hole", "NIN", 1990));
        industrialMusic.add(new Song("Headhunter", "Front 242", 1988));
        industrialMusic.add(dubstepMusic);

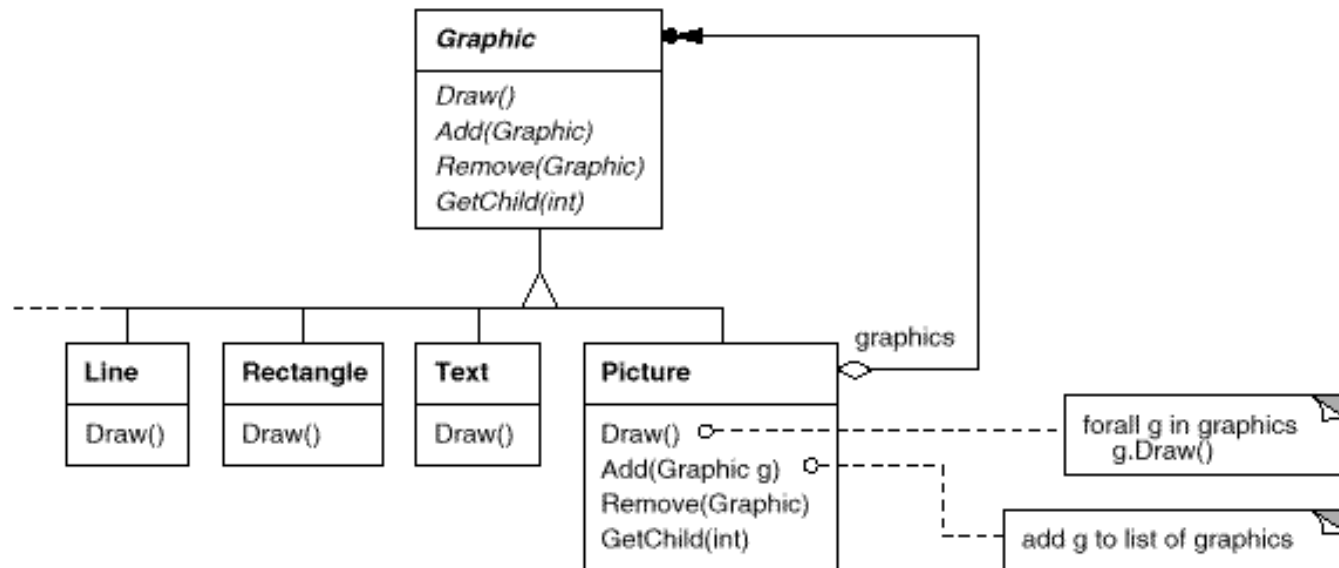
        dubstepMusic.add(new Song("Centipede", "Knife Party", 2012));
        dubstepMusic.add(new Song("Tetris", "Doctor P", 2011));
        .....
```

Composite Pattern – Example (step4 - cont.)

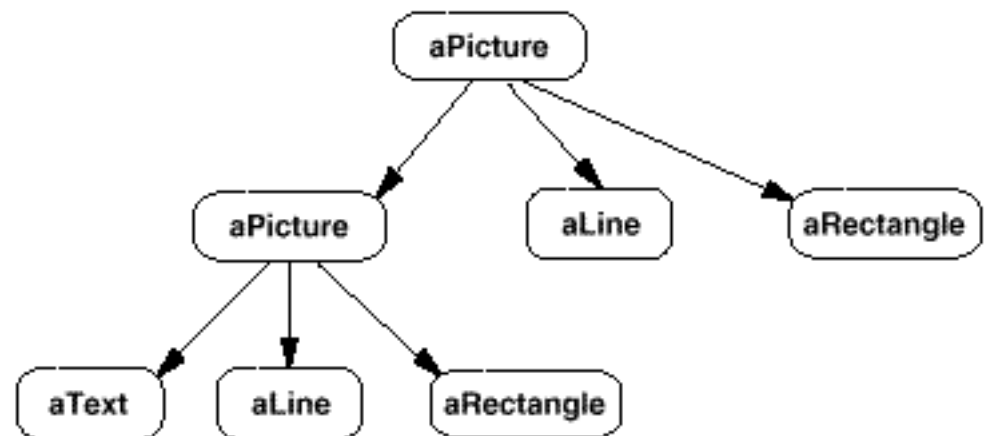
Client program:

```
public class SongListGenerator {  
  
    public static void main(String[] args){  
  
        .....  
        everySong.add(new Song("Hello", "Adele", 2015));  
  
        everySong.add(heavyMetalMusic);  
        heavyMetalMusic.add(new Song("Ace of Spades", "Motorhead", 1980));  
  
