Structural Design Patterns

Paul Fodor

CSE316: Fundamentals of Software Development

Stony Brook University

http://www.cs.stonybrook.edu/~cse316

Structural Design Patterns

- Design patterns that ease the **design** by identifying a simple way to realize relationships between entities.
  - *Decorator pattern*: adds additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes.
  - *Adapter pattern*: "*adapts*" one interface for a class into one that a client expects.
  - *Facade pattern*: creates a simplified interface of an existing interface to ease usage for common tasks.
  - *Flyweight pattern*: a high quantity of objects share a common properties object to save space.
  - *Bridge pattern*: decouples an abstraction from its implementation so that the two can vary independently.

Common Design Patterns

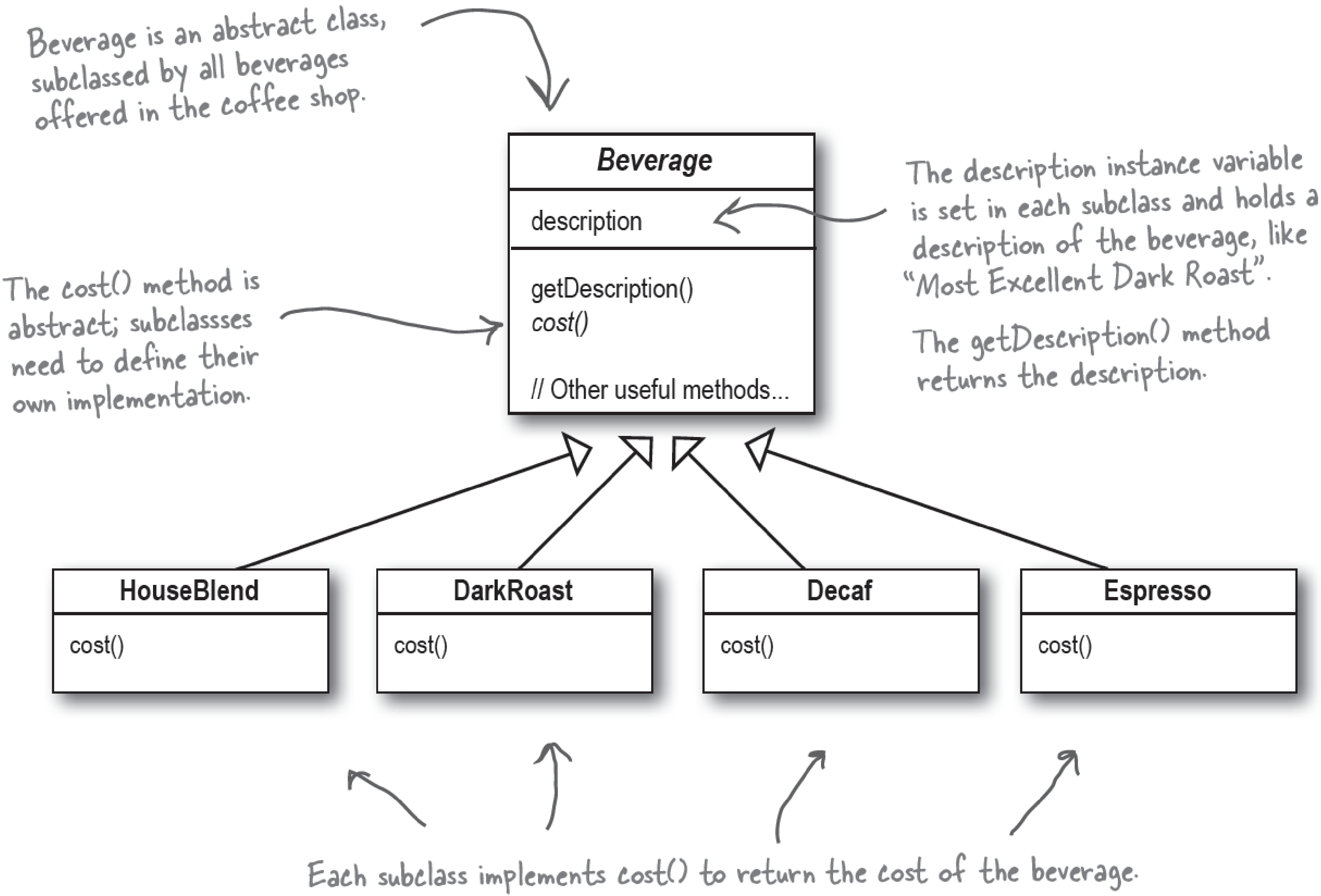
Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Bridge	• Iterator
		• State

Textbook: Head First Design Patterns

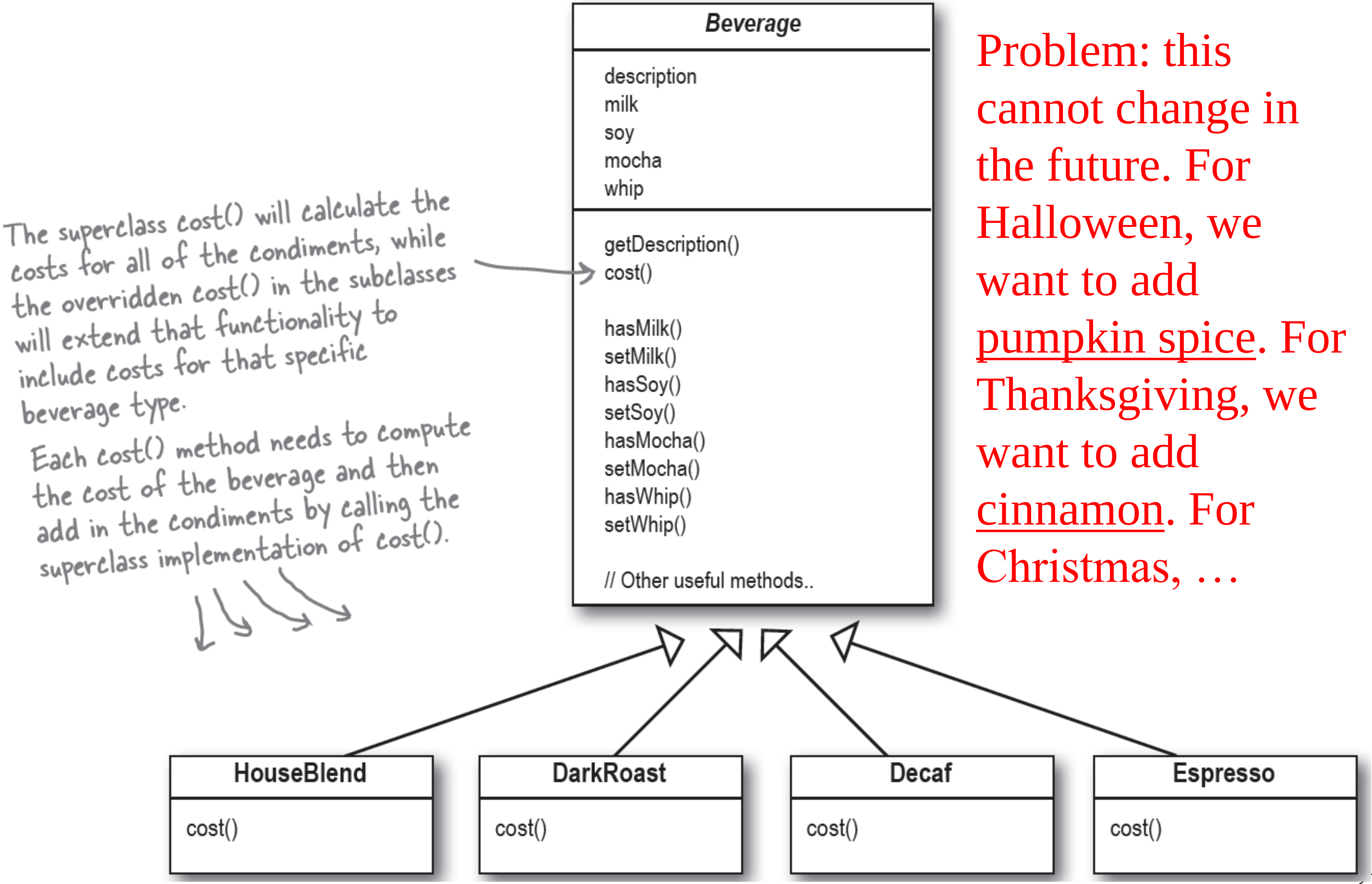
The Decorator Pattern

- Attaches additional responsibilities to an object using delegation
  - i.e. by decorating an object
- Decorators provide a flexible alternative to sub-classing for extending functionality
- How?
  - By wrapping an object in another object
- Works on the principle that classes should be open to extension but closed to modification
  - Allow classes to be easily extended to incorporate new behavior without modifying existing code
    - Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements

Starbuzz Coffee



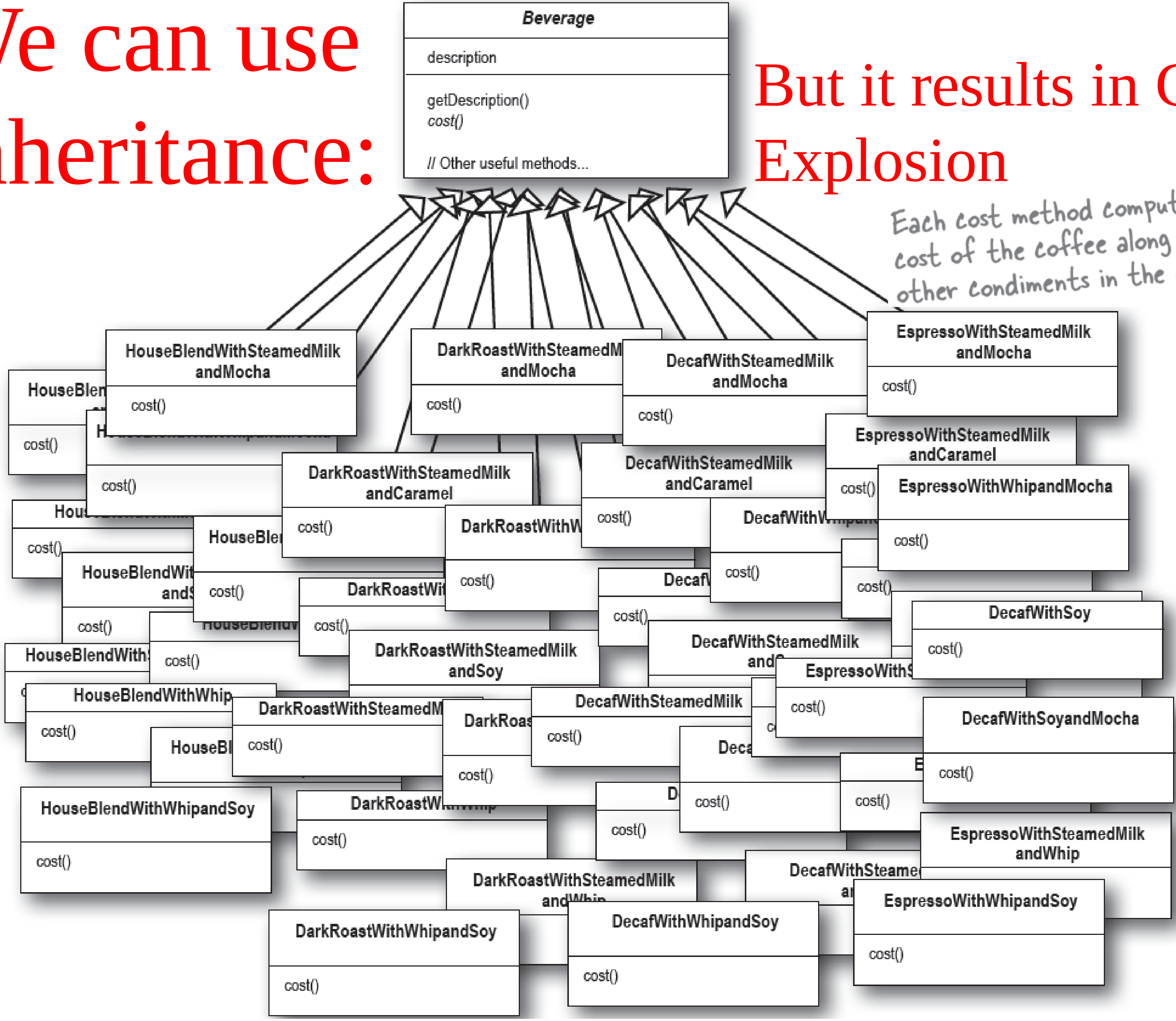
We want to represent whether or not each beverage has milk, mocha, whip, soy, etc.



Problem: this cannot change in the future. For Halloween, we want to add pumpkin spice. For Thanksgiving, we want to add cinnamon. For Christmas, …

We can use inheritance:

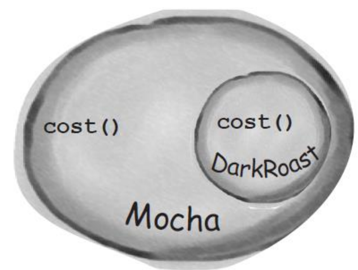
But it results in Class Explosion

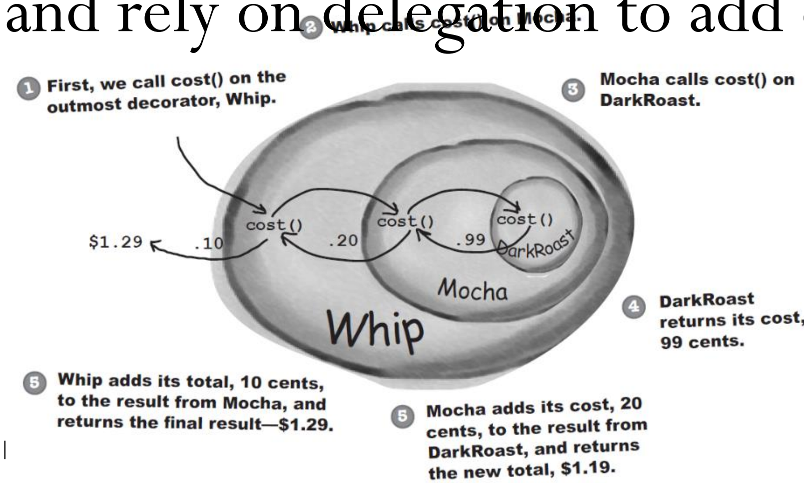


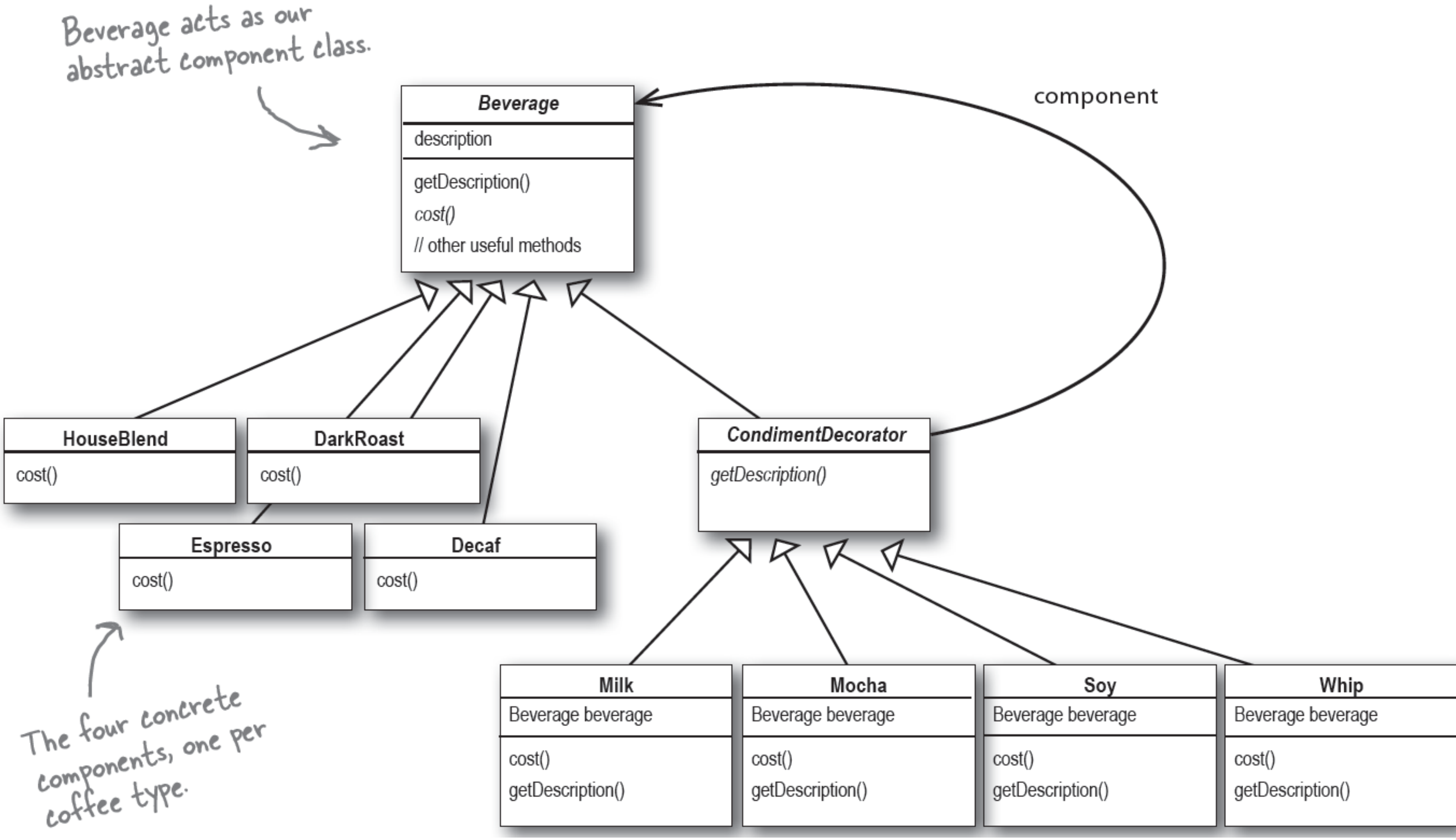
Decorator pattern

- We’ll start with a beverage and "*decorate*" it with the condiments at runtime.
  - For example, if the customer wants a Dark Roast with Mocha and Whip, then we’ll:

1. Take a DarkRoast object.
2. Decorate it with a Mocha object.
3. Decorate it with a Whip object.
4. Call the cost() method and rely on delegation to add on the condiment costs.






Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
The four concrete components, one per coffee type.

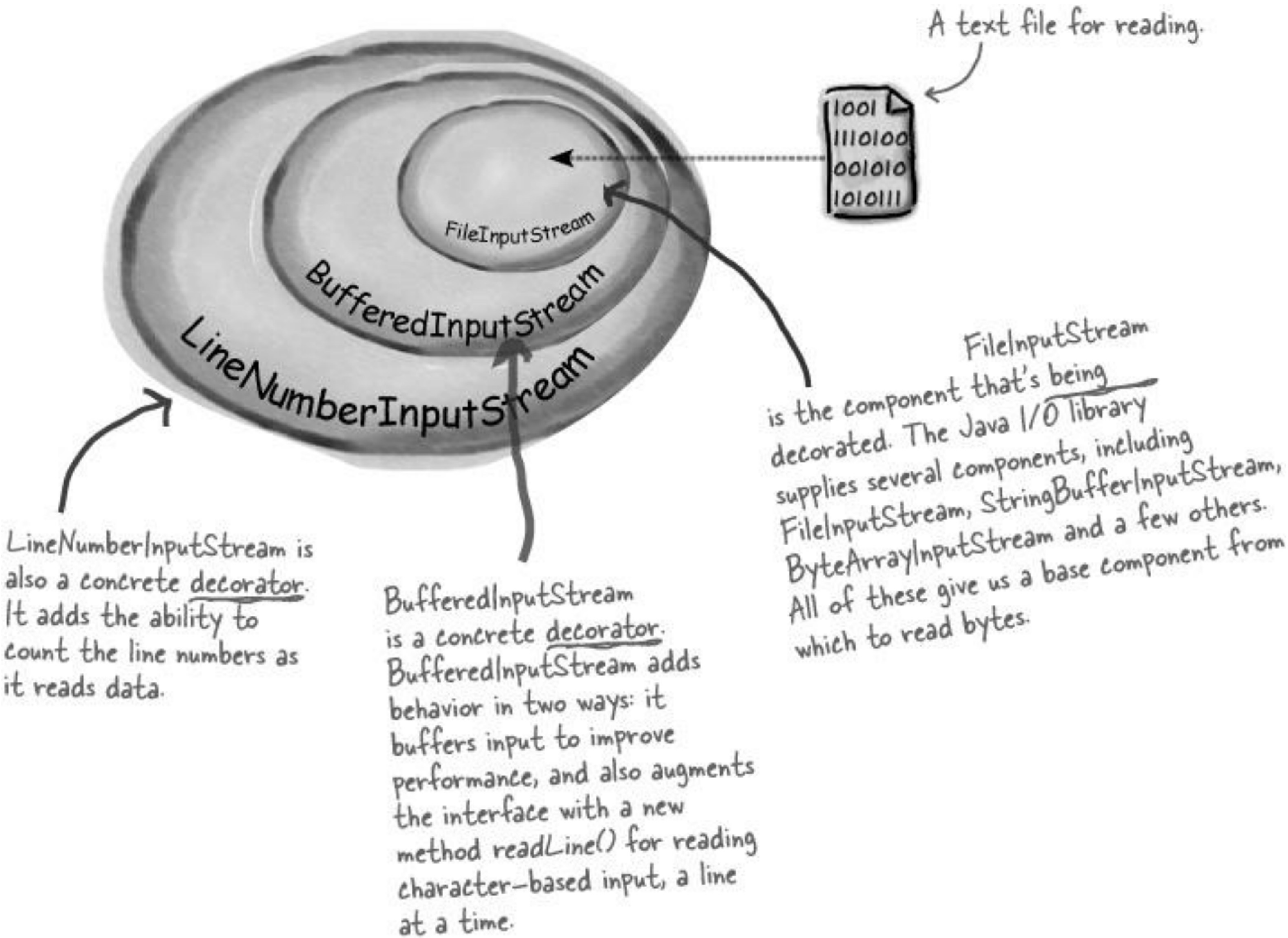
StarbuzzCoffee.java

```java
abstract class Beverage {
  String description = "Unknown Beverage";
  public String getDescription() {
    return description;
  }
  public abstract double cost();
}
class Espresso extends Beverage {
  public Espresso() {
    description = "Espresso";
  }
  public double cost() {
    return 1.99;
  }
}
class DarkRoast extends Beverage {
  public DarkRoast() {
    description = "DarkRoast";
  }
  public double cost() {
    return 0.99;
  }
}
class HouseBlend extends Beverage {
  public HouseBlend() {
    description = "HouseBlend";
  }
  public double cost() {
    return 0.99;
  }
}
```

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

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();
  }
}

class Whip extends CondimentDecorator {
  Beverage beverage;
  public Whip(Beverage beverage) {
    this.beverage = beverage;
  }
  public String getDescription() {
    return beverage.getDescription() + ", Whip";
  }
  public double cost() {
    return .10 + beverage.cost();
  }
}
```

```java
class Soy extends CondimentDecorator {
  Beverage beverage;
  public Soy(Beverage beverage) {
    this.beverage = beverage;
  }
  public String getDescription() {
    return beverage.getDescription() + ", Soy";
  }
  public double cost() {
    return .15 + 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 Mocha(new Whip(new HouseBlend()));
    System.out.println(beverage3.getDescription() + " $" + beverage3.cost());

    Beverage beverage4 = new Soy(new Mocha(new Whip(new HouseBlend())));
    System.out.println(beverage4.getDescription() + " $" + beverage4.cost());

