SOFTWARE ENGINEERING

Software Design

Design Patterns (Structural)

Course ID 06016410, 06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

2) Structural Design Patterns
Rationale

- Deal with the composition of classes or objects
- Address issues concerned with the way in which classes and objects are organized
  - Focus on inheritance, aggregation and composition
- For example, if an application is required to be extensible
  - Application can be structured in order to minimize effects of future change

2) Structural Design Patterns Catalogue (1)

- **Adapter pattern**: Convert the interface of a class into another interface that clients expect. The Adapter helps classes work together by overcoming incompatible interfaces.
- **Decorator pattern**: Attach additional responsibilities to an object dynamically. The Decorator provides a flexible alternative to subclassing for extending functionality.
- **Composite Pattern**: Compose objects into tree structures to represent part/whole hierarchies. The Composite lets clients treat individual objects and compositions of objects uniformly.

2) Structural Design Patterns Catalogue (2)

- **Façade pattern:** Provide a unified interface to a set of interfaces in a subsystem. The Facade defines a higher-level interface that makes the subsystem easier to use.
- **Proxy pattern:** Provide a surrogate or placeholder for another object to control access to it.
- **Bridge pattern:** De-couple an abstraction from its implementation so that the two can vary independently.
- **Flyweight pattern:** Use sharing of state to support large numbers of fine-grained objects efficiently.

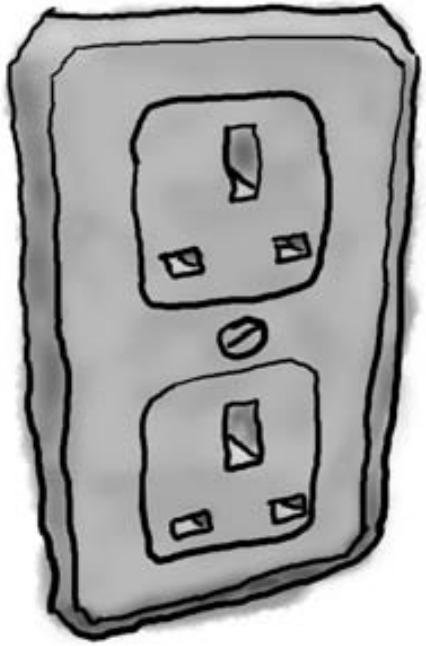
Adapter Pattern – Description (1)

- **Name:** Adapter, aka Wrapper/Translator pattern
- **Problem:** How can one design a class that allow classes with incompatible interfaces to work together? This design pattern translate one interface for a class into a compatible interface.
- **Context:** In some contexts, our application needs to be able to interact with all objects that have *different* methods.
  But, those methods have the same basic functionalities.
  It is necessary to have an Adapter object that translate methods of its interface into methods of the original interface.

Adapter Pattern – Description (2)

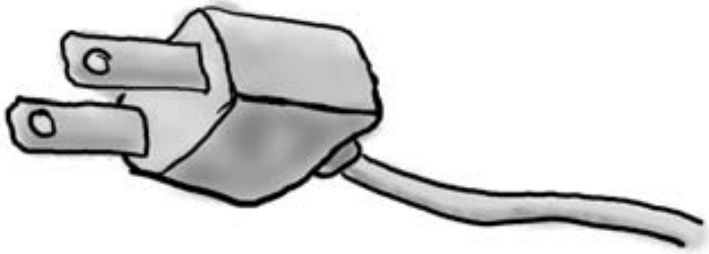
Adaptee **Adapter** **Client**

European Wall Outlet



AC Power Adapter

Standard AC Plug



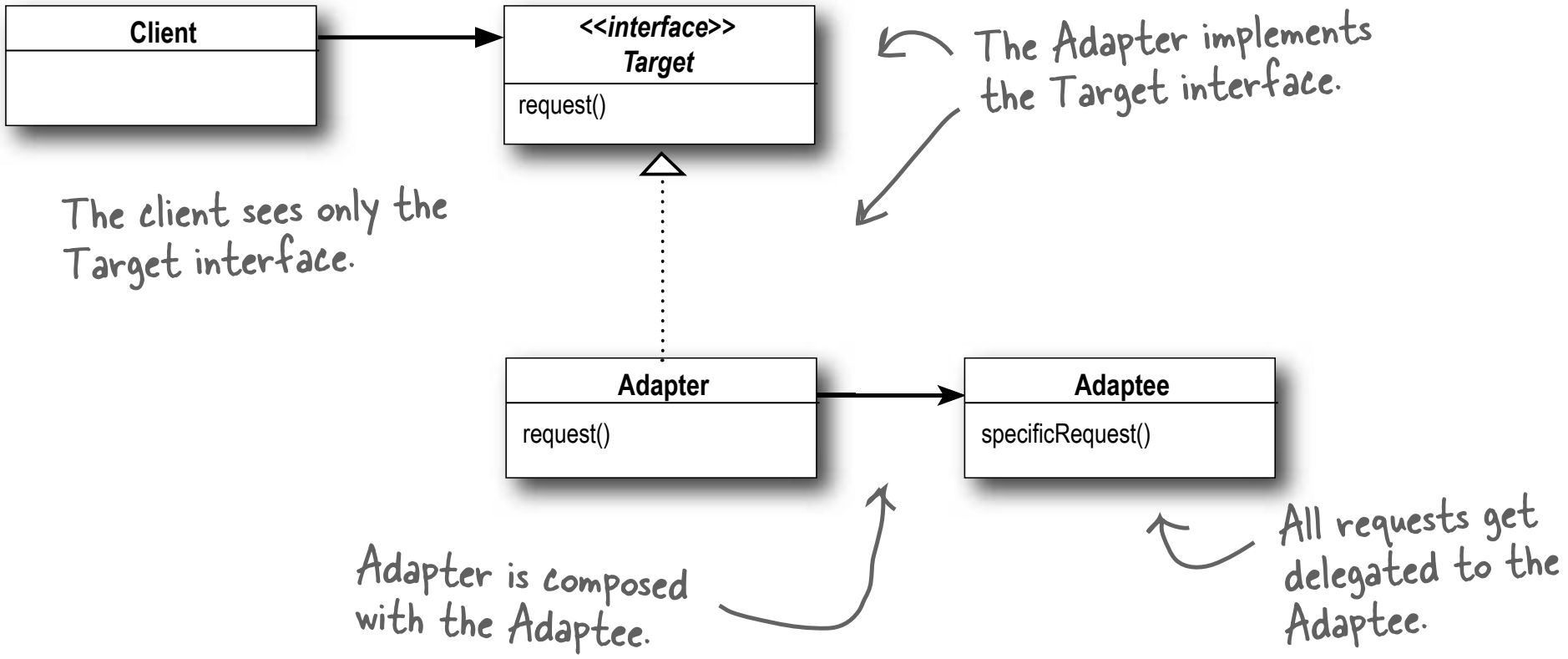
The US laptop expects another interface.

The European wall outlet exposes one interface for getting power.

The adapter converts one interface into another.

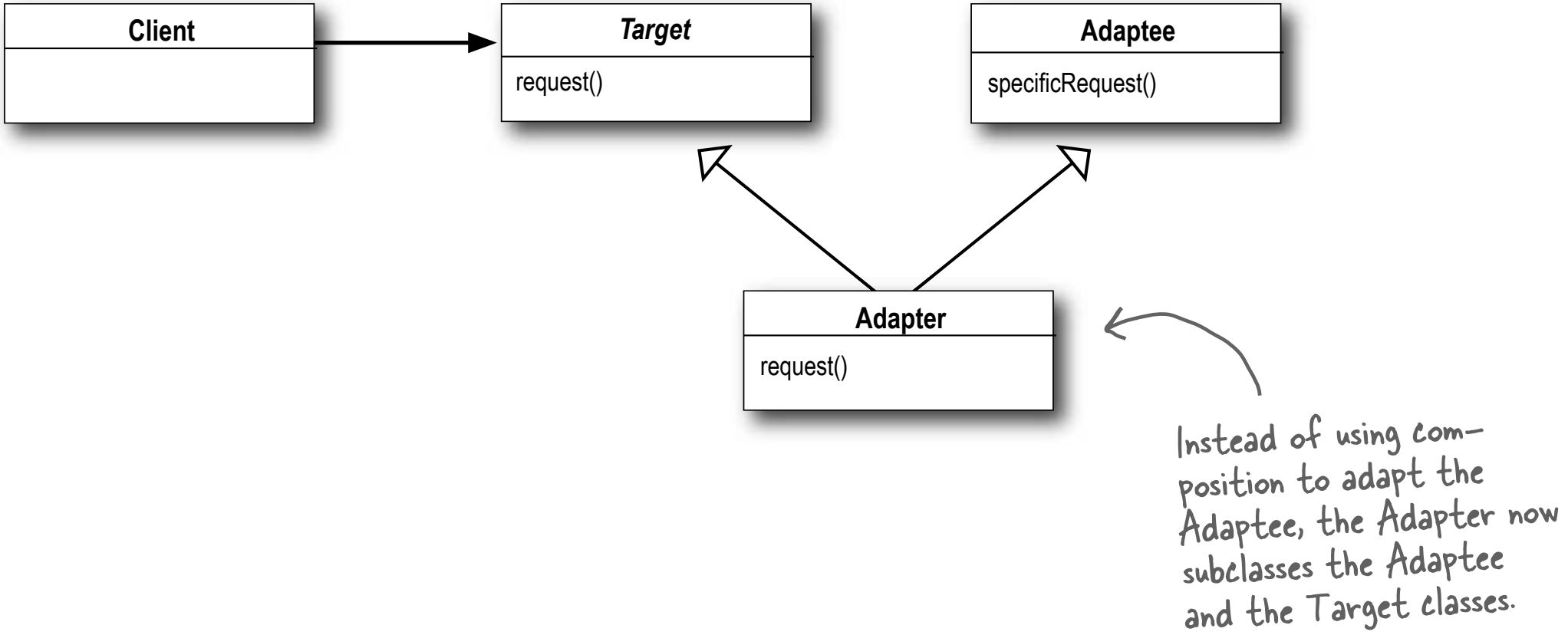
Adapter Pattern – Two Types (1)

1) **Object Adapter Pattern** – more universal

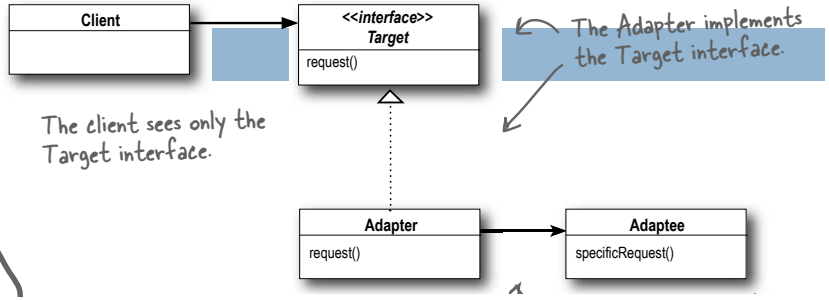


Adapter Pattern – Two Types (2)