        everySong.displaySongInfo()  
    }  
  
}
```

Composite Pattern –Another Example



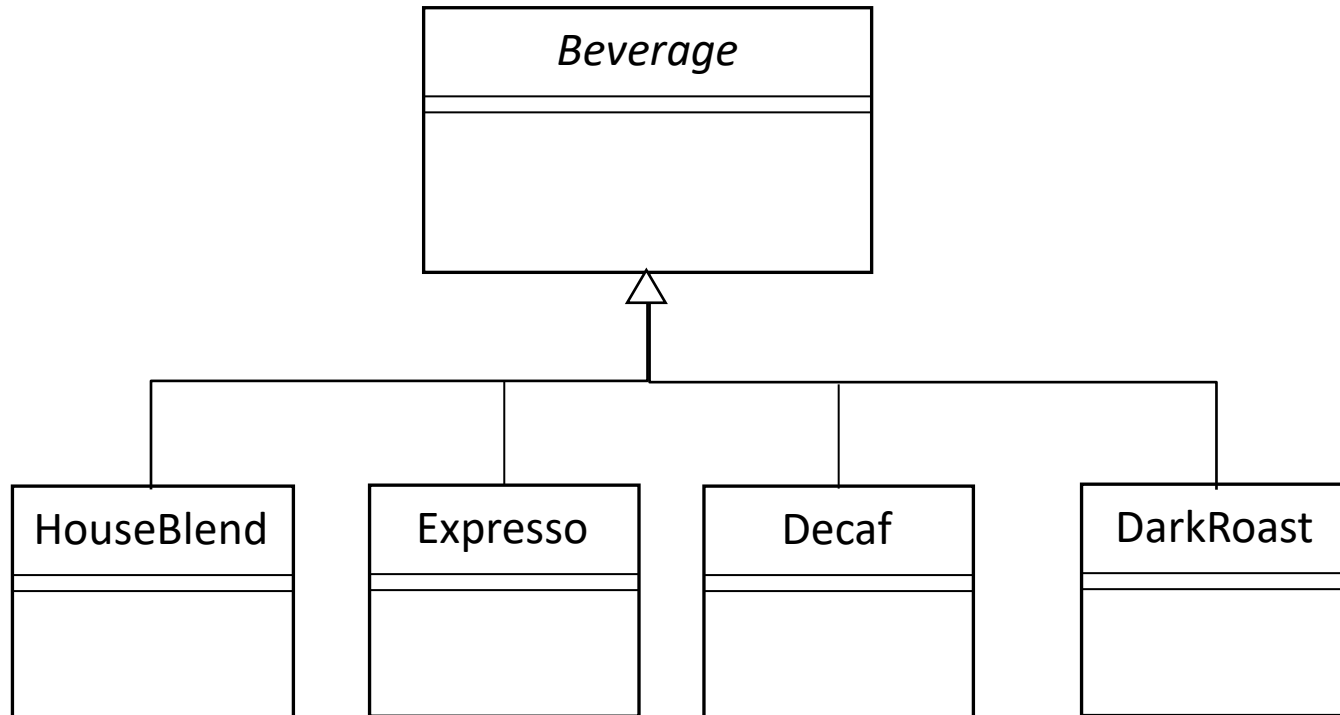
- In composite pattern there can be several leaf classes (all inheriting from the Component abstract class) .



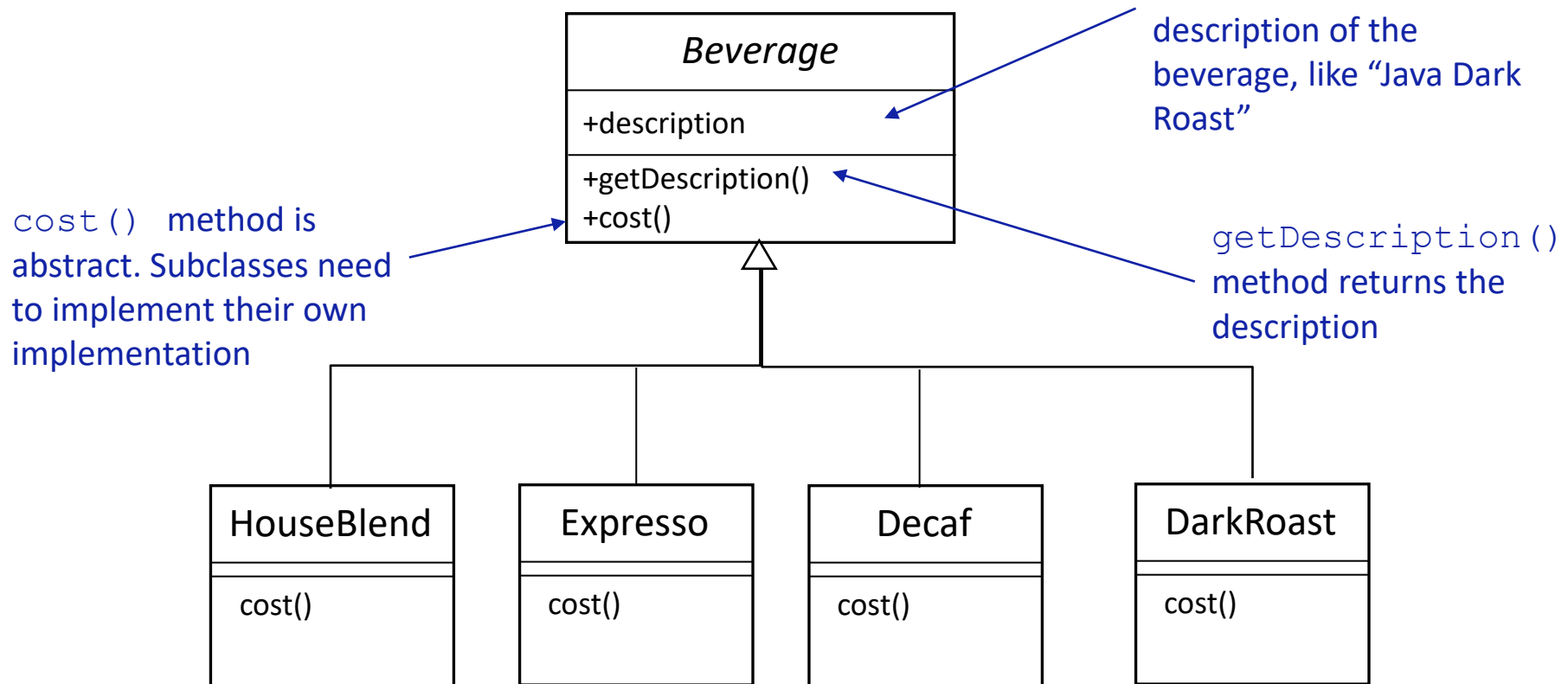
4. Decorator Pattern

- Attaches additional responsibilities to an object dynamically.
- Decorator provides a flexible alternative to subclassing for extending functionality.
- Example: StarBuzz Coffee
 - Several blends
 - HouseBlend, DarkRoast, Decaf, Espresso
 - Condiments
 - Steamed milk, soy, mocha, whipped milk
 - Extra charge for each
 - How do we charge all combinations?
 - First attempt: inheritance

StarBuzz Coffee Example



StarBuzz Coffee Example

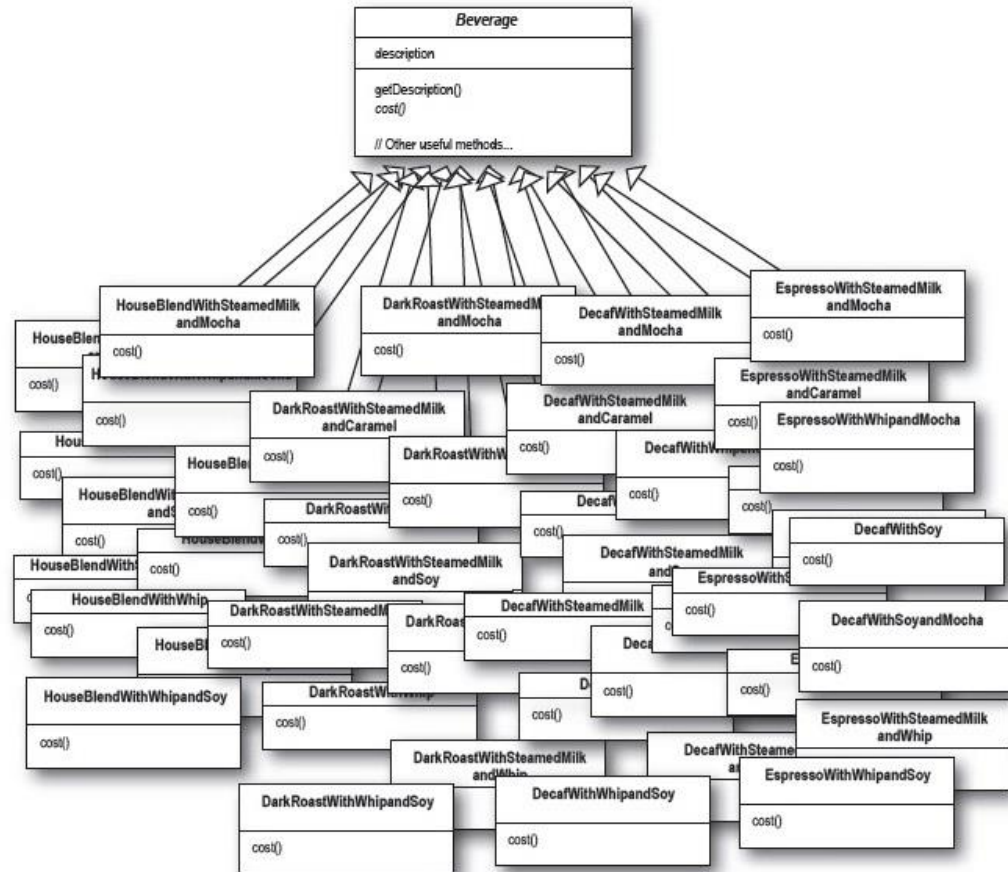


Each subclass implements `cost()` to return the cost of the beverage

StarBuzz Coffee Example

Problem with first attempt:

- How do we add the condiments?
 - Apply subclass and inheritance
 - Add subclasses and inheritance for condiments
 - But there can be combinations of condiments
 - Class explosion when all combinations are considered
- There must be a better way



StarBuzz Coffee Example

<i>Beverage</i>
+description: String +milk: boolean +soy: boolean +mocha: boolean +whip: boolean
+getDescription() +cost() +hasMilk() +setMilk() +hasSoy() +setSoy() +hasMocha() +setMocha() +hasWhip() +setWhip()

New boolean values for each condiment

Beverage's cost() calculates the cost of condiments.

The subclasses calculate the cost of the beverage and add the cost of condiments

Problems with this solution?

- New condiments – add new methods and alter cost method in superclass
 - Double mocha?
- Our goal is to simplify maintenance.
 - Code changes in the superclass when the above happens or in using combinations

Design Principle

- Classes should be open for extension, but closed for modification
 - Apply the principle to the areas that are most likely to change
- We want our designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements
- *How?* We want techniques *to allow code to be extended without direct modification*

StarBuzz Coffee Example

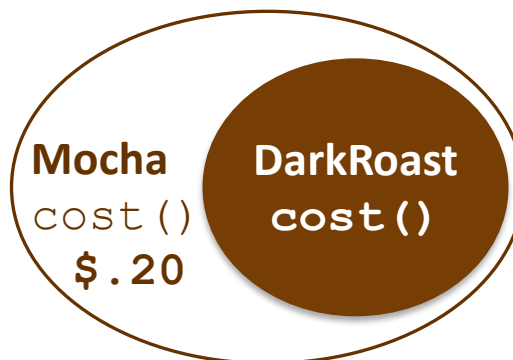
Use decorator pattern for starbuzz:

- Start with a beverage and decorate it with condiments at run time
- Example:
 - Take a **DarkRoast** object
 - Decorate it with a Mocha object
 - Decorate it with a Whip object
 - Call the `cost()` method and rely on delegation to add the condiment costs

Total \$.99



Total \$1.19

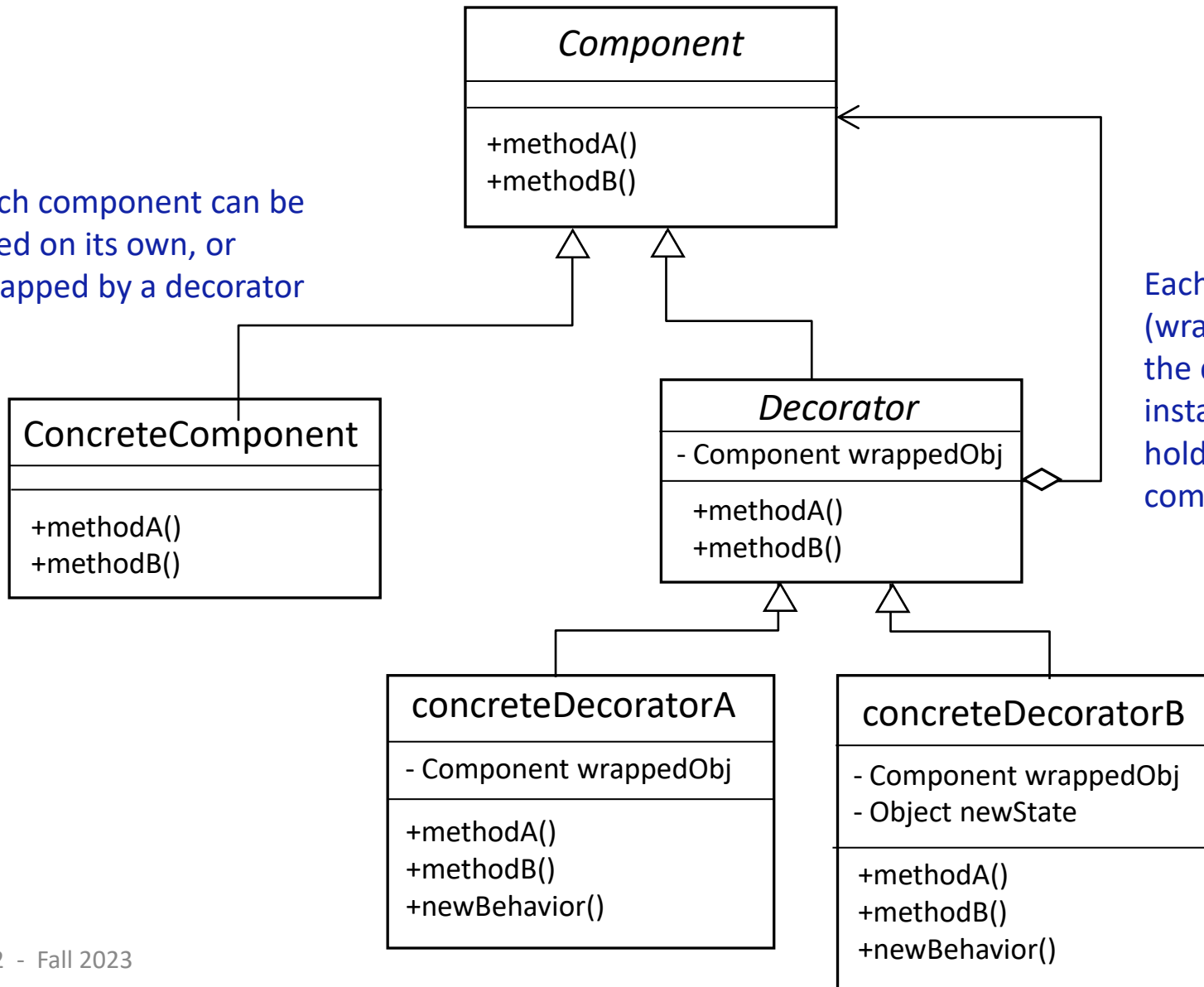


Total \$1.29



Decorator Pattern

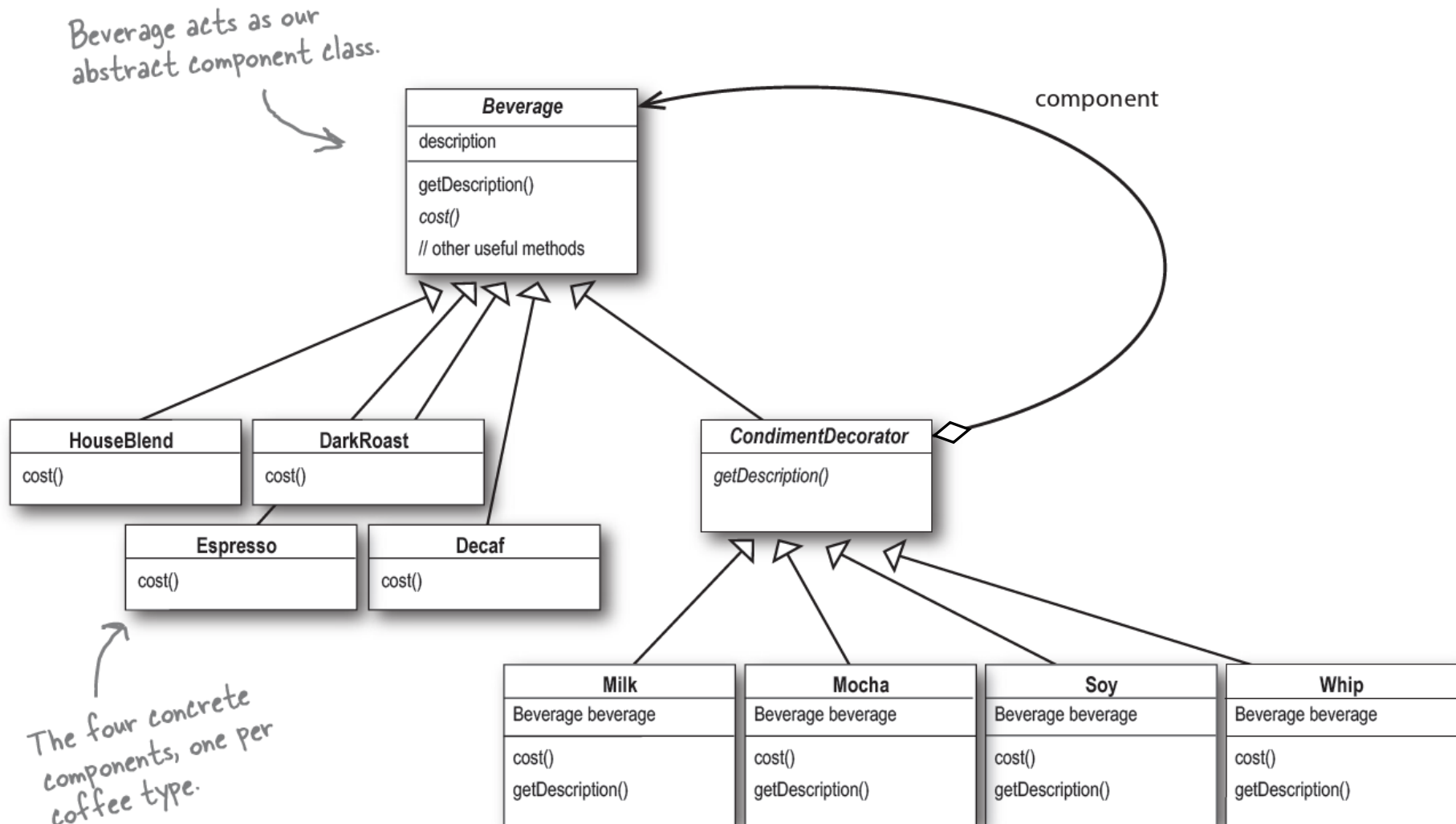
Each component can be used on its own, or wrapped by a decorator



Each decorator HAS_A (wraps) a component i.e. the decorator has an instance variable that holds a reference to a component

May include other object instances.

Decorator Example - StarBuzz Coffee



Decorator Example - StarBuzz Coffee

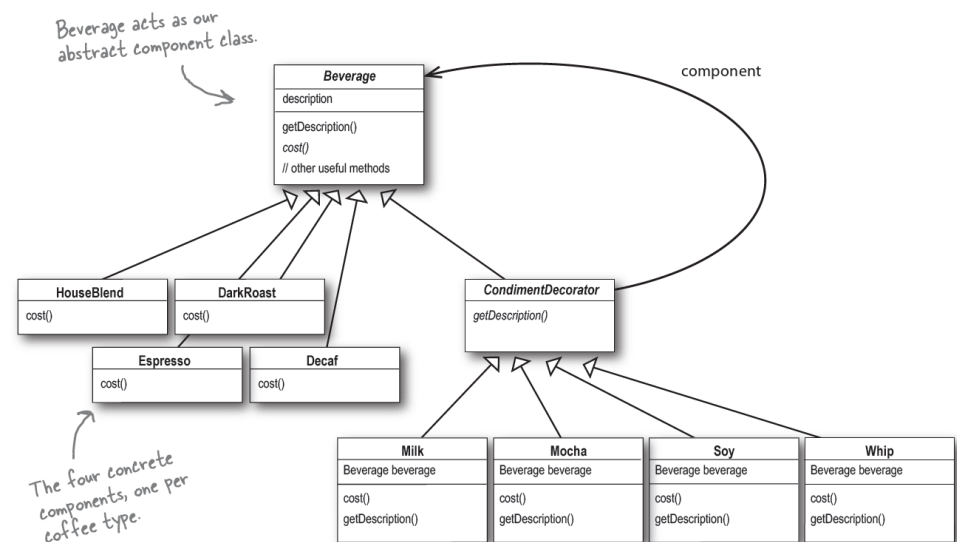
- **Discussion**

- It appears that the `condimentDecorator` is a subclass of `Beverage`
 - Really we are subclassing to get the correct type, not to inherit its behavior
 - We can mix and match decorators any way we like at runtime.

Decorator Example– StarBuzz Coffee

Beverage abstract class (Component):