  }
}
```

Java's IO Library



```java
// Adoption in many GUI frameworks for scrollbars
public interface Window {
    public void draw(); // Draws the Window
    public String getDescription(); // Returns a description
}
// Extension of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // Draw window
    }
    public String getDescription() {
        return "simple window";
    }
}
// abstract decorator class - it implements Window
abstract class WindowDecorator implements Window {
    protected Window windowToBeDecorated;
    public WindowDecorator (Window windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }
    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }
```

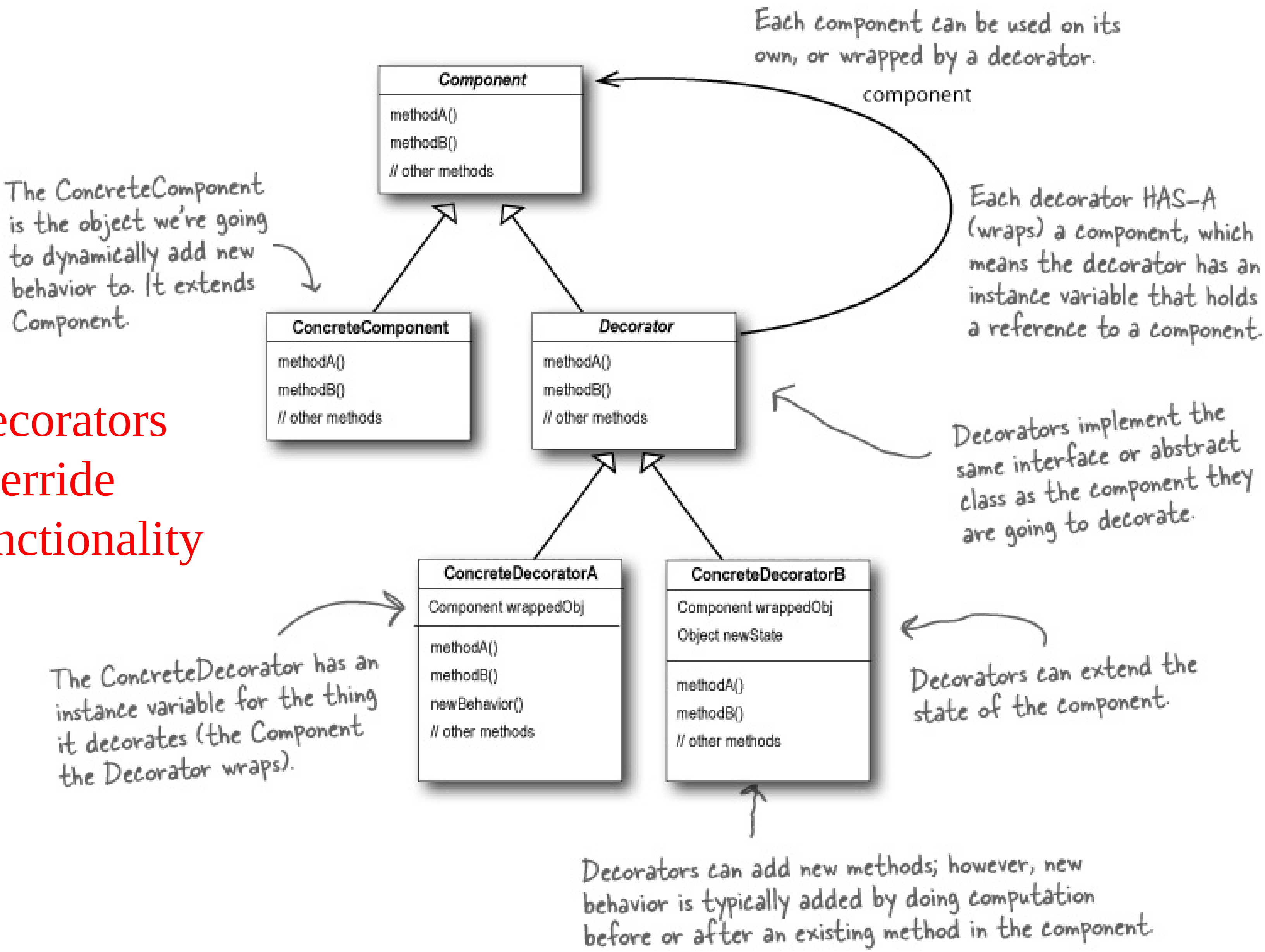
```
    public String getDescription() {
        return windowToBeDecorated.getDescription();//Delegation
    }
}
// The first concrete decorator - adds vertical scrollbar
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator(Window windowToBeDecorated){
        super(windowToBeDecorated);
    }
    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }
    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }
    @Override
    public String getDescription() {
        return super.getDescription()
            + ", including vertical scrollbars";
    }
}
```

```java
// The second concrete decorator - adds horizontal scrollbar
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator(Window windowToBeDecorated){
        super(windowToBeDecorated);
    }
    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }
    private void drawHorizontalScrollBar() {
        // Draw the horizontal scrollbar
    }
    @Override
    public String getDescription() {
        return super.getDescription()
              + ", including horizontal scrollbars";
    }
}
```

```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // Create a decorated Window with horizontal and
        //  vertical scrollbars
        Window decoratedWindow =
              new HorizontalScrollBarDecorator(
               new VerticalScrollBarDecorator(
                new SimpleWindow()));
        // Print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

The output of this program is "simple window, including vertical scrollbars, including horizontal scrollbars" (each decorator decorates the window description with a suffix).

Template for Decorators



Decorators override functionality

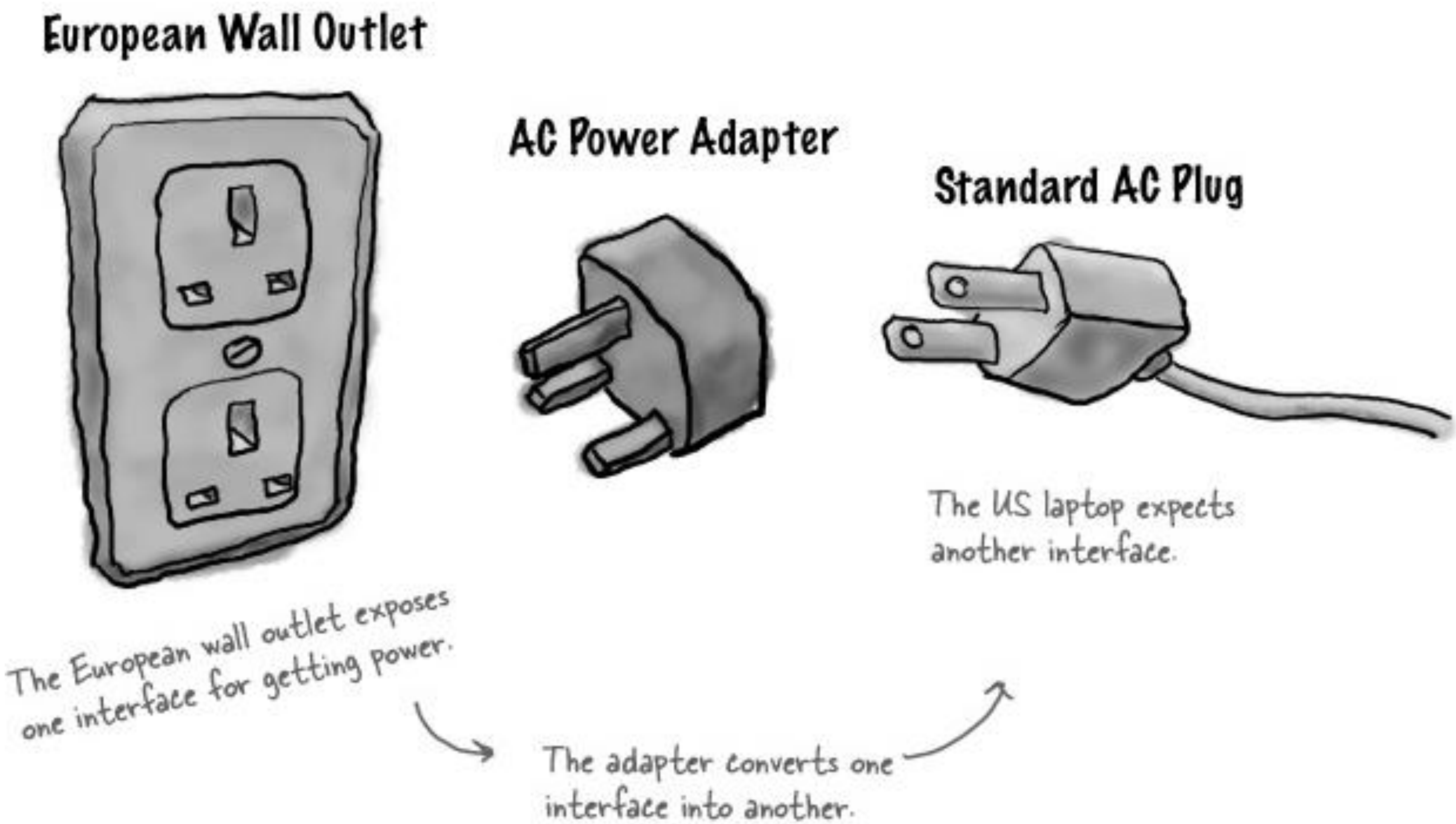
Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Bridge	• Iterator
		• State

Textbook: Head First Design Patterns

Ever been to Europe, Asia or Australia?

- What are adapters?



The Adapter Pattern

- Converts the interface of a class into another interface a client expects
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Device drivers.

The Adapter Pattern

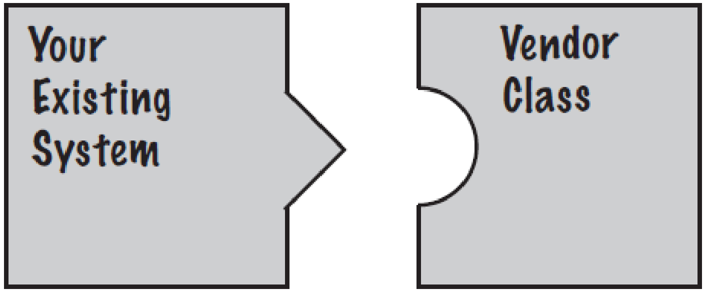
- Do you know what a driver is?

- Problem:
  - Existing system uses a driver through an interface
  - New hardware has a new different interface
- Solution:
  - Adapter can adapt differences

Device drivers.