2) Class Adapter Pattern – Java doesn't support multiple inheritance



Ex: Duck and Turkey (1)



Duck interface

```
public interface Duck {
    public void quack();
    public void fly();
}
```

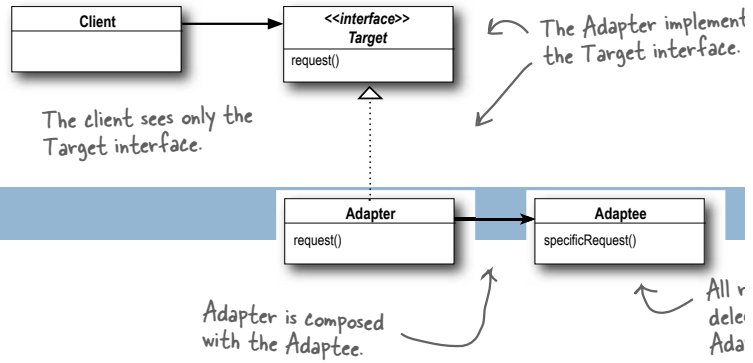
This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

MallardDuck class (subclass of Duck interface)

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

Ex: Duck and Turkey (2)



Turkey interface

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

WildTurkey class (subclass of Turkey interface)

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Ex: Duck and Turkey (3)

- **Aim**: assume that we are short of duck objects and want to use Turkey objects instead of duck ones.
- How do we use Turkey object since our class know only the way to use duck interface?

```
public class DuckTestDrive() {
    public static void main(String[] args) {
        Duck d = new MallardDuck();
        d.quack();
        d.fly();
    }
}
```

Client class

What shall we do if there was *no* MallardDuck object anymore after our using package had been patched?

Example: Duck and Turkey (4)

Use TurkeyAdapter that acts like a duck

TurkeyAdapter implements Duck interface.

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

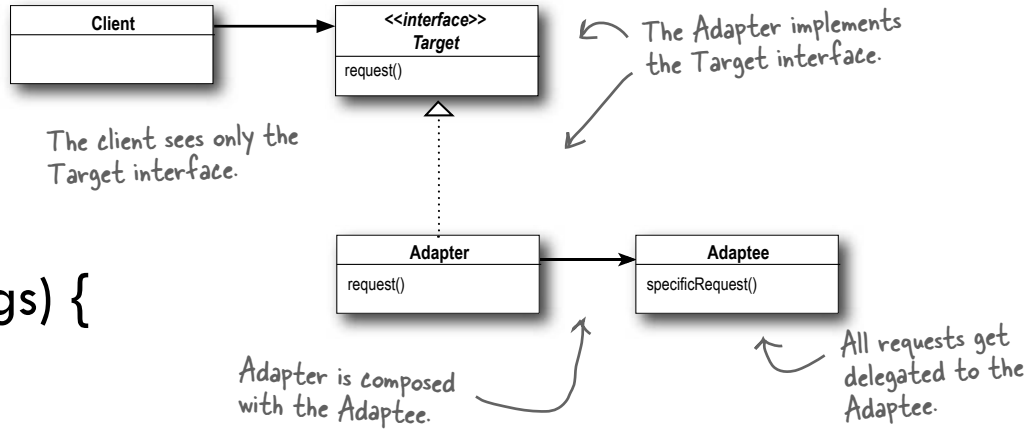
Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Example: Duck and Turkey (5)

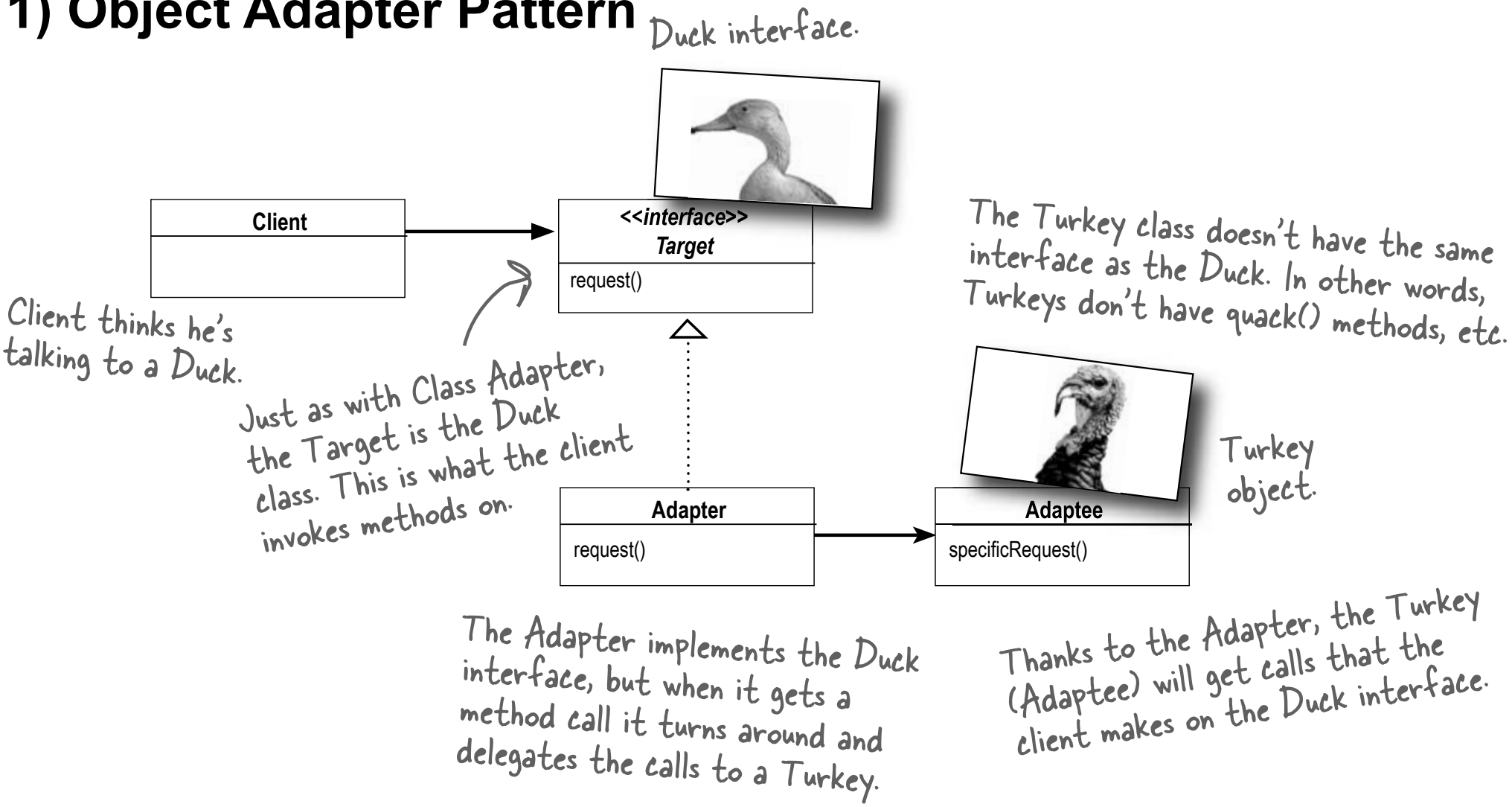
- Now, we can still use the duck interface with its method for the client class.




```
public class DuckTestDrive() {
    public static void main(String[] args) {
        Turkey t = new WildTurkey();
        Duck d = new TurkeyAdapter(t);
        d.quack(); //print "Gobble"
        d.fly();     //print "I'm flying a short distance" 5 times
    }
}
```

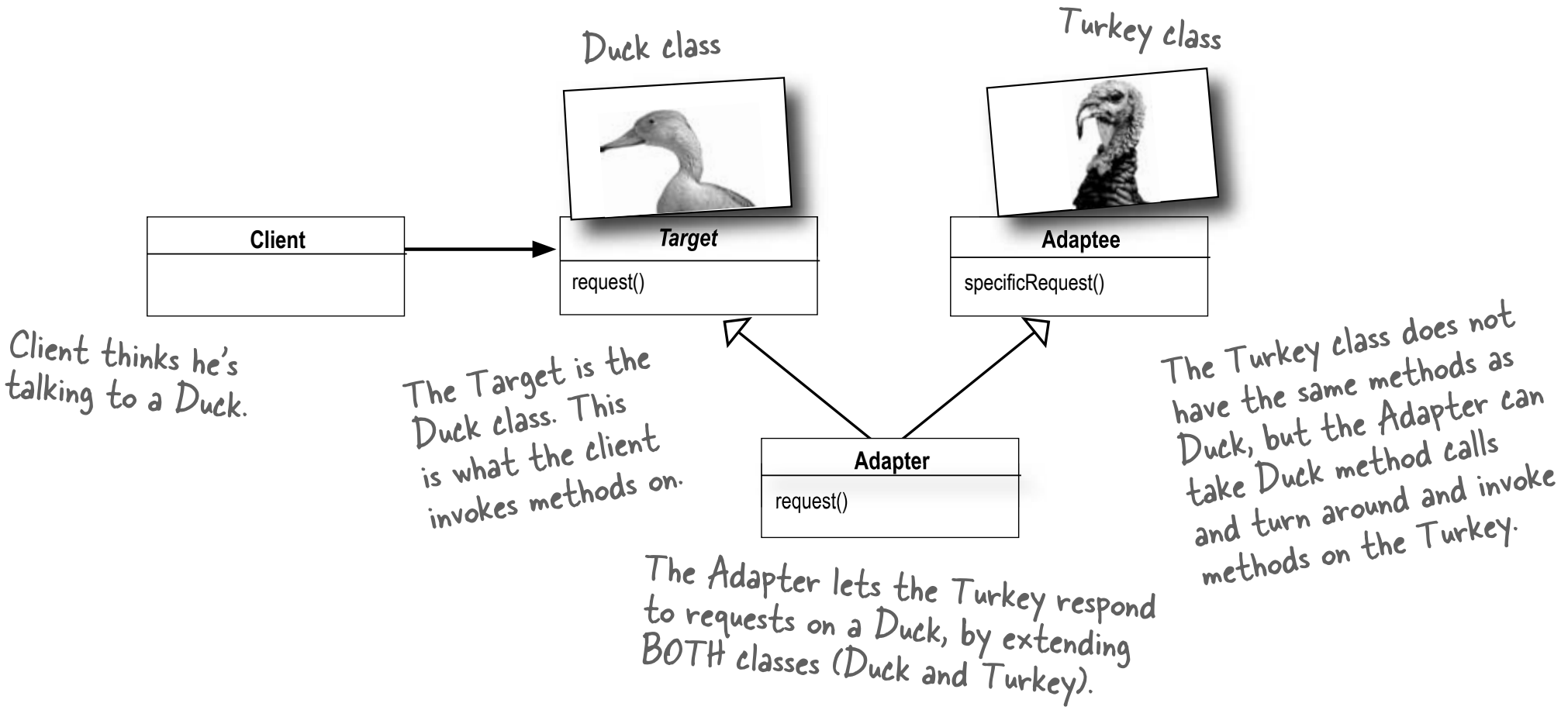
Duck & turkey in Object Adapter pattern?