```
public abstract class Beverage {  
  
    String description = "Unknown Beverage";  
  
    public String getDescription() { // already implemented  
        return description;  
    }  
    public abstract double cost(); // Need to implement cost()  
}
```



Decorator Example– StarBuzz Coffee

Condiments class (Decorator):

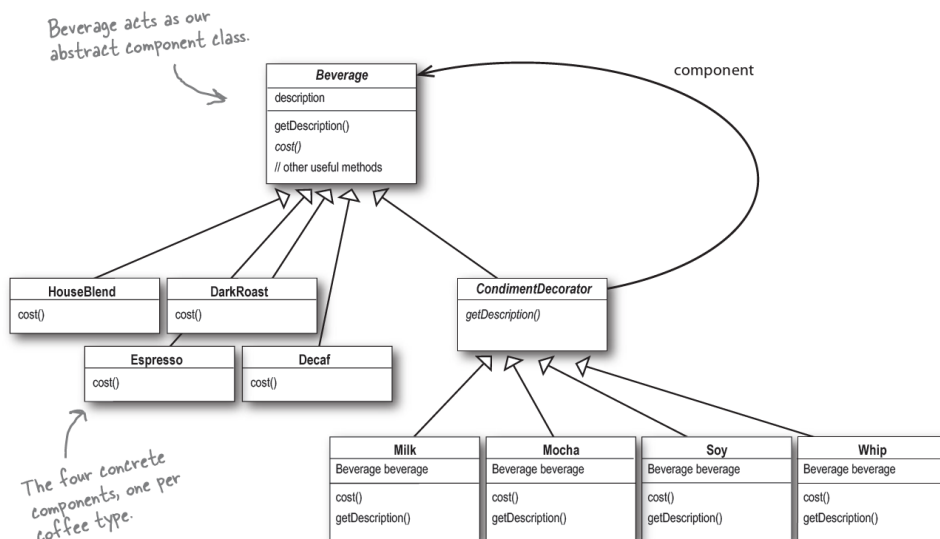
```
// We need to be interchangeable with a Beverage, so extend the  
// Beverage class - not to get its behavior but its type public
```