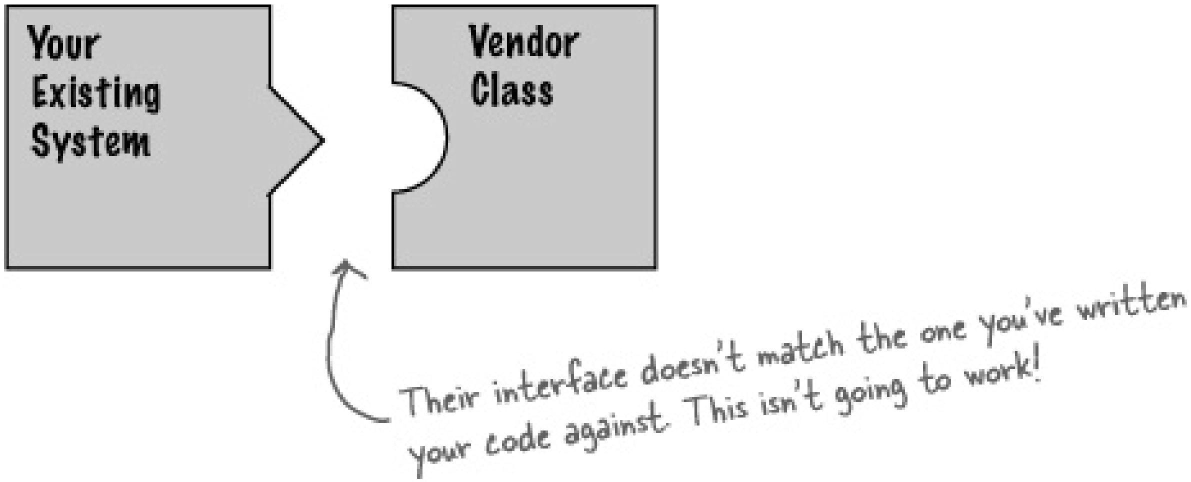
Object oriented adapters

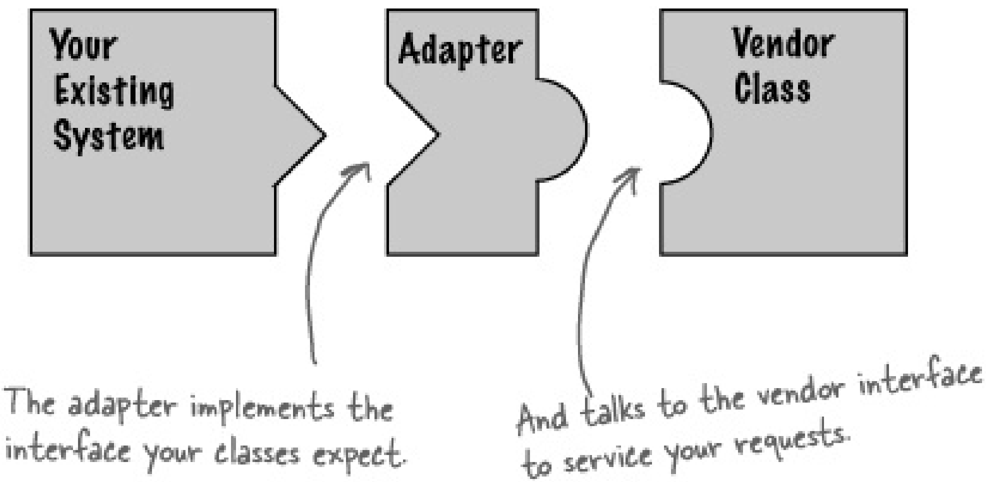
- You have an existing system
- You need to work a vendor library into the system
- The new vendor interface is different from the last vendor

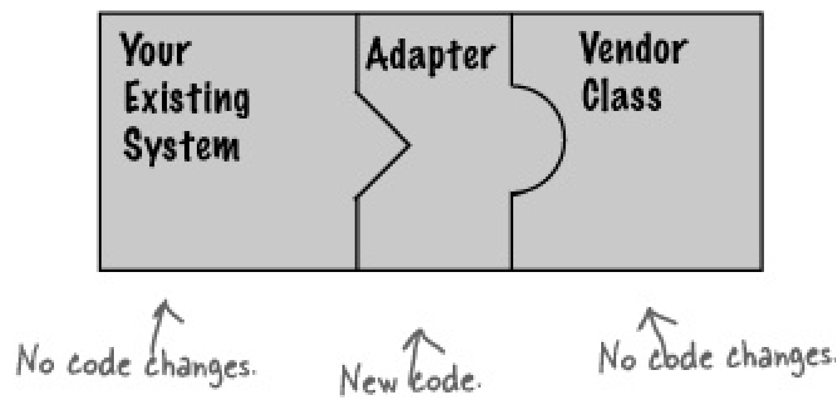


- You really don't want to change your existing system
- Solution: Make a class that adapts the new vendor interface into what the system uses.

Adapter Visualized







How do we do it?

- Existing system HAS-A OldInterface
- Adapter implements OldInterface and HAS-A NewInterface
- Existing system calls OldInterface methods on adapter, adapter forwards them to NewInterface implementations

What's good about this?

- Decouple the client from the implemented interface
- If we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

```java
public interface Duck {
        void quack();
        void walk();
}
public class MallardDuck implements Duck {
        @Override
        public void quack() {
                System.out.println("Quack... quack...");
        }
        @Override
        public void walk() {
                System.out.println("Walking duck ...");
        }
}
public class Main {
        public static void main(String[] args) {
                System.out.println("Duck: ");
                Duck duck = new MallardDuck();
                test(duck);
        }

        static void test(Duck duck) {
                duck.quack();
                duck.walk();
        }
}
```

```java
public class Turkey {
        public void gobble() {
                System.out.println("Gobble ... gobble ...");
        }
        public void walk() {
                System.out.println("Walking turkey ...");
        }
}
public class TurkeyAdapter implements Duck {
        private Turkey turkey;
        public TurkeyAdapter(Turkey turkey) {
                this.turkey = turkey;
        }
        @Override
        public void quack() {
                turkey.gobble();
        }
        @Override
        public void walk() {
                turkey.walk();
        }
}
public class Main {
        public static void main(String[] args) {
                System.out.println("Fake duck (i.e., turkey): ");
                Duck x = new TurkeyAdapter(new Turkey());
                test(x);
        }
        static void test(Duck duck) {
                duck.quack();    duck.walk();
        }
}
```

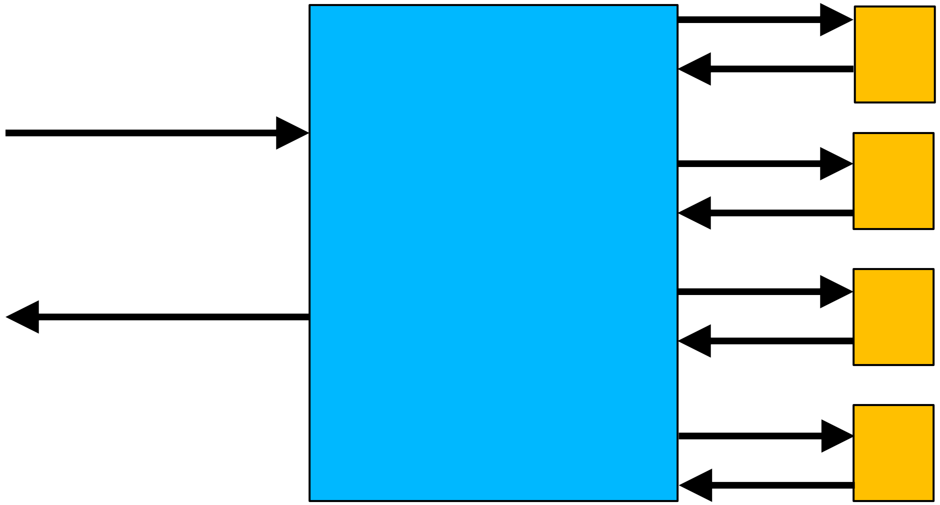
Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Bridge	• Iterator
		• State