1) Object Adapter Pattern



Duck & turkey in Class Adapter pattern?

2) Class Adapter Pattern



More Adaptees?

- How can we design classes if we have more adaptee?
  - For example: WildChicken, MountainEagle

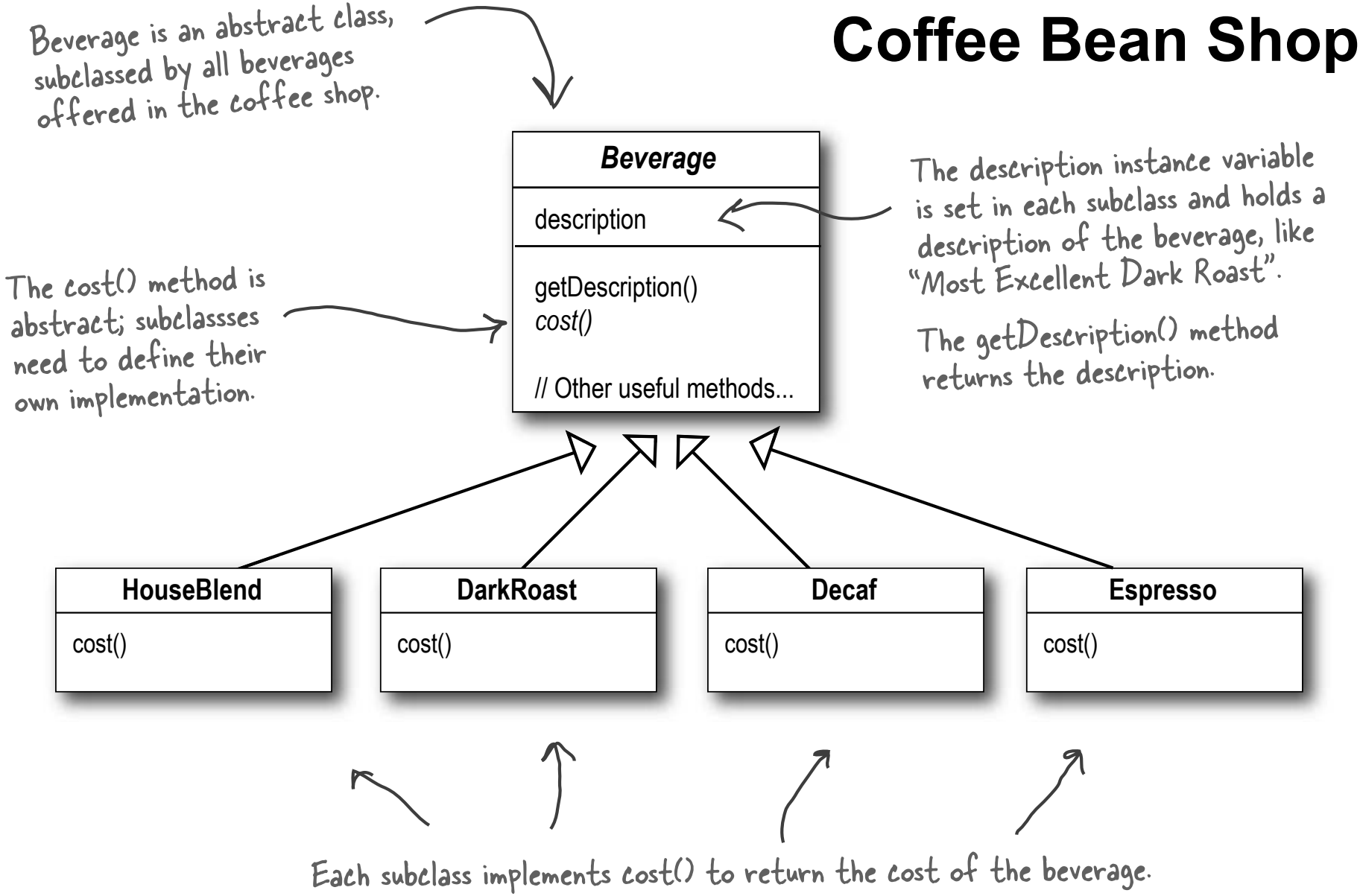
Adapter Pattern: Consequences

- Trade-offs
  - Object Adapter
    - hard to override Adaptee behavior
  - Class Adapter
    - a class adapter won't work when we want to adapt a class and all its subclasses

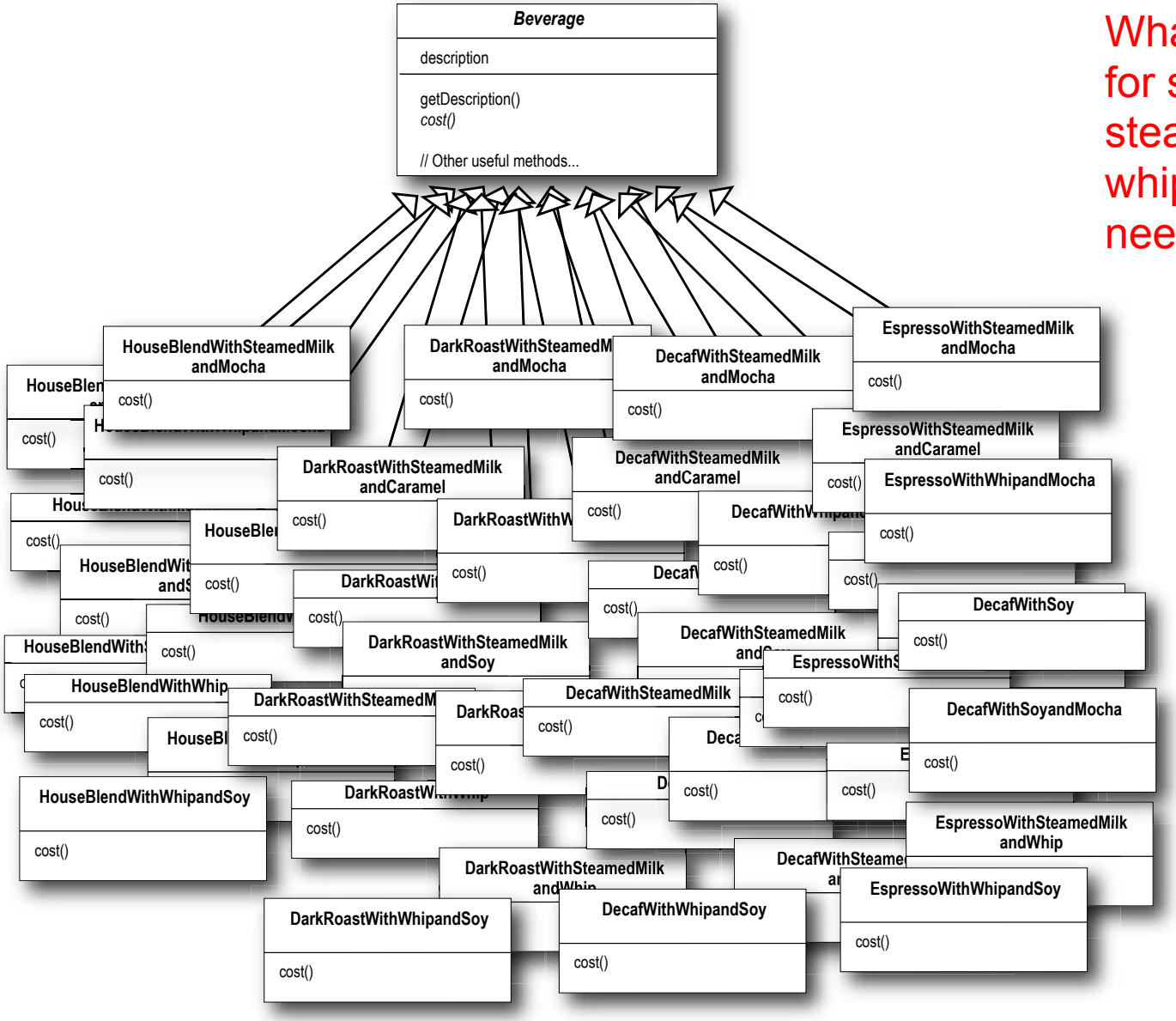
Decorator Pattern – Description (1)

- **Name:** Decorator pattern
- **Problem:** Do we overuse inheritance in our software design? How can one decorate many classes at runtime using a form of object composition?
- **Context:** Sometimes we want to add responsibilities to individual objects, not to an entire class.

Decorator Pattern – Example (1)



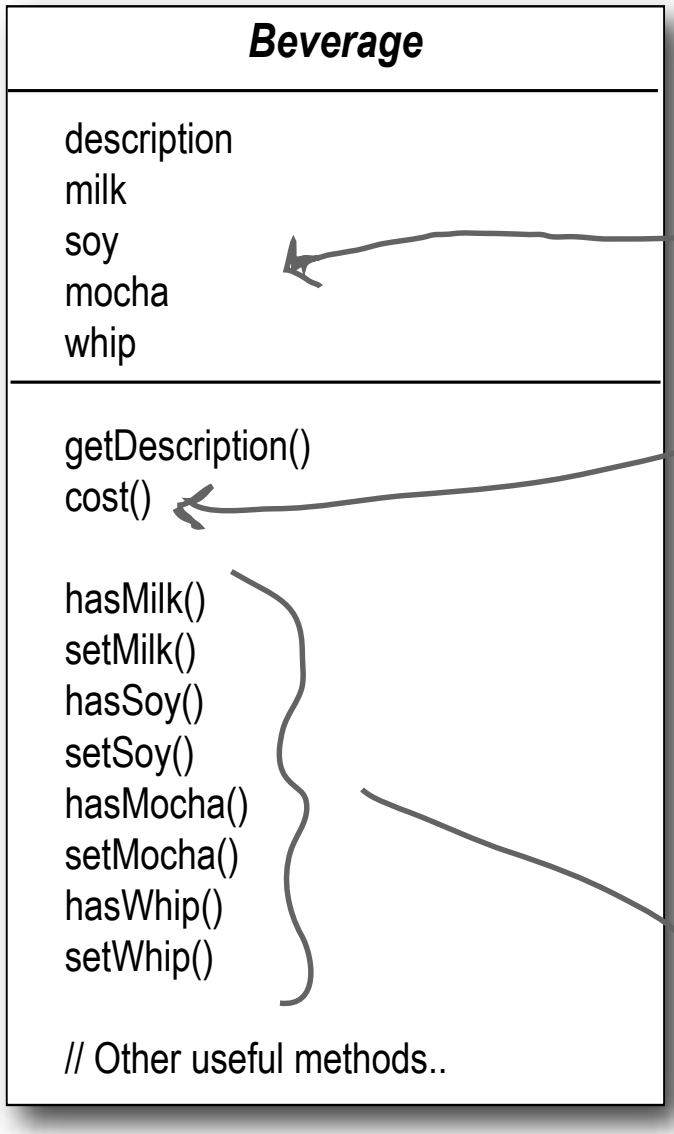
Example – Coffee Bean Shop (2)



What will happen if customers ask for several condiments, e.g. steamed milk, soy, mocha, or whipped cream, does the shop need put them all to the systems?