```
abstract class CondimentDecorator extends Beverage {
```

```
    public abstract String getDescription();
```

```
}
```

Here we require all the condiment decorators re-implement the `getDescription()` method. Why?



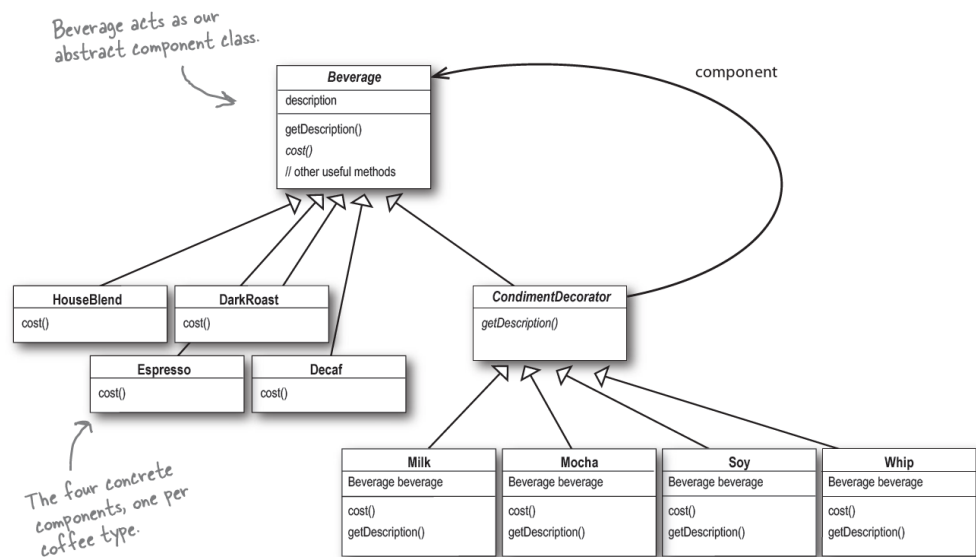
Decorator Example– StarBuzz Coffee

HouseBlend class (concrete component):

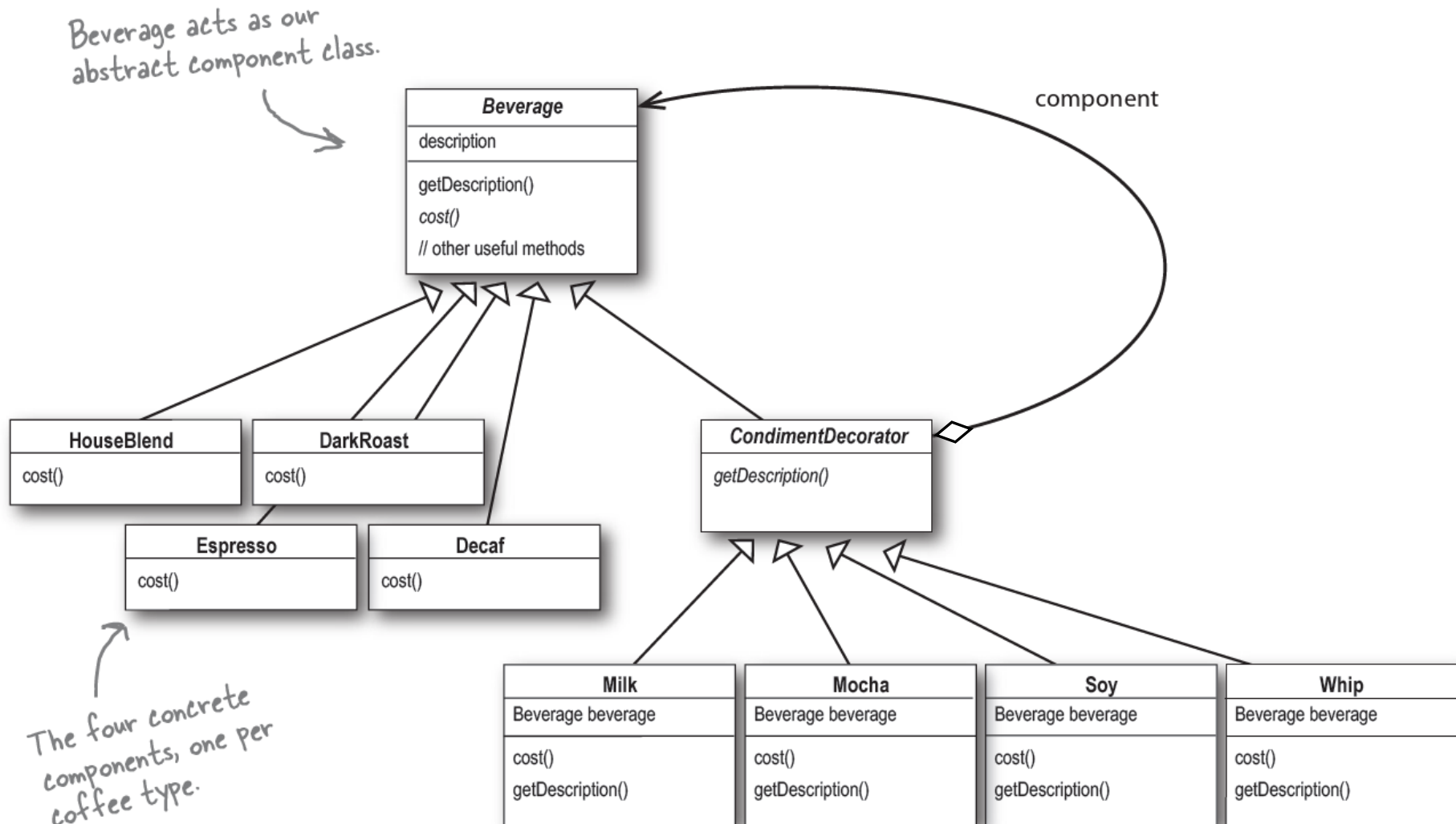
```
public class HouseBlend extends Beverage {  
  
    public HouseBlend() {  
        description = "House Blend Coffee";  
    }  
    public double cost() {  
        return .89;  
    }  
}
```

Note the description variable is inherited from Beverage. To take care of description, put this in the constructor for the class.

Compute the cost of a HouseBlend.
Need not to worry about condiments.



Decorator Example - StarBuzz Coffee



Decorator Example– StarBuzz Coffee

Mocha class (concrete decorator):

```
//Mocha is a decorator instance, which extends
//CondimentDecorator

public class Mocha extends CondimentDecorator {

    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription()+" Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

Instantiate Mocha with a reference to a Beverage using:

1. An instance variable to hold the beverage we are wrapping
2. A way to set this instance to the object we are wrapping – we pass the beverage we are wrapping to the decorator's constructor

Cost of condiment + cost of beverage

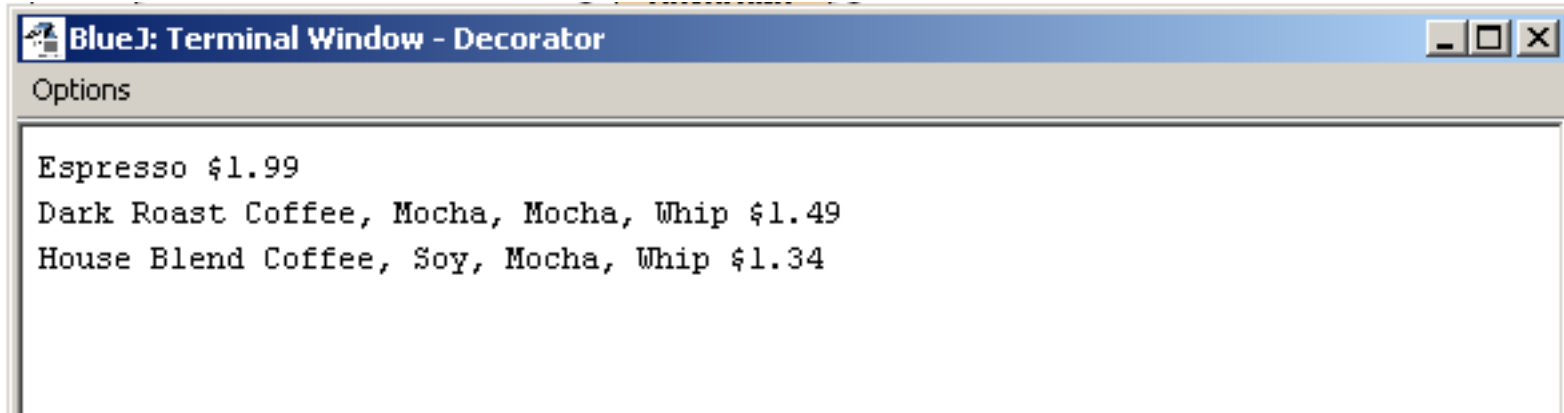
We want the description to include the beverage – say Houseblend – and the condiments