Each "**cost**" method computes the cost of the coffee along with the other condiments in the order.

Example – Coffee Bean Shop (3)



New boolean values for each condiment.

Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

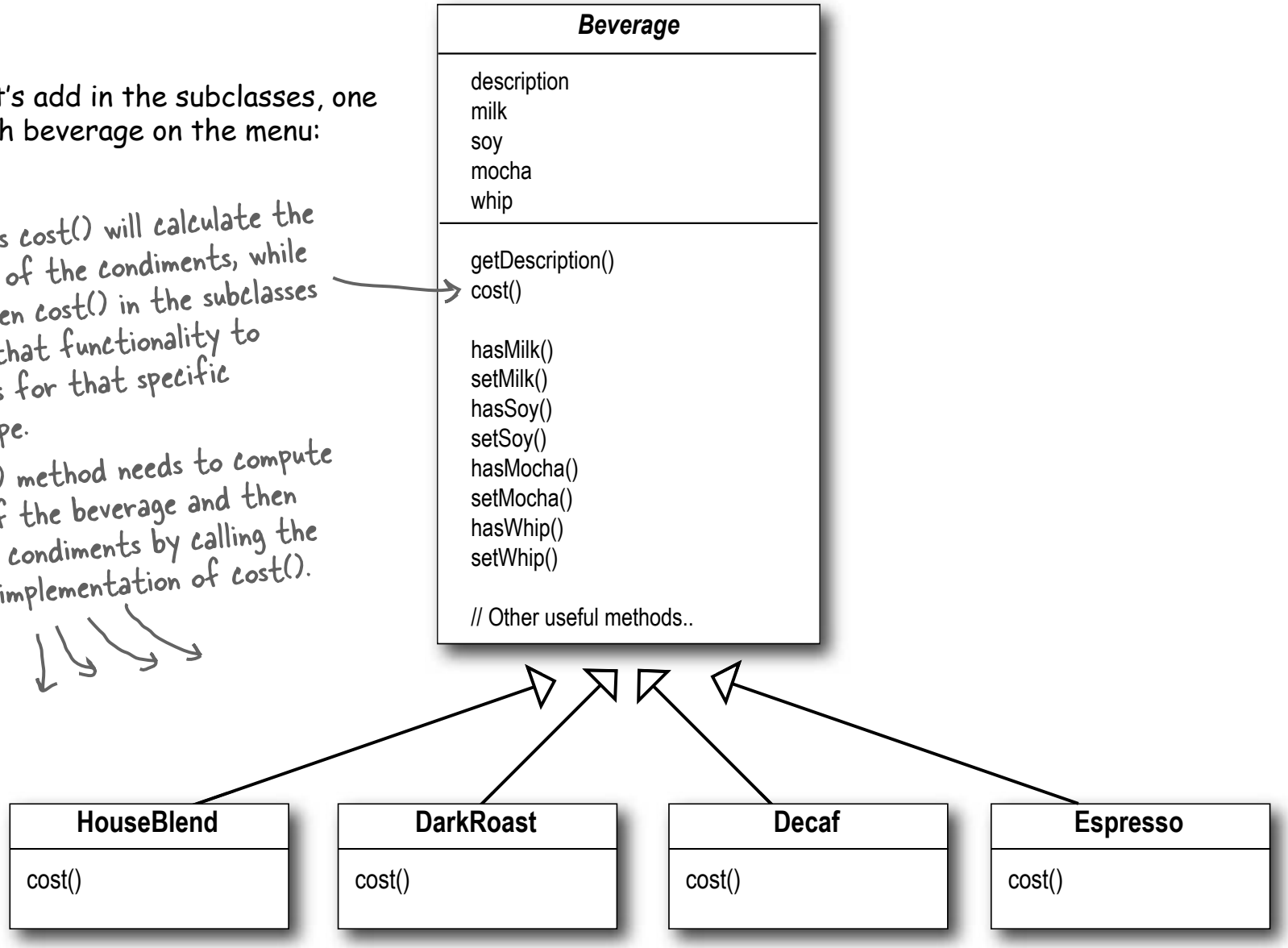
These get and set the boolean values for the condiments.

Example – Coffee Bean Shop (4)

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

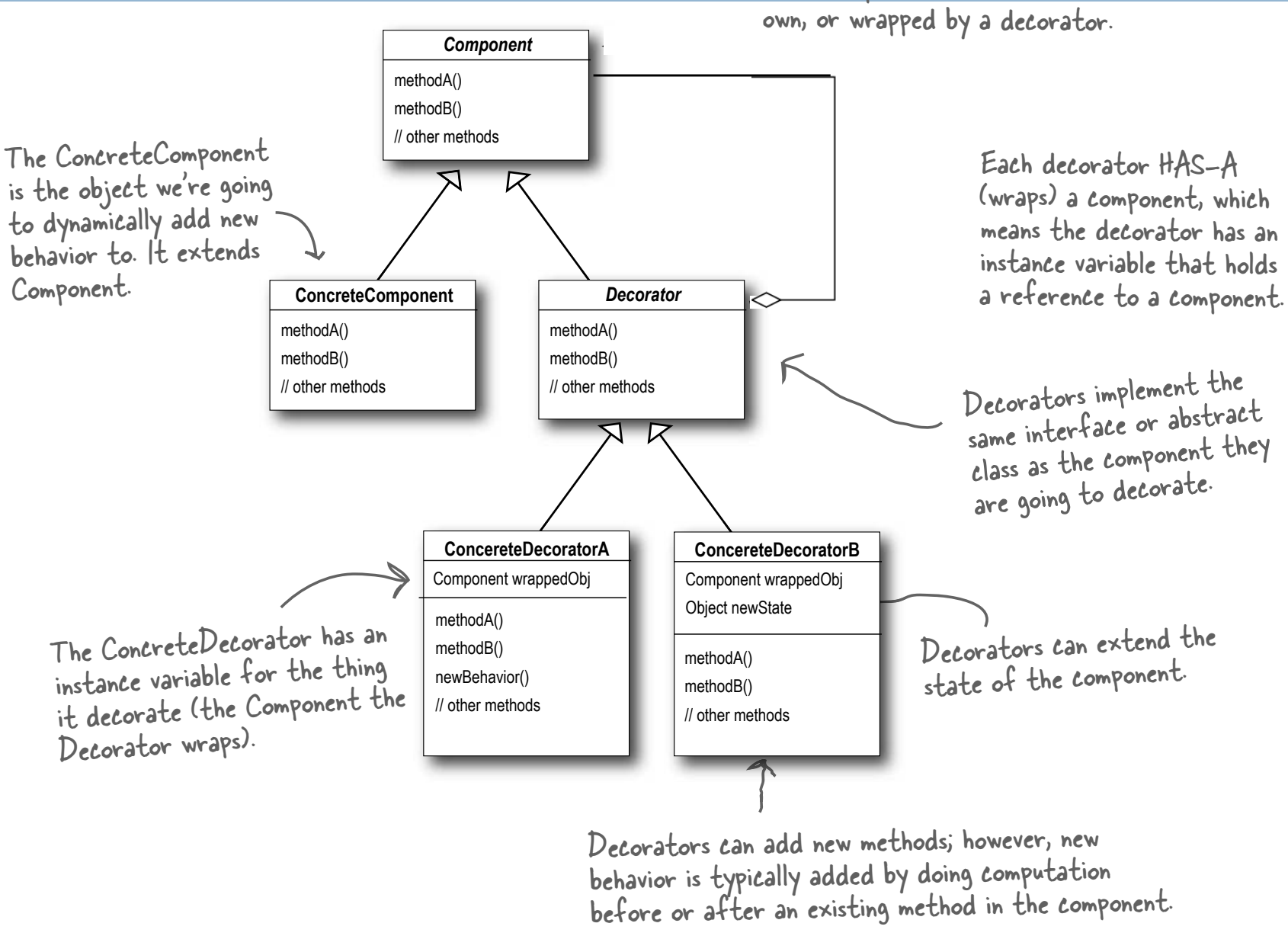


What's wrong with the previous design?