Decorator Example– StarBuzz Coffee

StarbuzzCoffee class (client):

```
public class StarbuzzCoffee {  
  
    public static void main(String args[]) {  
        Beverage beverage = new Espresso(); //espresso order,no condiments  
        System.out.println(beverage.getDescription()  
                            + " $" + beverage.cost());  
  
        Beverage beverage2 = new DarkRoast(); // get a DarkRoast  
        beverage2 = new Mocha(beverage2); // wrap it with Mocha  
        beverage2 = new Mocha(beverage2); // wrap it with Mocha  
        beverage2 = new Whip(beverage2); // wrap it with a Whip  
        System.out.println(beverage2.getDescription()  
                            + " $" + beverage2.cost());  
  
        Beverage beverage3 = new HouseBlend(); // get a Houseblend  
        beverage3 = new Soy(beverage3); // wrap with Soy  
        beverage3 = new Mocha(beverage3); // wrap with Mocha  
        beverage3 = new Whip(beverage3); // wrap with Whip  
        System.out.println(beverage3.getDescription()  
                            + " $" + beverage3.cost());  
    }  
}
```


Executing StarBuzzCoffee

A screenshot of a BlueJ terminal window titled "BlueJ: Terminal Window - Decorator". The window has a standard Mac OS-style title bar with minimize, maximize, and close buttons. Below the title bar is a tab labeled "Options". The main area of the window displays three lines of text in a monospaced font, representing the output of the StarBuzzCoffee program.

```
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
```

Another Decorator Pattern Example: Java IO

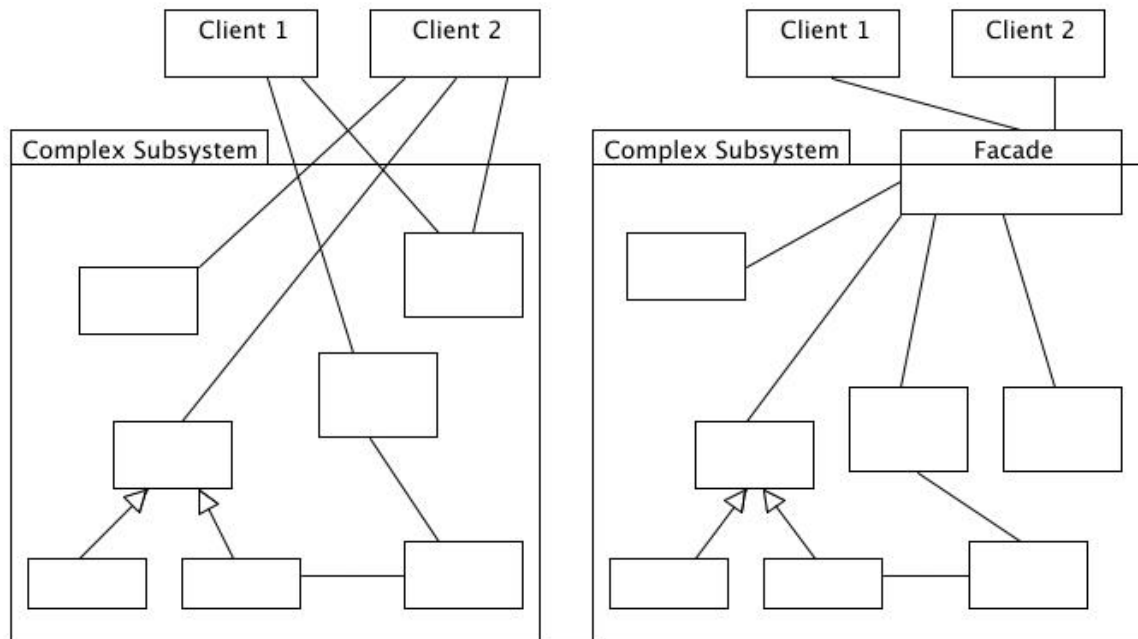
- `InputStream` class has only `public int read()` method to read one letter at a time.
- Decorators such as `BufferedReader` add functionality to read the stream more easily.

```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new InputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// With a BufferedReader decorator, read an
// entire line from the file in one call
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```

Façade Pattern

- The façade pattern provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

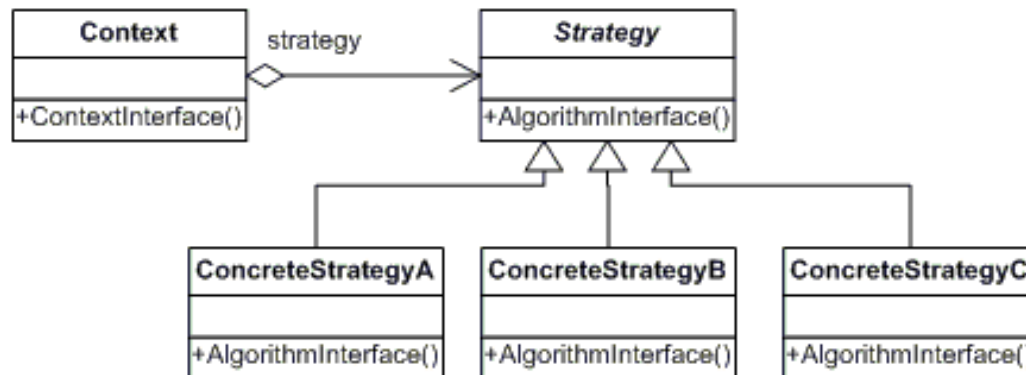


Kinds of Behavioral Patterns

- Strategy
 - Template method
 - Iterator
 - Null Object
 - ...
- Behavioral patterns identify and capture common patterns of communication between objects.

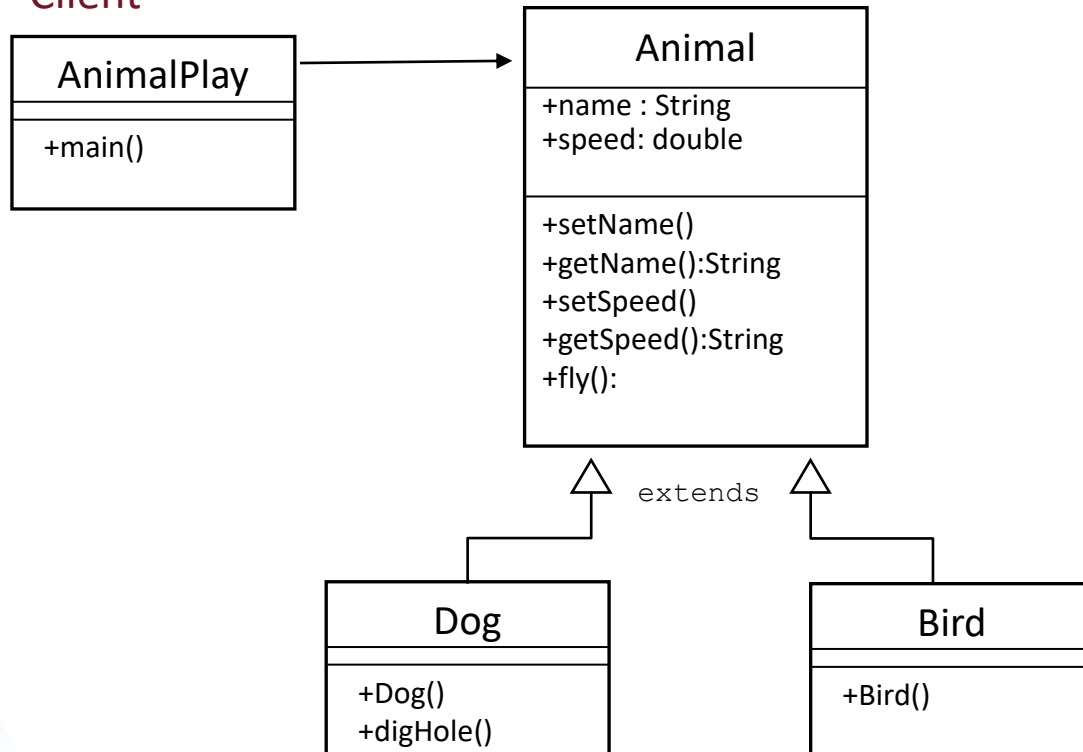
5. Strategy Pattern

- The Strategy Design Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
 - Strategy lets the algorithms changed independently from the clients that use it.
- Use the strategy pattern when you need to use one of several behaviors dynamically.



Strategy Pattern

Client



Strategy Example - Motivation

- Assume we want to add a method to animal that will tell whether the animal can fly.