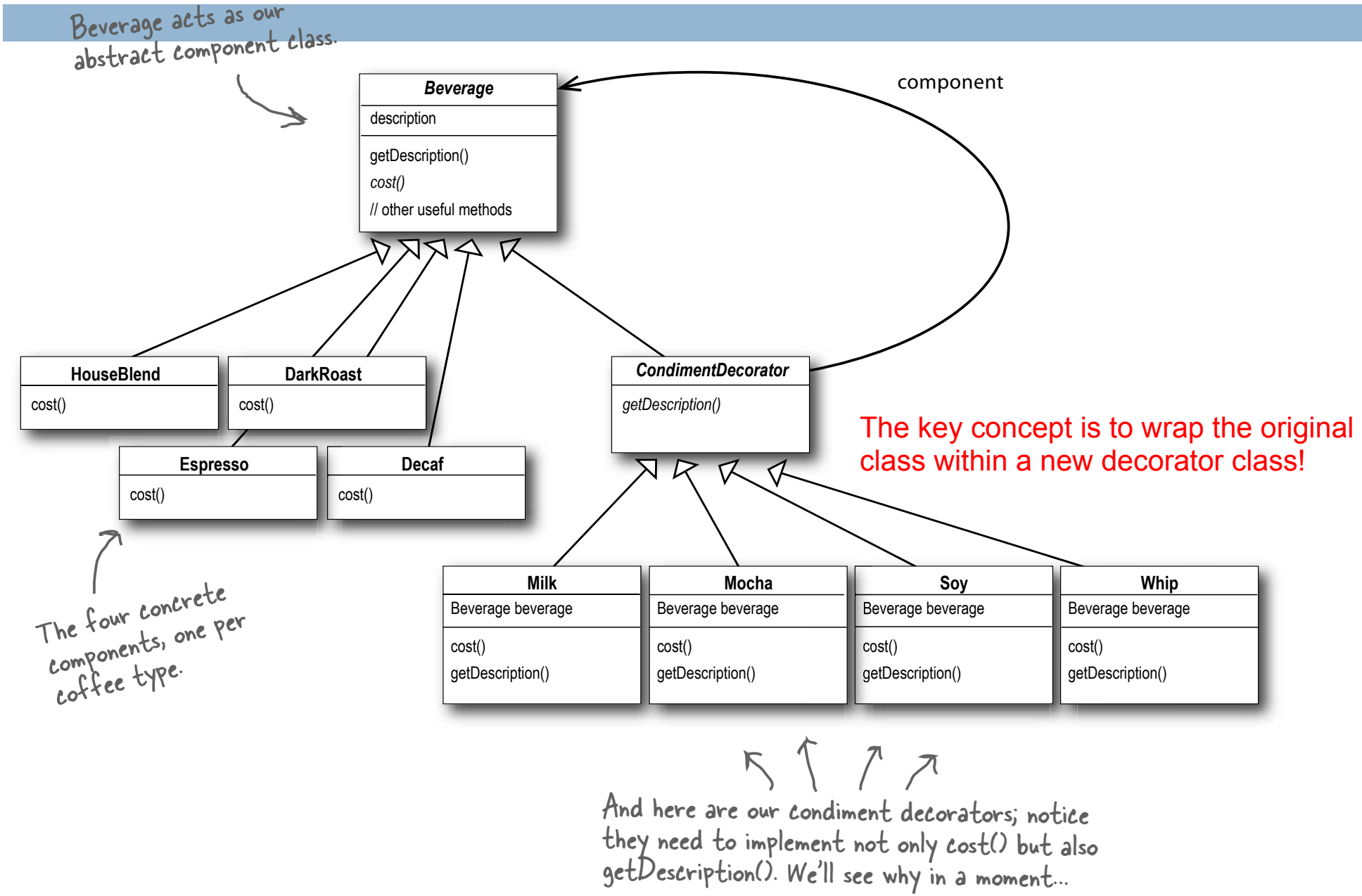
- Price changes for condiments will force us to alter existing code.
- New condiments will force us to add new methods and alter the cost method in the superclass.
- We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate. For example, the "Tea" subclass will still inherit methods like hasWhip().
- What if a customer wants a double mocha?

Class should be open for extension of new behaviors (methods), but closed for modification (need to keep existing codes correct)

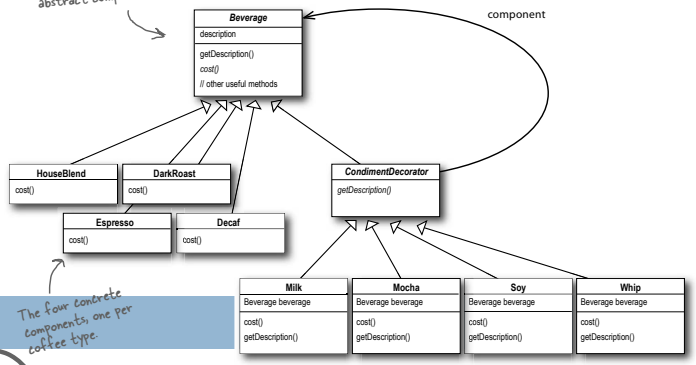
Decorator Pattern – Description (2)



Decorating our Coffee



Example code (1)




```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

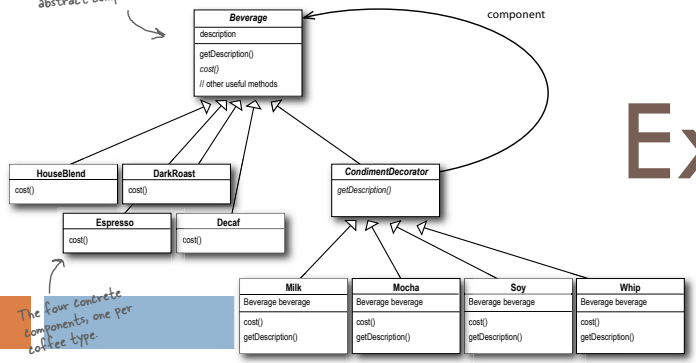
Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...



Example code (2)

Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99

Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

First we extend the Beverage class, since this is a beverage.

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

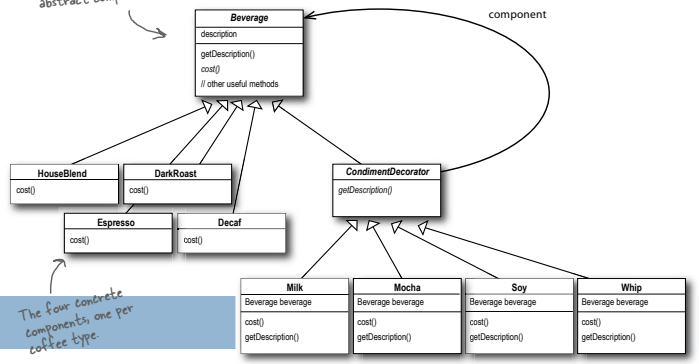
Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

You can create the other two Beverage classses (DarkRoast and Decaf) in exactly the same way.

Example code (3)



Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
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();
    }
}
```

We're going to 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 variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Example code (4)

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

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



```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

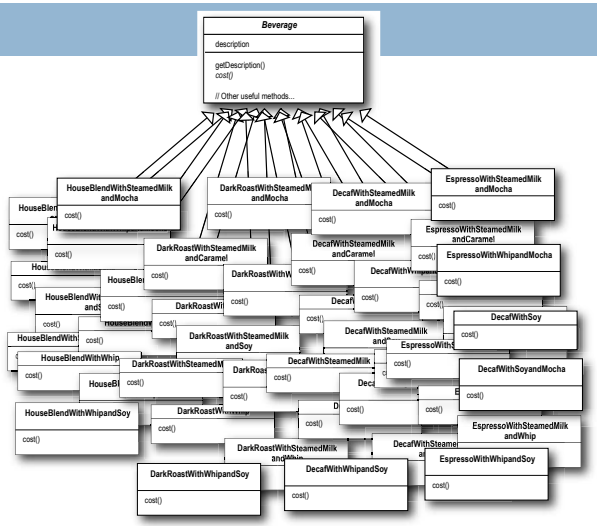
Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

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

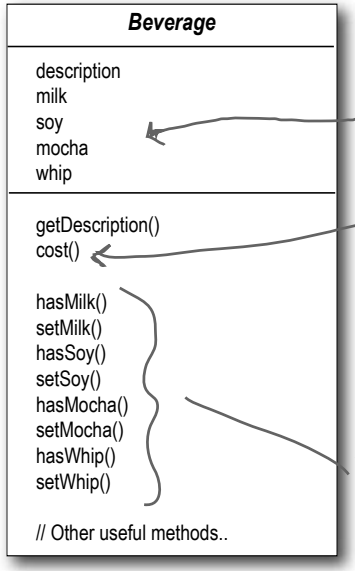
Decorator Pattern: Consequences

- More flexibility than static inheritance.



- Avoids feature-laden classes high up in the hierarchy.

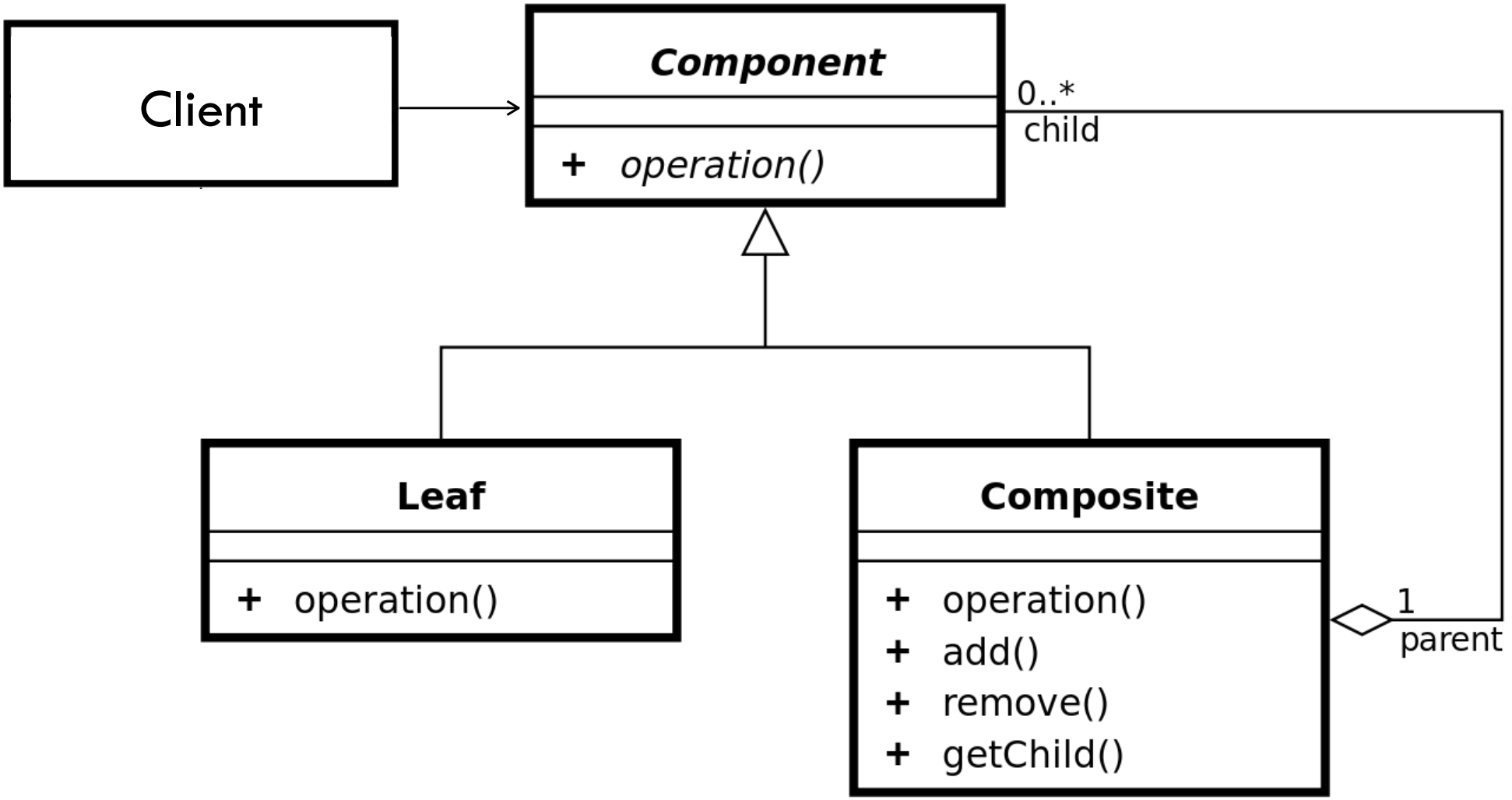
- Lots of little objects.



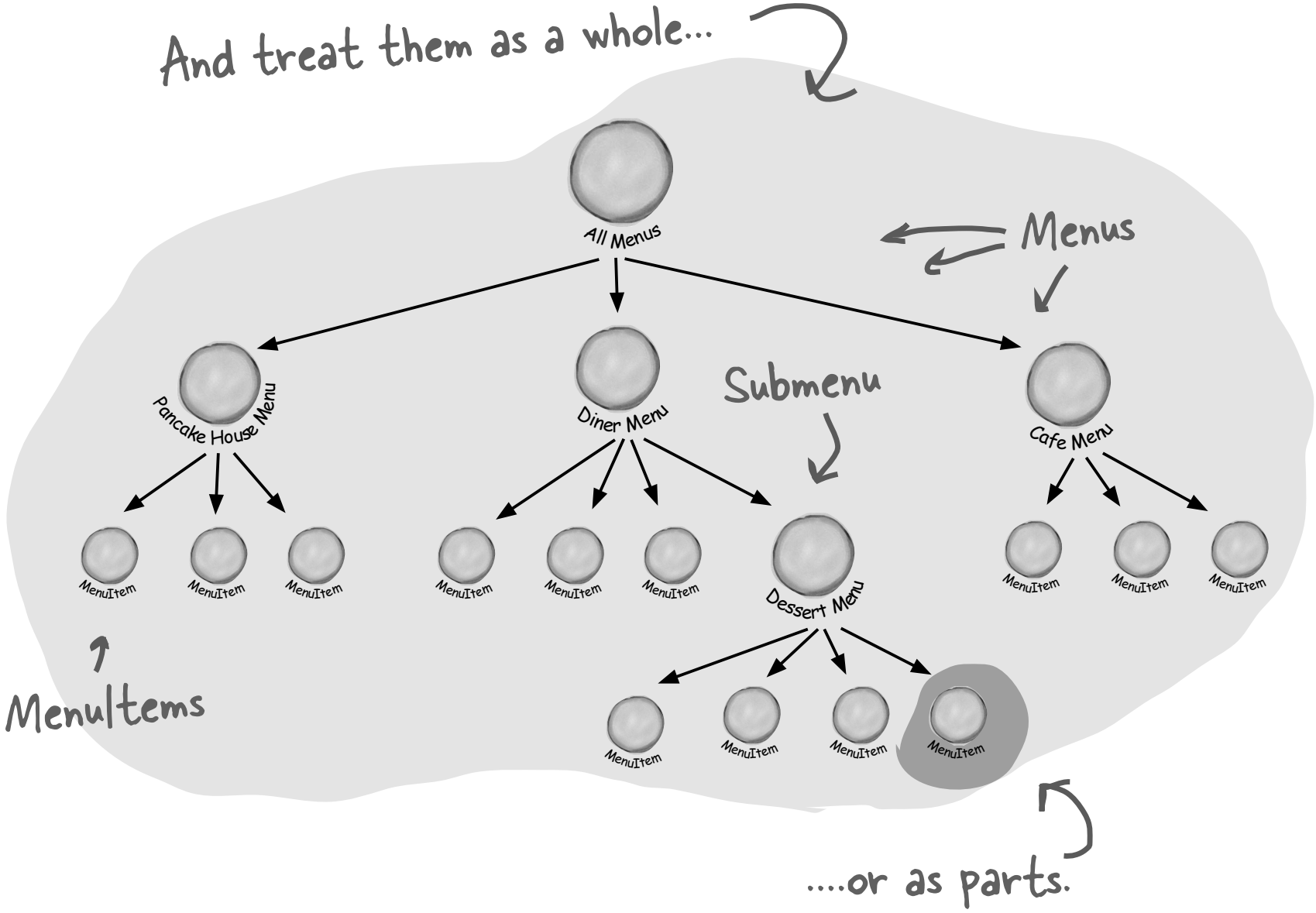
Composite Pattern – Description (1)

- **Name:** Composite pattern
- **Problem:** How can one treat a group of objects in the same way as a single instance of an object?
- **Context:** When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly.

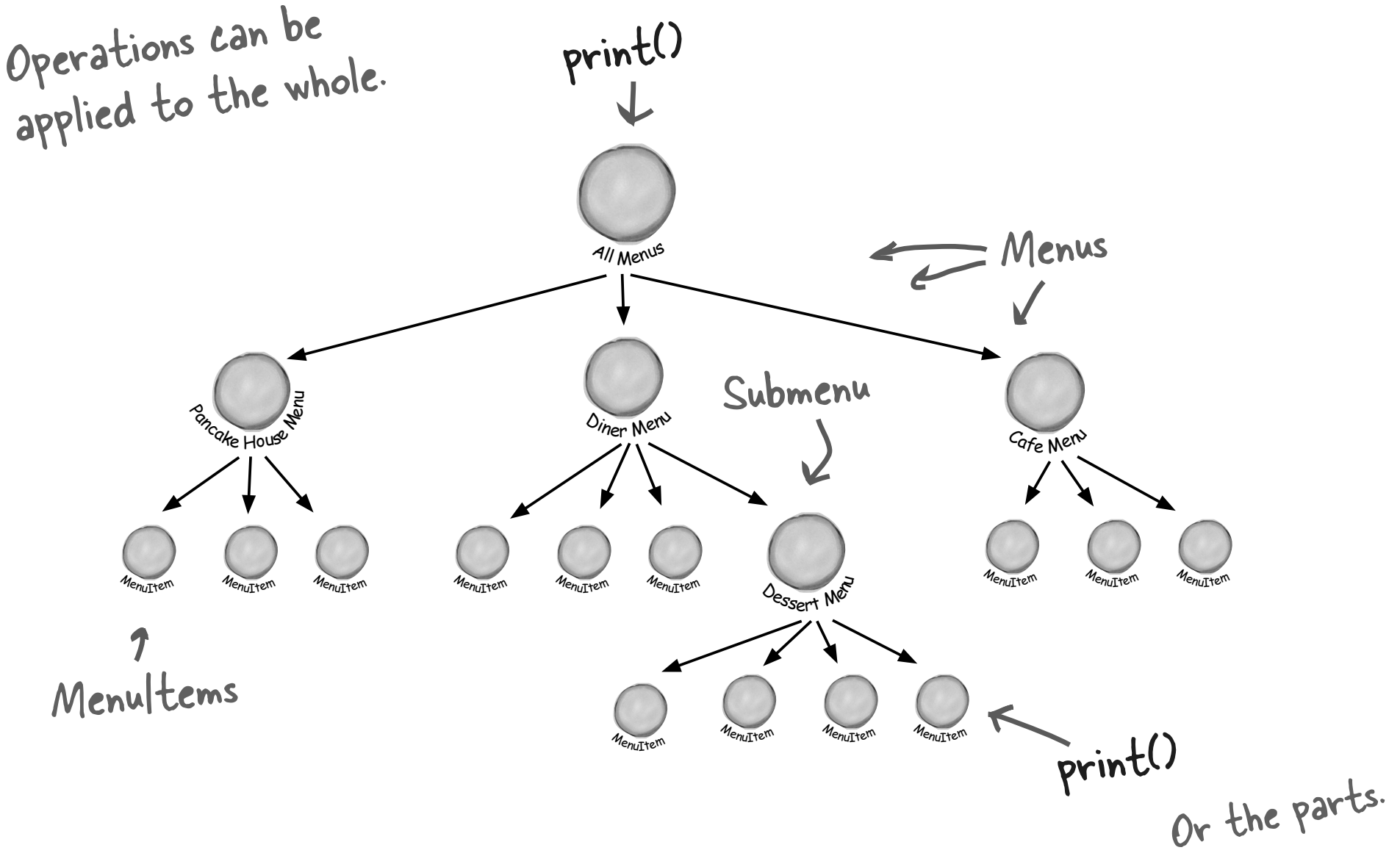
Composite Pattern – Description (2)



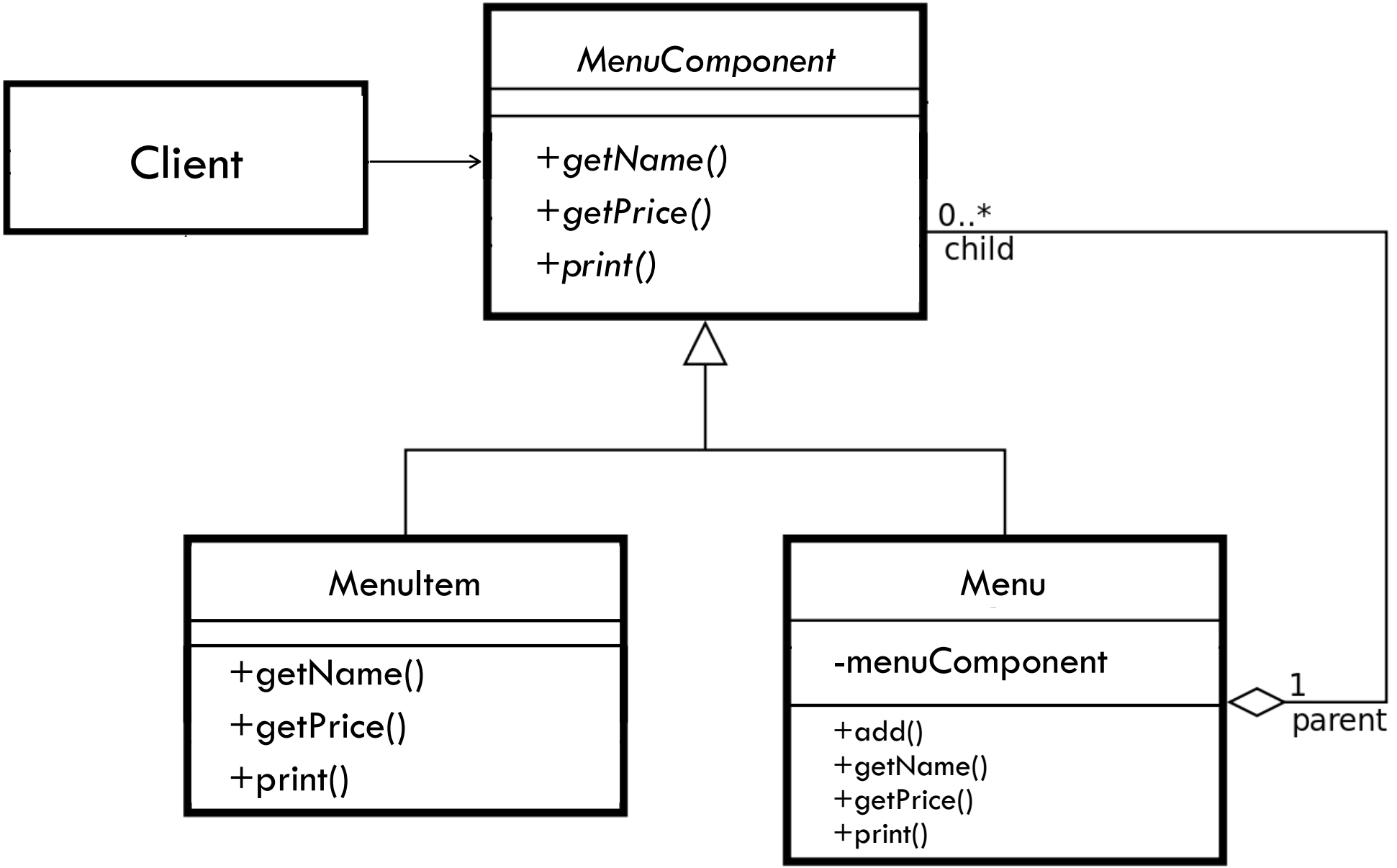
Composite Pattern – Example (1)



Composite Pattern – Example (1)



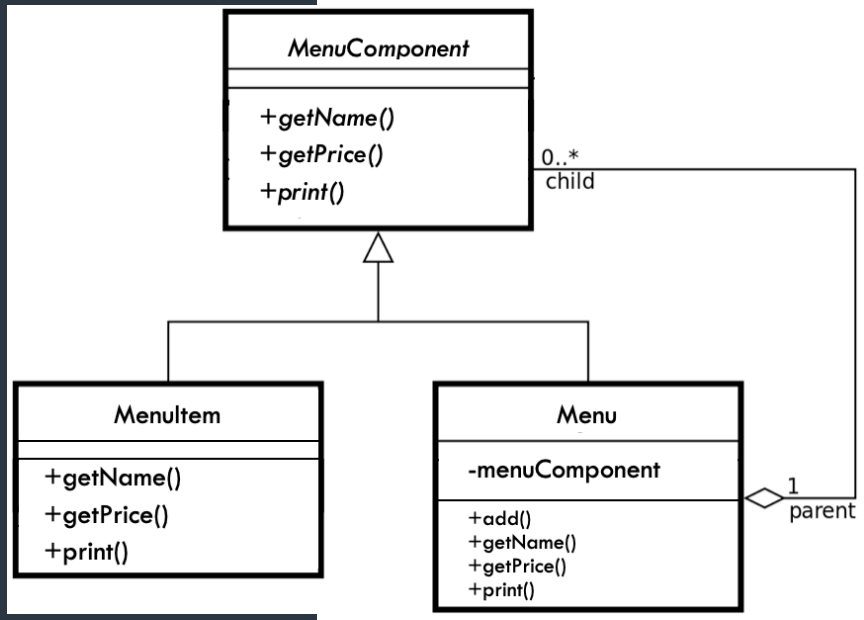
Composite Pattern – Example (2)



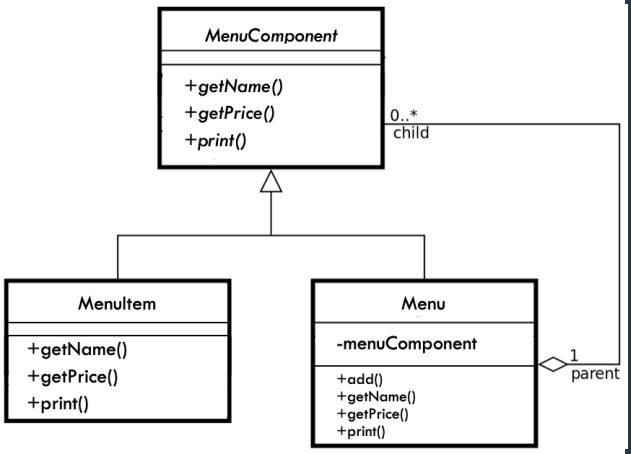
Example code (1)