```
public class Animal {  
  
    private String name;  
    private double speed;  
  
    public void setName(String newName){ name = newName; }  
    public String getName(){ return name; }  
  
    public void setSpeed(double newSpeed){ speed = newSpeed; }  
    public double getSpeed(){ return speed; }  
  
    public void fly(){  
        System.out.println("I'm flying");  
    }  
  
}
```

Strategy Example - Motivation

```
public class Dog extends Animal{
```

```
    public void digHole(){  
        System.out.println("Dug a hole");  
    }
```

```
    public Dog(){  
        super();  
        setSound("Bark");  
    }
```

Can't change the "flying" capability during run time!

```
    public void fly(){  
        System.out.println("I can't fly");  
    }  
}
```

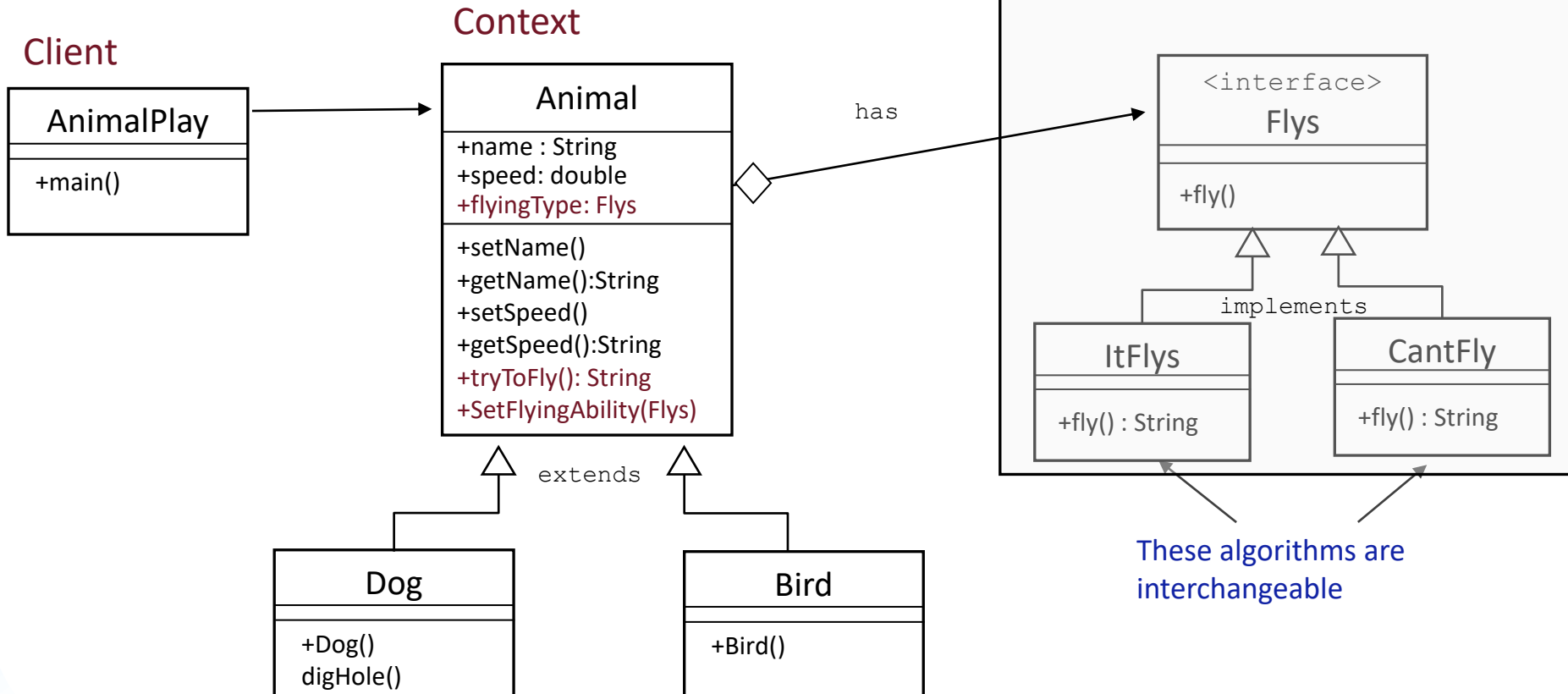
```
public class Bird extends Animal{
```

```
    public Bird(){  
        super();  
        setSound("Tweet");  
    }
```

```
}
```


Strategy Pattern

Think of each set of behaviors as a family of algorithms



Animal makes use of an encapsulated family of algorithms for flying

Strategy Example

Flys interface (Strategy):

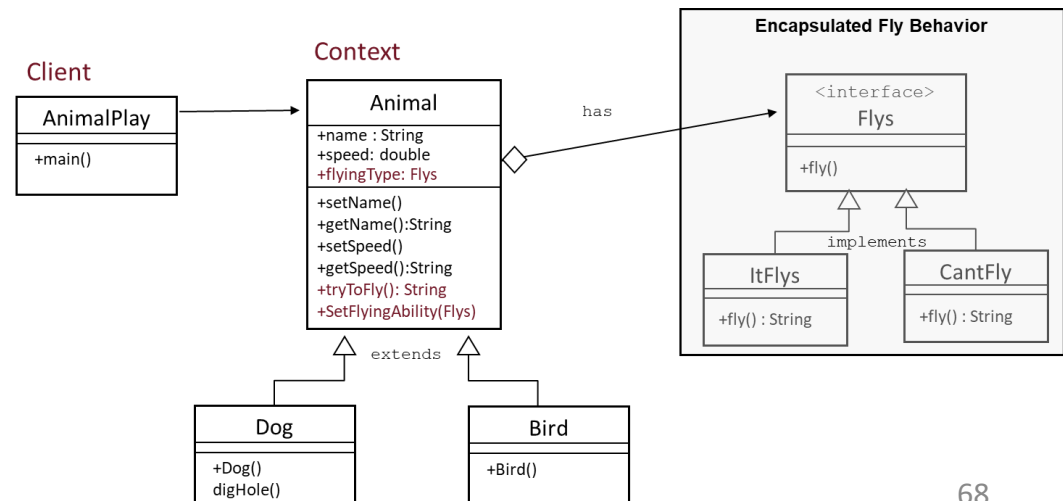
```
public interface Flys {  
    String fly();  
}
```

```
// Class ItFlys is used if the Animal can fly  
class ItFlys implements Flys{  
    public String fly() {  
        return "Flying High";  
    }  
}
```

```
//Class CantFly is used if the Animal can't fly  
class CantFly implements Flys{  
    public String fly() {  
        return "I can't fly";  
    }  
}
```

This approach eliminates
code duplication and
improves decoupling

The interface can be implemented by
many other subclasses that allow for
many types of flying without effecting
Animal, or Flys.



Strategy Example

Animal class (Context):

```
public class Animal {
```

```
    private String name;  
    private double speed;
```

```
    public Flys flyingType;
```

```
    public void setName(String newName) { name = newName; }  
    public String getName() { return name; }
```

```
    public void setSpeed(double newSpeed) { speed = newSpeed; }  
    public double getSpeed() { return speed; }
```

```
    public String tryToFly() {  
        return flyingType.fly();  
    }
```

```
    public void setFlyingAbility(Flys newFlyType) {  
        flyingType = newFlyType;  
    }
```

```
}
```

- Instead of using an interface in a traditional way we use an instance variable.
- Animal doesn't care what `flyingType` does, it just knows the behavior is available to its subclasses.
- This is known as Composition : Instead of inheriting an ability through inheritance the class is composed with Objects
- Composition allows you to change the capabilities of objects at run time!

Animal pushes off the responsibility for flying to `flyingType`

to change the `flyingType` dynamically
add this method

Strategy Example

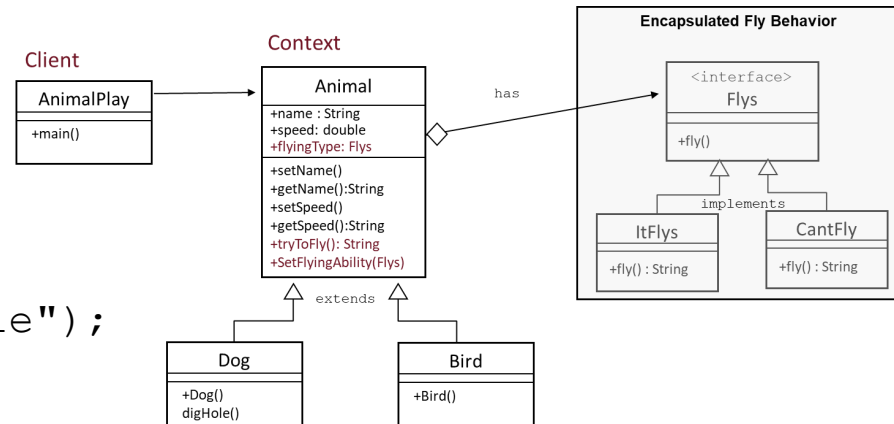
- Subclasses of Animal

```
public class Dog extends Animal{

    public void digHole(){
        System.out.println("Dug a hole");
    }
    public Dog(){
        super();
        setSound("Bark");
        flyingType = new CantFly();
    }
}

public class Bird extends Animal{

    public Bird(){
        super();
        setSound("Tweet");
        flyingType = new ItFllys();
    }
}
```



This sets the behavior as a non-flying Animal

This sets the behavior as a flying Animal

Strategy Example

AnimalPlay class (Client):