```java
// Component
interface MenuComponent {
    String getName();
    double getPrice();
    void print();
}
```

```java
// Leaf
class MenuItem implements MenuComponent {
    private String name;
    private double price;

    public MenuItem(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public double getPrice() {
        return price;
    }

    @Override
    public void print() {
        System.out.println("Item: " + name + ", Price: $" + price);
    }
}
```

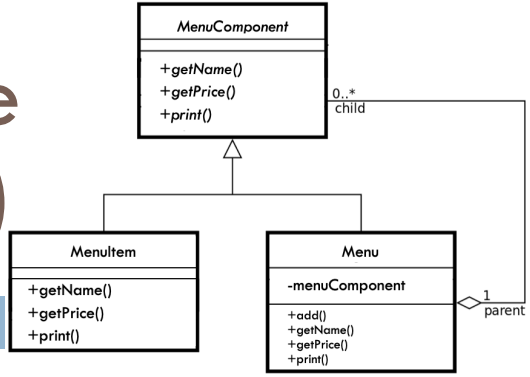


Example code (2)




```java
// Composite
class Menu implements MenuComponent {
    private String name;
    private List<MenuComponent> menuComponent = new ArrayList<>();

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

    public void addSubItem(MenuComponent subItem) {
        menuComponent.add(subItem);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public double getPrice() {
        double totalPrice = 0;
        for (MenuComponent subItem : menuComponent) {
            totalPrice += subItem.getPrice();
        }
        return totalPrice;
    }

    @Override
    public void print() {
        System.out.println("Category: " + name);
        for (MenuComponent subItem : menuComponent) {
            subItem.print();
        }
    }
}
```

Example code (3)




```
Item: Tomato Soup, Price: $6.99
------------------------------
```

```
Category: Vegetarian Menu
Item: Caesar Salad, Price: $8.99
Item: Tomato Soup, Price: $6.99
------------------------------
```

```
Category: Restaurant Menu
Category: Main Course
Item: Spaghetti, Price: $12.99
Category: Vegetarian Menu
Item: Caesar Salad, Price: $8.99
Item: Tomato Soup, Price: $6.99
Category: Recommended Menu
Item: Tomato Soup, Price: $6.99
------------------------------
```

```
Total Price: $35.96
```

```java
// Client
public class MenuManagement {
    public static void main(String[] args) {
        // Creating leaf menu items
        MenuComponent pasta = new MenuItem("Spaghetti", 12.99);
        MenuComponent salad = new MenuItem("Caesar Salad", 8.99);
        MenuComponent soup = new MenuItem("Tomato Soup", 6.99);

        // Printing a menu item
        soup.print();
        System.out.println("------------------------------");

        // Creating a submenu
        Menu vegetarianMenu = new Menu("Vegetarian Menu");
        vegetarianMenu.addSubItem(salad);
        vegetarianMenu.addSubItem(soup);

        // Printing a submenu
        vegetarianMenu.print();
        System.out.println("------------------------------");

        // Creating composite menu items (categories)
        Menu mainCourse = new Menu("Main Course");
        mainCourse.addSubItem(pasta);
        mainCourse.addSubItem(vegetarianMenu);

        Menu recommendedMenu = new Menu("Recommended Menu");
        recommendedMenu.addSubItem(soup);

        // Creating the root of the menu (composite)
        Menu restaurantMenu = new Menu("Restaurant Menu");
        restaurantMenu.addSubItem(mainCourse);
        restaurantMenu.addSubItem(recommendedMenu);

        // Printing the entire menu
        restaurantMenu.print();
        System.out.println("------------------------------");