```
public class AnimalPlay{

    public static void main(String[] args){

        Animal sparky = new Dog();
        Animal tweety = new Bird();

        System.out.println("Dog: " + sparky.tryToFly());
        System.out.println("Bird: " + tweety.tryToFly());

        // This allows dynamic changes for flyingType
        sparky.setFlyingAbility(new ItFlies());
        System.out.println("Dog: " + sparky.tryToFly());

    }

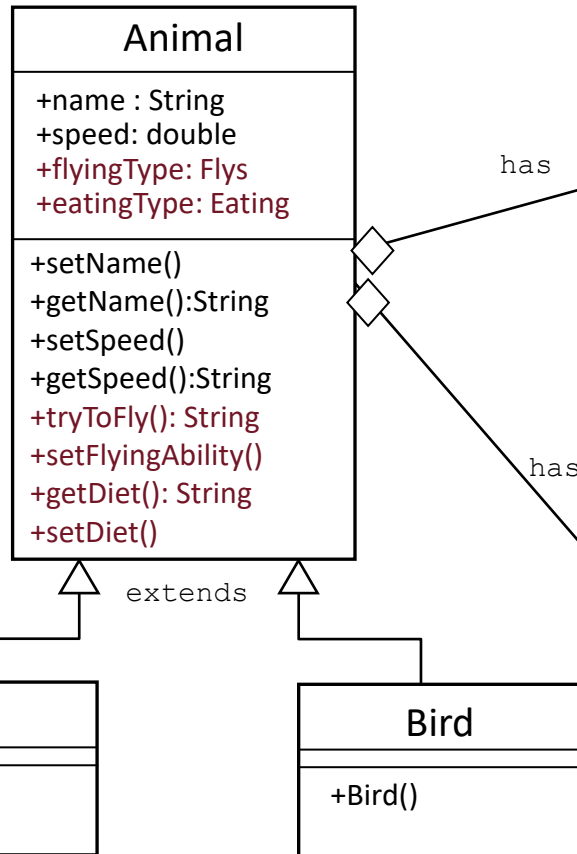
}
```

Output:

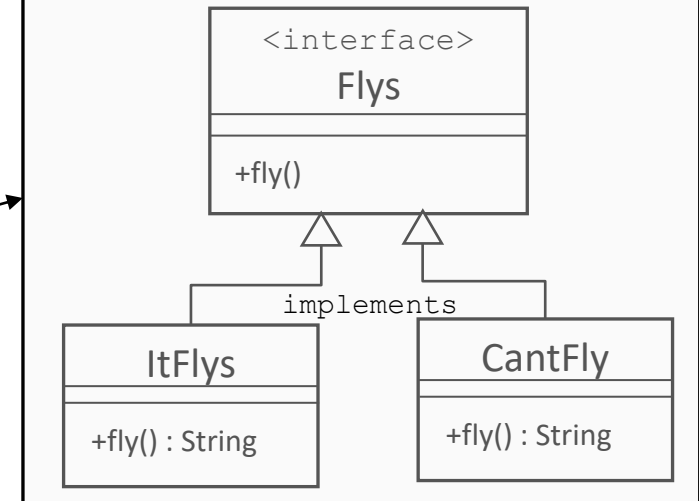
```
Dog: I can't fly
Bird: Flying High
Dog: Flying High
```

Strategy Pattern

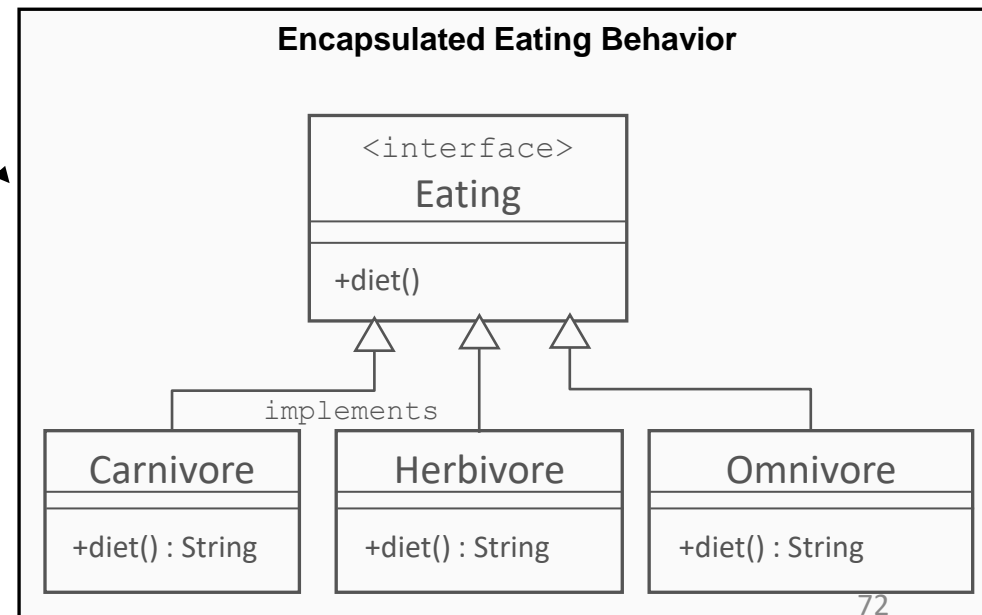
Context



Encapsulated Fly Behavior



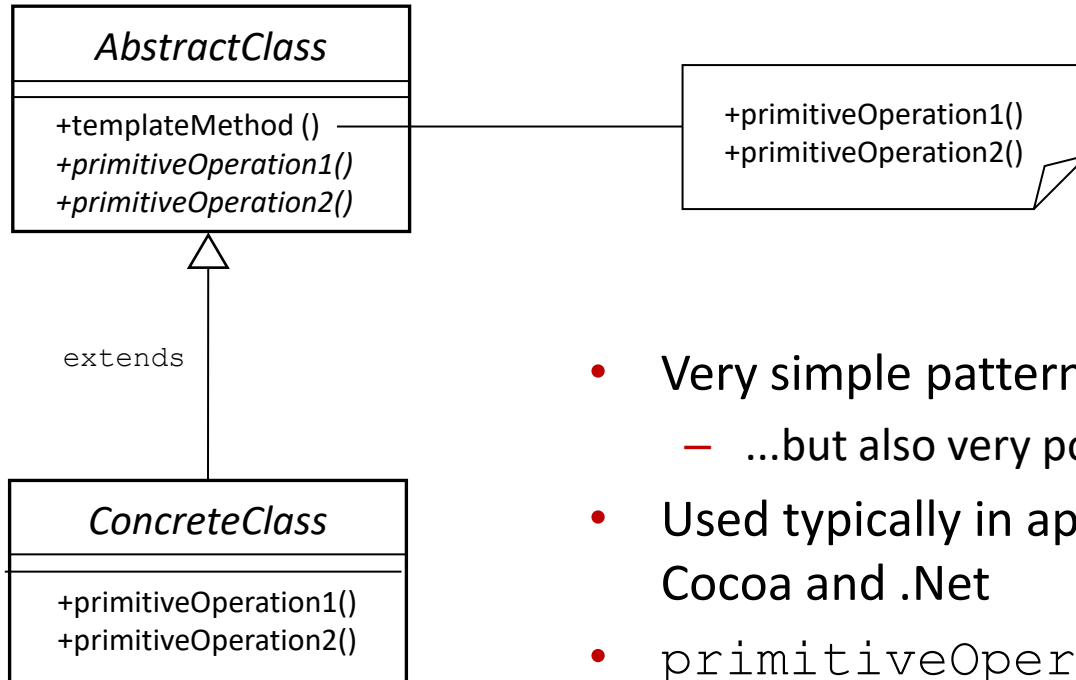
Encapsulated Eating Behavior



6. Template Method Pattern (skip)

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
 - Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps

Template Method Pattern



- Very simple pattern...
 - ...but also very powerful
- Used typically in application frameworks, e.g. Cocoa and .Net
- `primitiveOperation1()` and `primitiveOperation2()` are sometimes referred to as hook methods as they allow subclasses to hook their behavior into the service provided by `AbstractClass`

Template Method Example - Motivation

- Consider another Starbuzz example in which we consider the recipes for making coffee and tea
- Coffee:
 1. Boil water
 2. Brew coffee in boiling water
 3. Pour coffee in cup
 4. Add sugar and milk
- Tea:
 1. Boil water
 2. Steep tea in boiling water
 3. Pour tea in cup
 4. Add lemon

Template Method Example - Motivation

- **Coffee** implementation