        // Calculating the total price
        double totalPrice = restaurantMenu.getPrice();
        System.out.println("\nTotal Price: $" + totalPrice);
    }
}
```

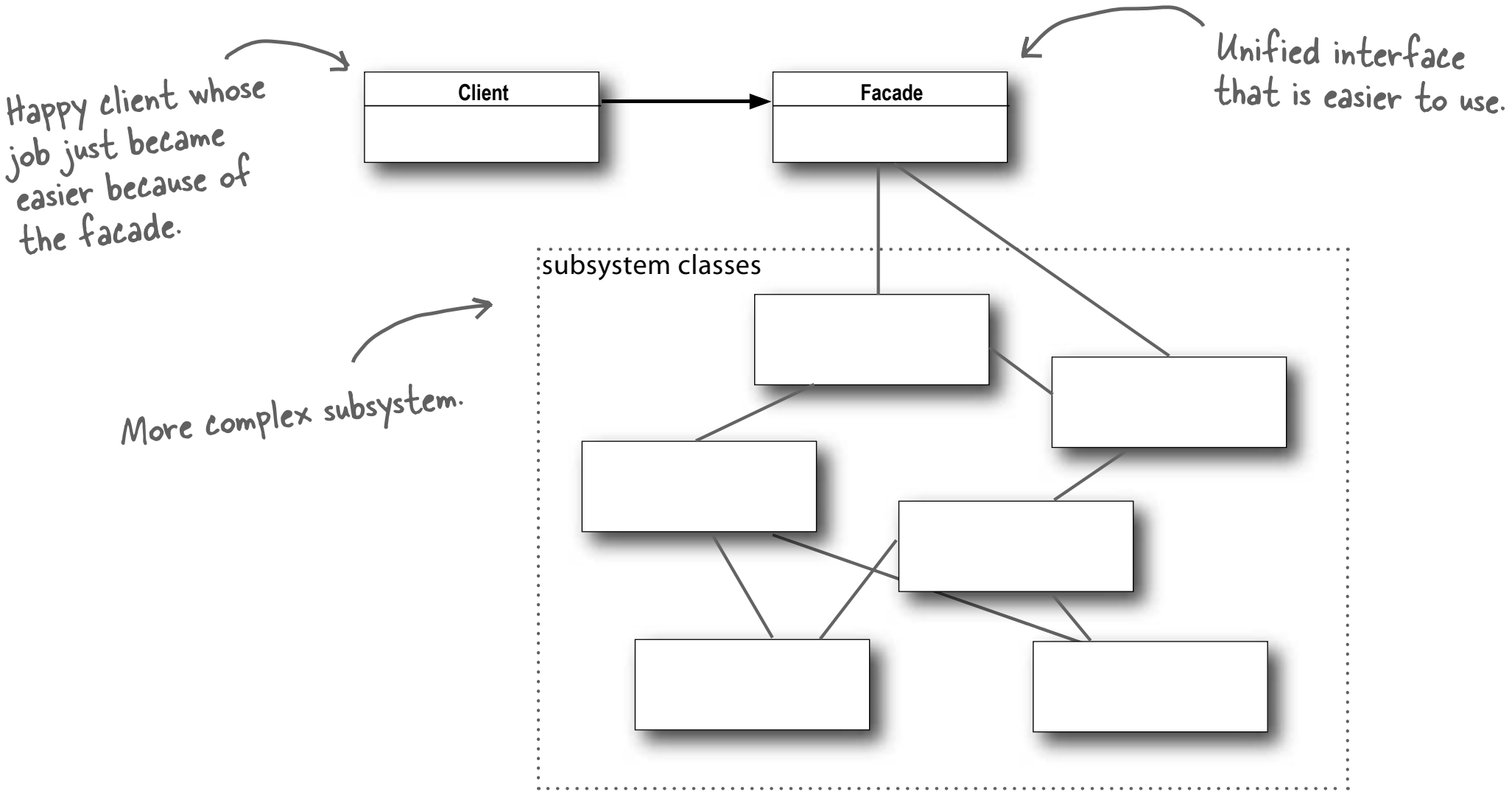
Composite Pattern: Consequences

- It defines class hierarchies consisting of primitive objects and composite objects.
- It makes the client simple.
- It makes it easier to add new kinds of components.
- It can make your design overly general.

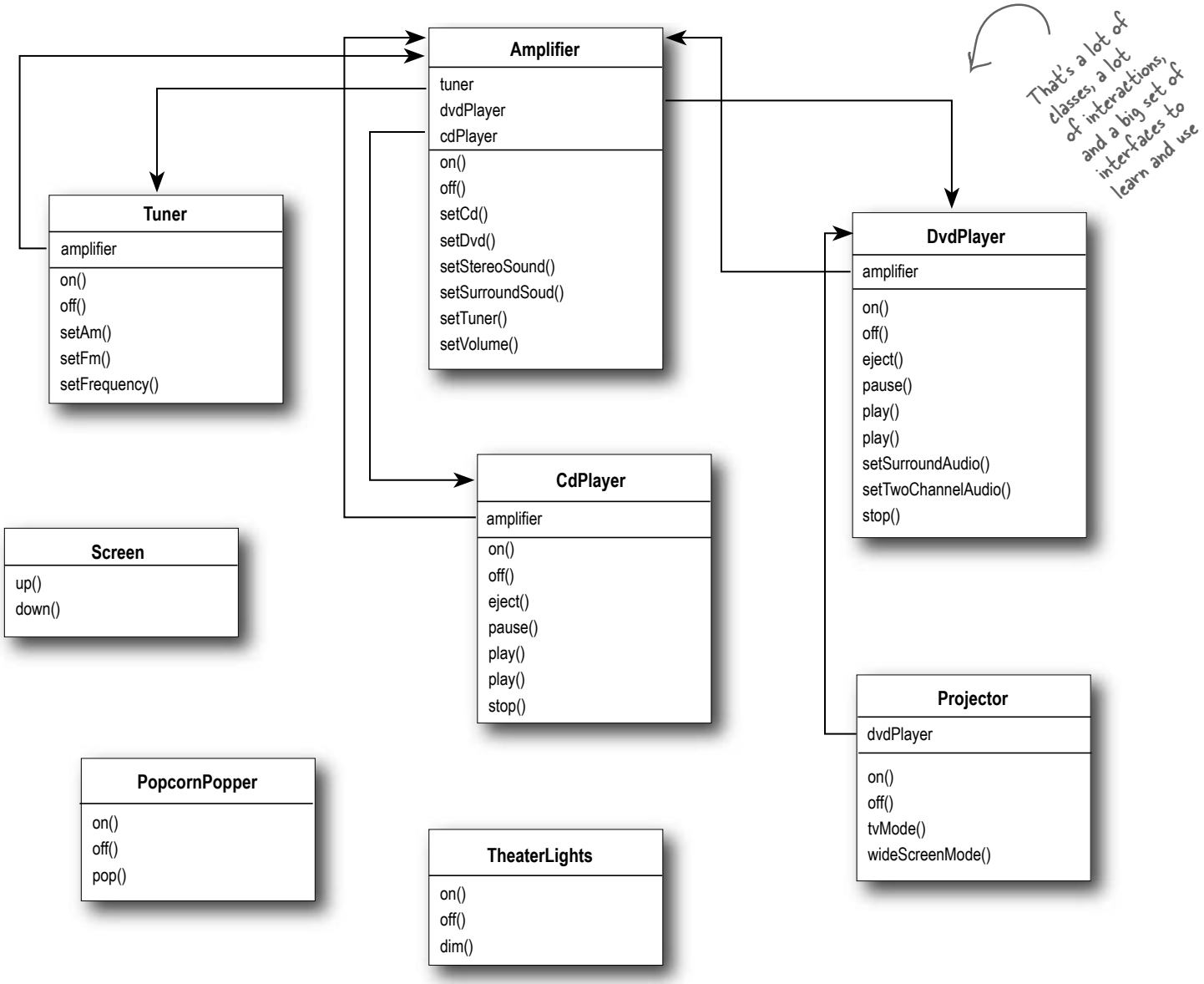
Façade Pattern – Description

- **Name:** Façade pattern
- **Problem:** How can one design a class that provides a simpler and unified interface to a set of interfaces in complex classes? This is to avoid repeated code and increases maintainability.
- **Context:** Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems.

Façade Pattern – Description



Example: Home Theater

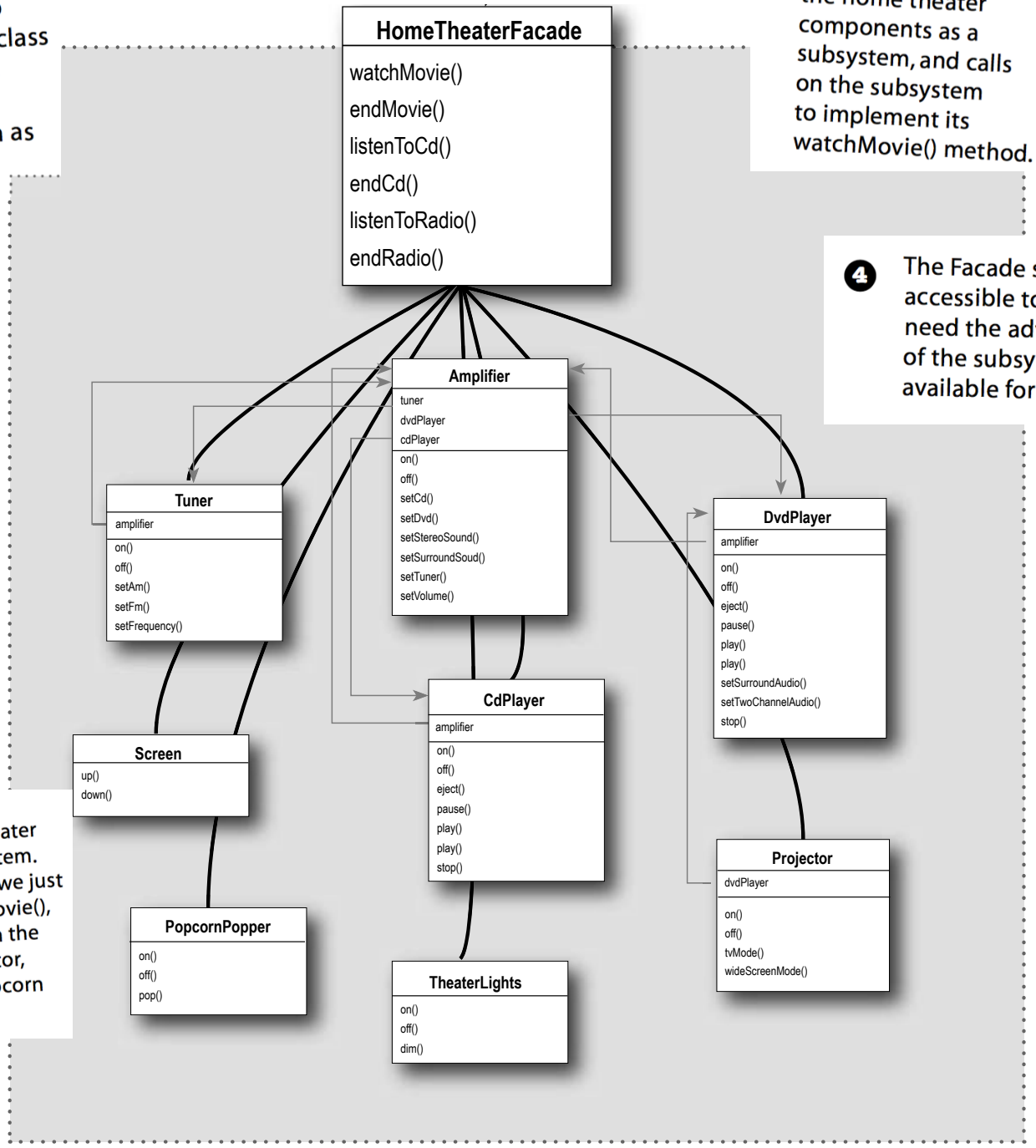


1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector on wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD Player on
13. Start the DVD Player playing

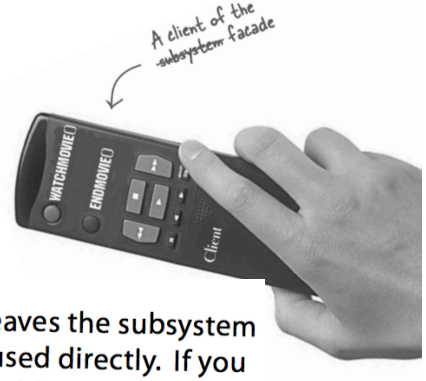
```
popper.on();
popper.pop();

lights.dim(10);

screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode()

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

Example: HomeTheaterFaçade

1. Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().



2. The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.



3. Your client code now calls methods on the home theater Facade, not on the subsystem. So now to watch a movie we just call one method, watchMovie(), and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.

4. The Facade still leaves the subsystem accessible to be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

Example: HomeTheaterFaçade

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                             Tuner tuner,
                             DvdPlayer dvd,
                             CdPlayer cd,
                             Projector projector,
                             Screen screen,
                             TheaterLights lights,
                             PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

        // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Implementing the simplified interface

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

Time to watch a movie (easier way)

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Façade Pattern: Consequences

- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It promotes weak coupling between the subsystem and its clients.
- It doesn't prevent applications from using subsystem classes if they need to.