```
public class Coffee {  
  
    public void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
  
    public void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

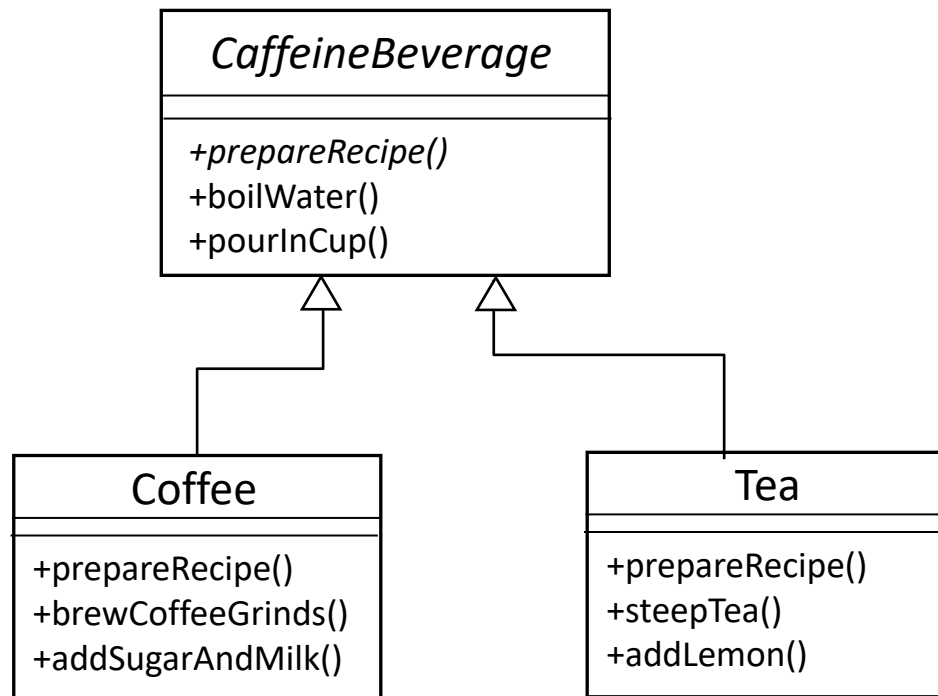
Template Method Example - Motivation

- **Tea** implementation

```
public class Tea {  
  
    public void prepareRecipe() {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea");  
    }  
  
    public void addLemon() {  
        System.out.println("Adding Lemon");  
    }  
  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Template Method Example - Motivation

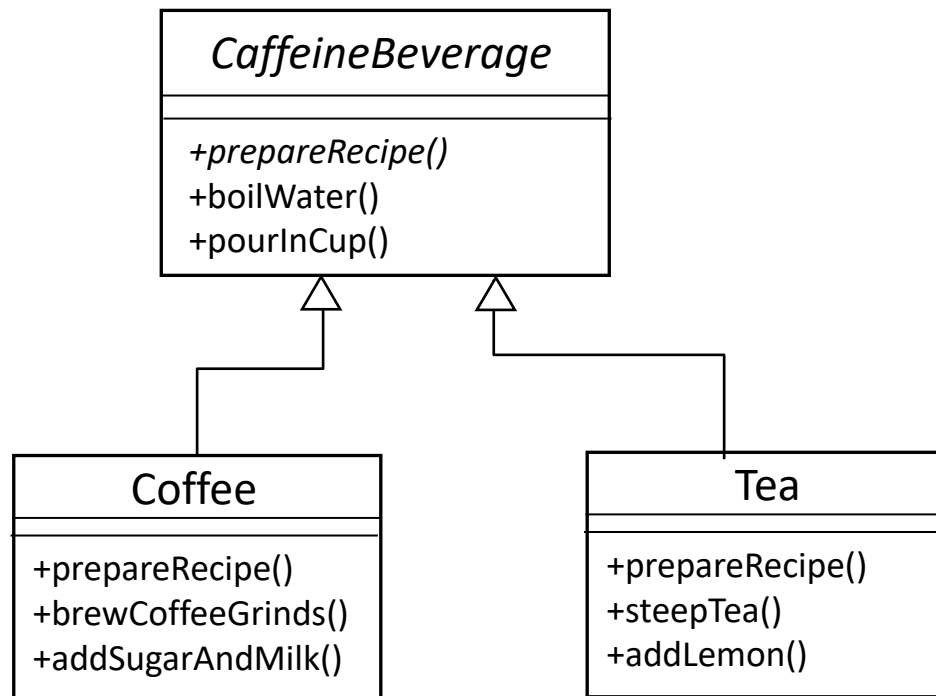
- Code duplication
 - We have code duplication occurring in these two classes
 - `boilWater()` and `pourInCup()` are exactly the same
 - Lets get rid of the duplication



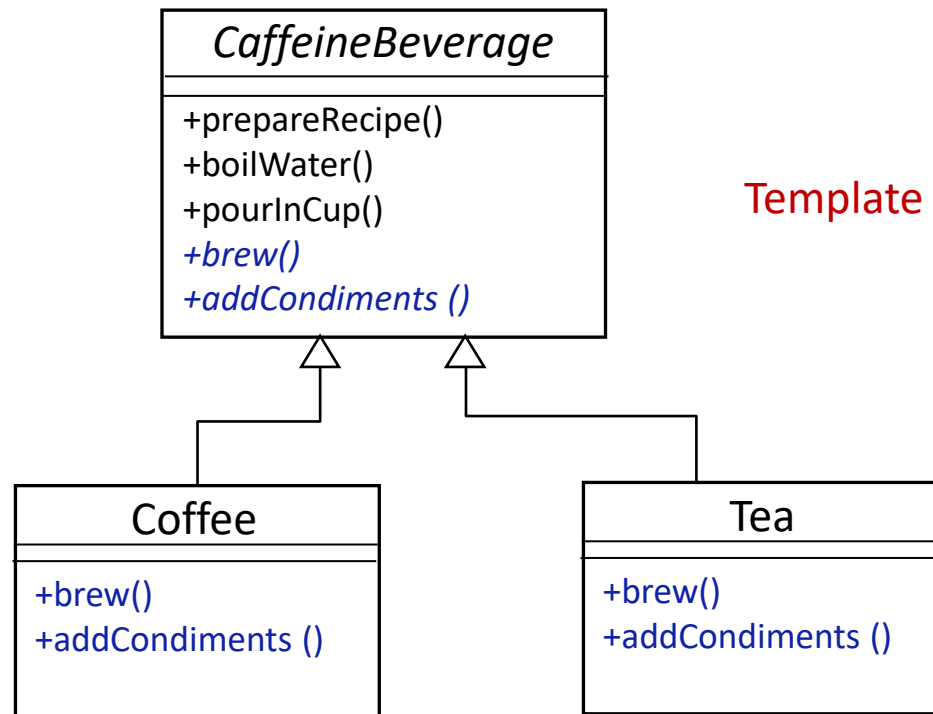
Template Method Example - Motivation

- Similar algorithms

- The structure of the algorithms in `prepareRecipe()` is similar for Tea and Coffee
- We can improve our code further by making the code in `prepareRecipe()` more abstract
 - `brewCoffeeGrinds()` and `steepTea()` → `brew()`
 - `addSugarAndMilk()` and `addLemon()` → `addCondiments()`



Template Method Example - Motivation



Template Method Pattern

Template Method Example

CaffeineBeverage class (Template):

```
public abstract class CaffeineBeverage {  
  
    public final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Template Method Example

CaffeineBeverage class (Template):

```
public abstract class CaffeineBeverage {  
  
    public final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
  
    abstract void brew();  
  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

Note: use of `final` keyword for `prepareReceipe()`

`brew()` and `addCondiments()` are abstract and must be supplied by subclasses

`boilWater()` and `pourInCup()` are specified and shared across all subclasses

Template Method Example

Coffee and Tea classes:

```
public class Coffee extends CaffeineBeverage {  
  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

```
public class Tea extends CaffeineBeverage {  
  
    public void brew() {  
        System.out.println("Steeping the tea");  
    }  
  
    public void addCondiments() {  
        System.out.println("Adding Lemon");  
    }  
}
```

What have we done?

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by introducing a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
 - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm... in a way that made sense for them

Comparison: Template Method (TM) vs. No TM

No Template Method

- `Coffee` and `Tea` each have own copy of algorithm
- Code is duplicated across both classes
- A change in the algorithm would result in a change in both classes
- Not easy to add new caffeine beverage
- Knowledge of algorithm distributed over multiple classes

Template Method

- `CaffeineBeverage` has the algorithm and protects it
- `CaffeineBeverage` shares common code with all subclasses
- A change in the algorithm likely impacts only `CaffeineBeverage`
- New caffeine beverages can easily be plugged in
- `CaffeineBeverage` centralizes knowledge of the algorithm; subclasses plug in missing pieces

Adding a Hook to Template Method

```
public abstract class CaffeineBeverageWithHook {
```

```
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        if (customerWantsCondiments()) {  
            addCondiments();  
        }  
    }
```

```
    abstract void brew();
```

```
    abstract void addCondiments();
```

```
    void boilWater() {  
        System.out.println("Boiling water");  
    }
```

```
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }
```

```
    boolean customerWantsCondiments() {  
        return true;  
    }
```

```
} 322 - Fall 2023
```

`prepareRecipe()` altered
to have a hook method:
`customerWantsCondiments()`

This method provides a method
body that subclasses can
override

To make the distinction between
hook and non-hook methods
more clear, you can add the
“final” keyword to all concrete
methods that you don’t want
subclasses to touch

Adding a Hook to Template Method

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();
        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;
        System.out.print("Would you like milk and sugar with coffee (y/n)? ");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) { return "no"; }
        return answer;
    }
}
```

Adding a Hook to
Coffee

Other Template Method Examples

- Template Method is used a lot since it's a great design tool for creating frameworks
 - the framework specifies how something should be done with a template method
 - that method invokes abstract hook methods that allow client-specific subclasses to “hook into” the framework and take advantage of its services
- Examples in the Java API
 - Sorting using `compareTo()` method
 - Frames in Swing
 - Applets