Textbook: Head First Design Patterns

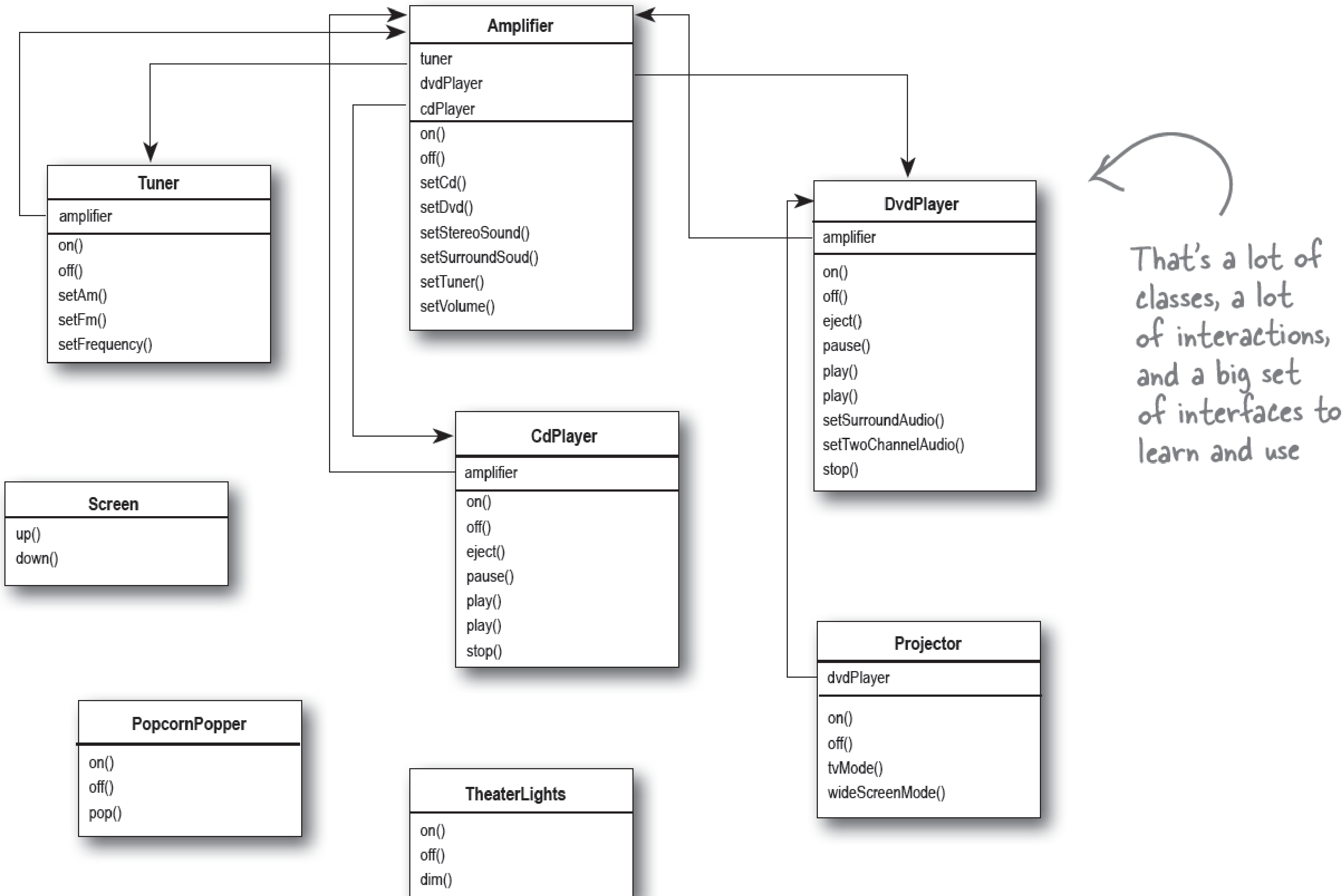
The Facade Pattern

- Provides 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
- Employs the principle of least knowledge



(c) Paul Fodor & O'Reilly Media

Home Sweet Home Theater

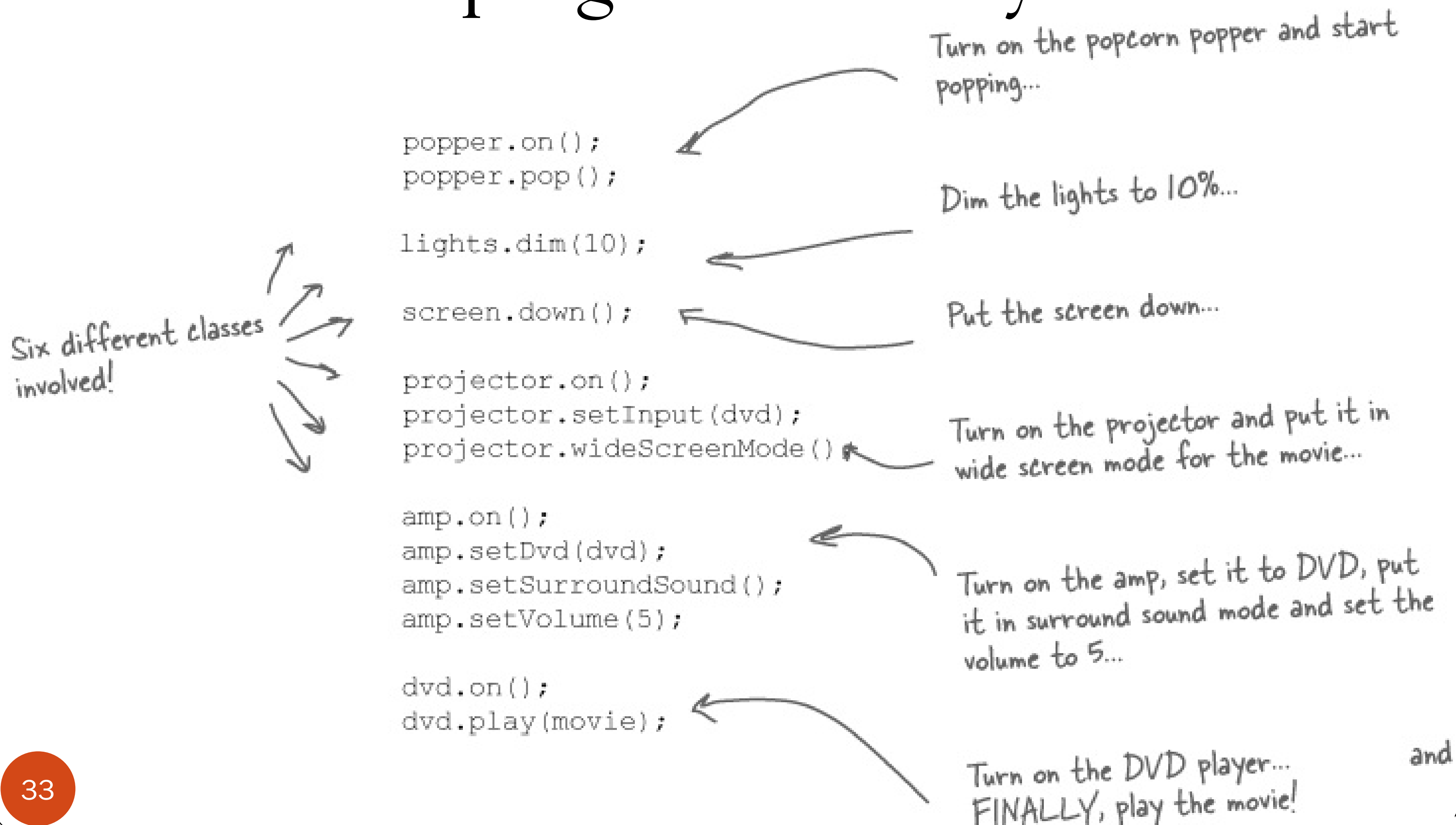


Scenario: Watching a movie

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

Scenario: Watching a movie

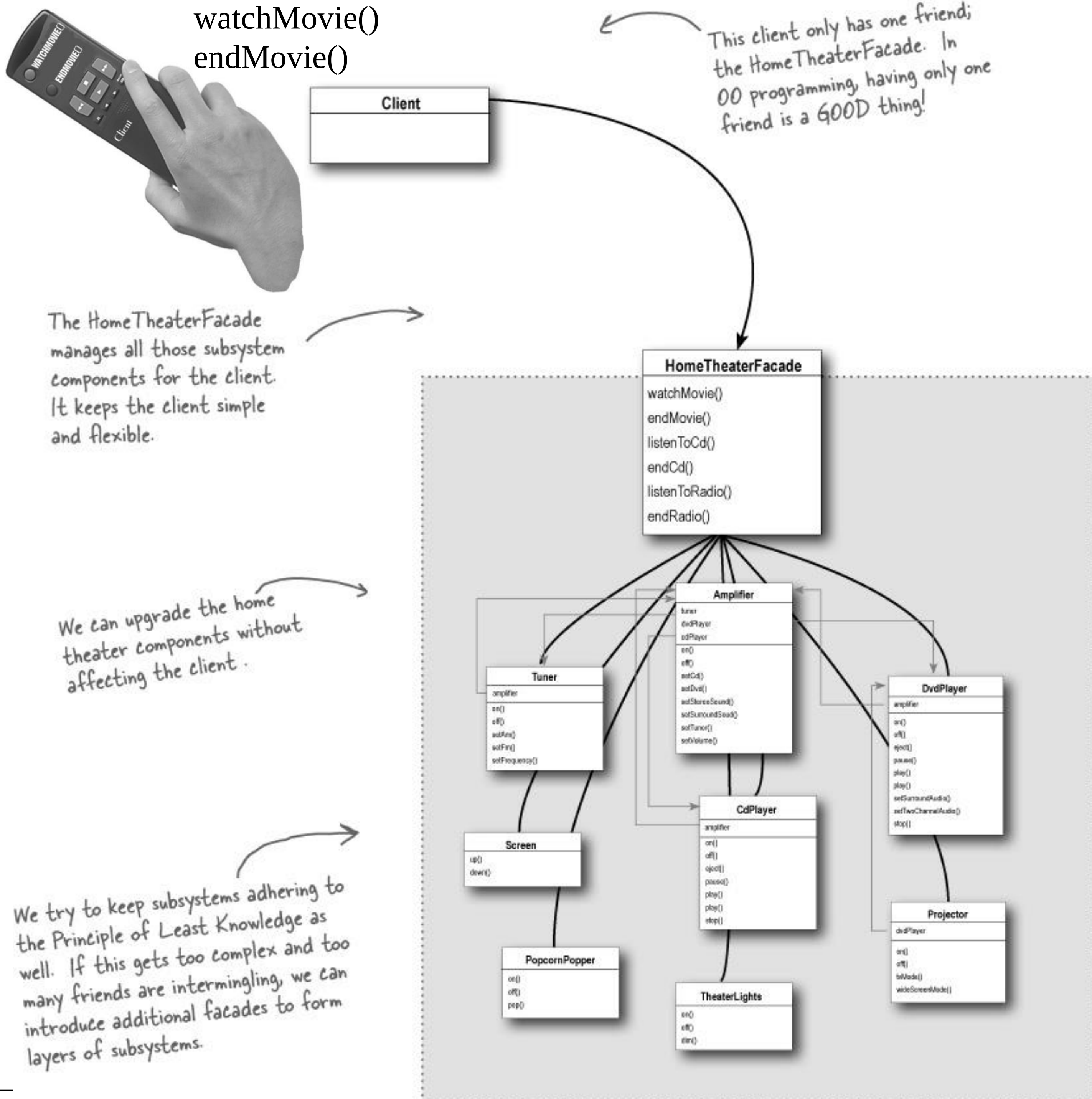
- Let's do it programmatically:



The Facade Pattern

- When the movie is over, how do you turn everything off ?
  - Wouldn't you have to do all of this over again, in reverse?
- If you decide to upgrade your system, you're probably going to have to learn a different procedure.

The Façade Pattern



```
public class HomeTheaterFacade {
  Amplifier amp;
  Tuner tuner;
  DvdPlayer dvd;
  CdPlayer cd;
  Projector projector;
  TheaterLights lights;
  Screen screen;
  ...
  public HomeTheaterFacade(Amplifier amp, Tuner tuner, DvdPlayer
    dvd, CdPlayer cd, Projector projector, Screen screen,
    TheaterLights lights, ...) { ... }
  public void watchMovie(String movie) {
    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() {
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject(); ... }
```

```java
public class Main {
  public static void main(String[] args) {
       HomeTheaterFacade facade = new HomeTheaterFacade(
                                      new Amplifier(),
                                      new Tuner(),
                                      new DvdPlayer(),
                                      new CdPlayer(),
                                      new Projector(),
                                      new Screen(),
                                      new TheaterLights(),...);
       facade.watchMovie("The Croods: A New Age");
       facade.endMovie();
  }
}
```

A computer is a Façade to many components

```
class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}
/* Facade */
class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }
    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
```

```
public static void main(String[] args) {
    ComputerFacade computer = new ComputerFacade();
    computer.start();
}
```

Quiz: Which is which?

- Converts one interface to another B
- Makes an interface simpler C
- Doesn’t alter the interface, but adds responsibility A

A) Decorator

B) Adapter

C) Facade

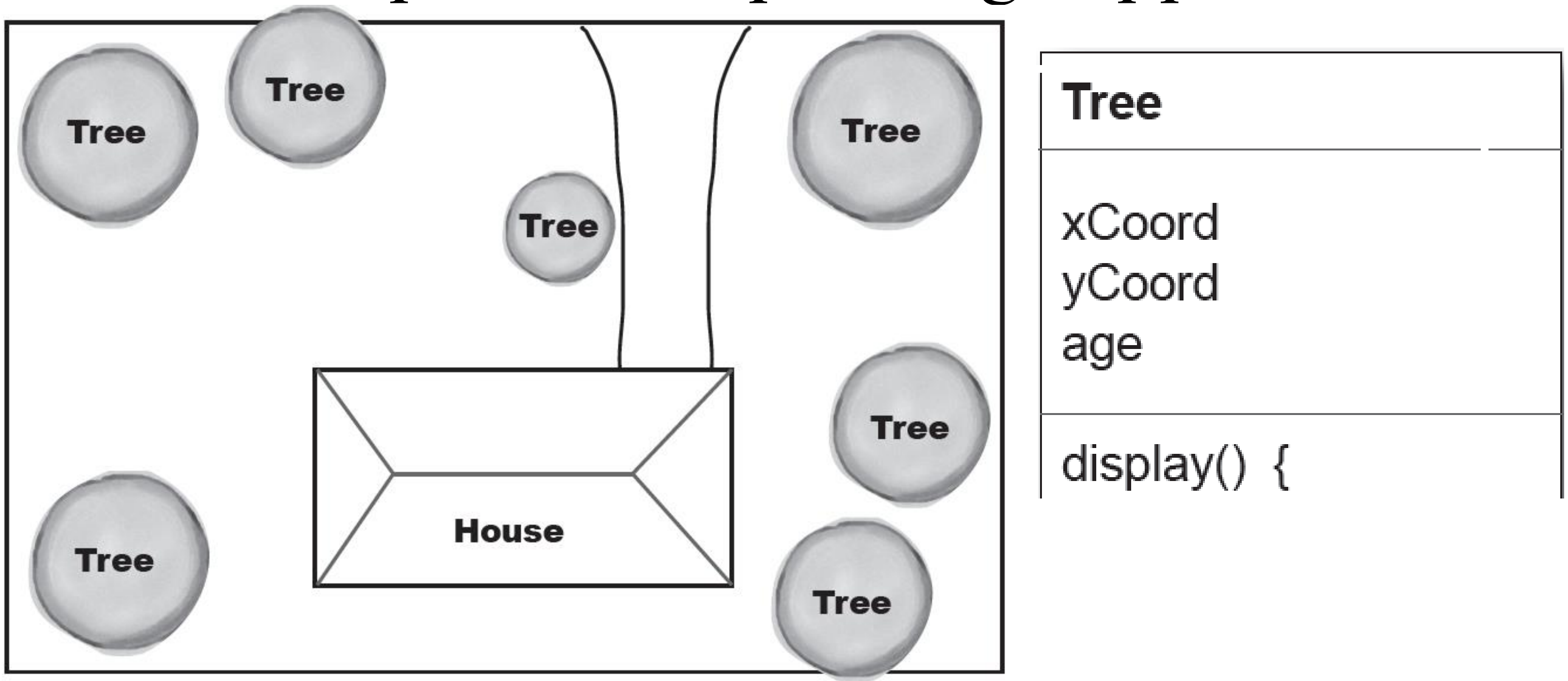
Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

Flyweight scenario

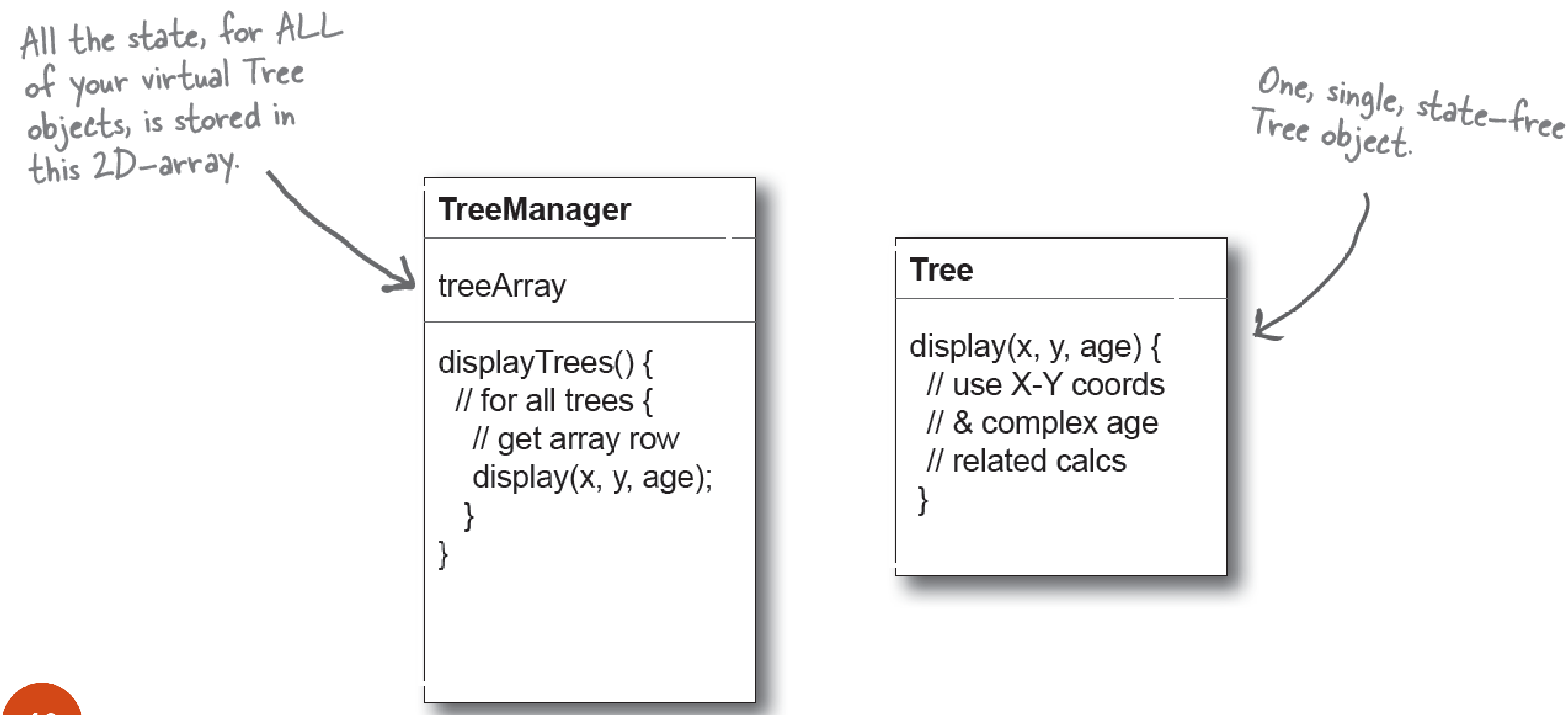
- You develop a landscape design application:



  - After using your software for a week, your client is complaining that when they create large groves of trees, the app starts getting sluggish.

Flyweight scenario

- Flyweight: only one instance of Tree, and a client object that maintains the state of ALL the trees



The Flyweight Pattern

- Allows one object to be used to represent many identical instances
  - The Flyweight is used when a class has many instances, and they can all be controlled identically.
- Flyweight Benefits:
  - Reduces the number of object instances at runtime, saving memory.
  - Centralizes state for many "virtual" objects into a single location.
- Flyweight Uses and Drawbacks:
  - Once you've implemented it, single, logical instances of the class will not be able to behave independently from the other instances.

The Flyweight Pattern

- Flyweights depend on an associated table
  - maps multiple instances to the single object that represents all of them
  - the object must be immutable
- Used in processing many large documents like the Web
  - search engines will use an array of immutable Strings
    - all repeated Words would share the same objects referenced all over the place
      - use **`static Hashtable`** to store mappings

Example: You have a **CoffeeShop**

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
// Instances of CoffeeFlavour will be the Flyweights
class CoffeeFlavour {
  private final String name;
  CoffeeFlavour(String newFlavor) {
    this.name = newFlavor;
  }
  @Override
  public String toString() {
    return name;
  }
}
// Menu acts as a factory and cache for CoffeeFlavour flyweight objects
class Menu {
  private Map<String, CoffeeFlavour> flavours =
    new HashMap<String,CoffeeFlavour>();
  CoffeeFlavour lookup(String flavorName) {  // also uses Singleton
    if (!flavours.containsKey(flavorName))
      flavours.put(flavorName, new CoffeeFlavour(flavorName));
    return flavours.get(flavorName);
  }
  int totalCoffeeFlavoursMade() {
    return flavours.size();
  }
}
```

```
class Order {
  private final int tableNumber;
  private final CoffeeFlavour flavour;
  Order(int tableNumber, CoffeeFlavour flavor) {
    this.tableNumber = tableNumber;
    this.flavour = flavor;
  }
  void serve() {
    System.out.println("Serving " + flavour + " to table " + tableNumber)
  }
}
public class CoffeeShop {
  private final Menu menu = new Menu();
  private final List<Order> orders = new ArrayList<Order>();
  void takeOrder(String flavourName, int table) {
    CoffeeFlavour flavour = menu.lookup(flavourName);
    Order order = new Order(table, flavour);
    orders.add(order);
  }
  void service() {
    for (Order order : orders)
      order.serve();
  }
  String report() {
    return "\ntotal CoffeeFlavour objects made: "
        + menu.totalCoffeeFlavoursMade();
  }
```

```
  public static void main(String[] args) {
    CoffeeShop shop = new CoffeeShop();
    shop.takeOrder("Cappuccino", 2);
    shop.takeOrder("Frappe", 1);
    shop.takeOrder("Espresso", 1);
    shop.takeOrder("Frappe", 8);
    shop.takeOrder("Cappuccino", 9);
    shop.takeOrder("Frappe", 3);
    shop.takeOrder("Espresso", 3);
    shop.takeOrder("Cappuccino", 3);
    shop.takeOrder("Espresso", 9);
    shop.takeOrder("Frappe", 5);
    shop.takeOrder("Cappuccino", 1);
    shop.takeOrder("Espresso", 1);
    shop.service();
    System.out.println(shop.report());
  }
}
Output:
 Serving Cappuccino to table 2
 Serving Frappe to table 1
 Serving Espresso to table 1
 Serving Frappe to table 8
 Serving Cappuccino to table 9
 Serving Frappe to table 3
 Serving Espresso to table 3
 ...
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

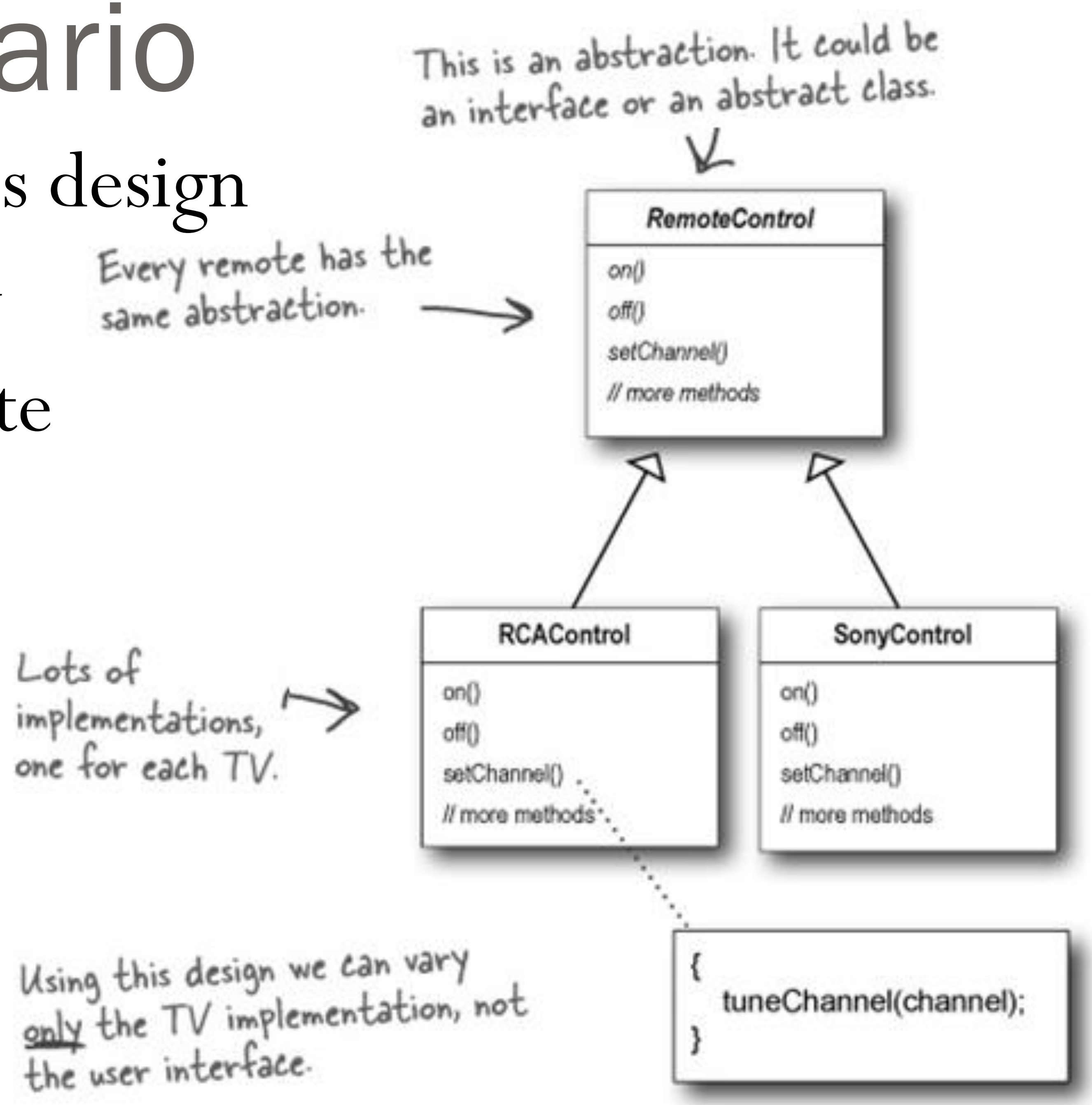
Textbook: Head First Design Patterns

The Bridge Pattern

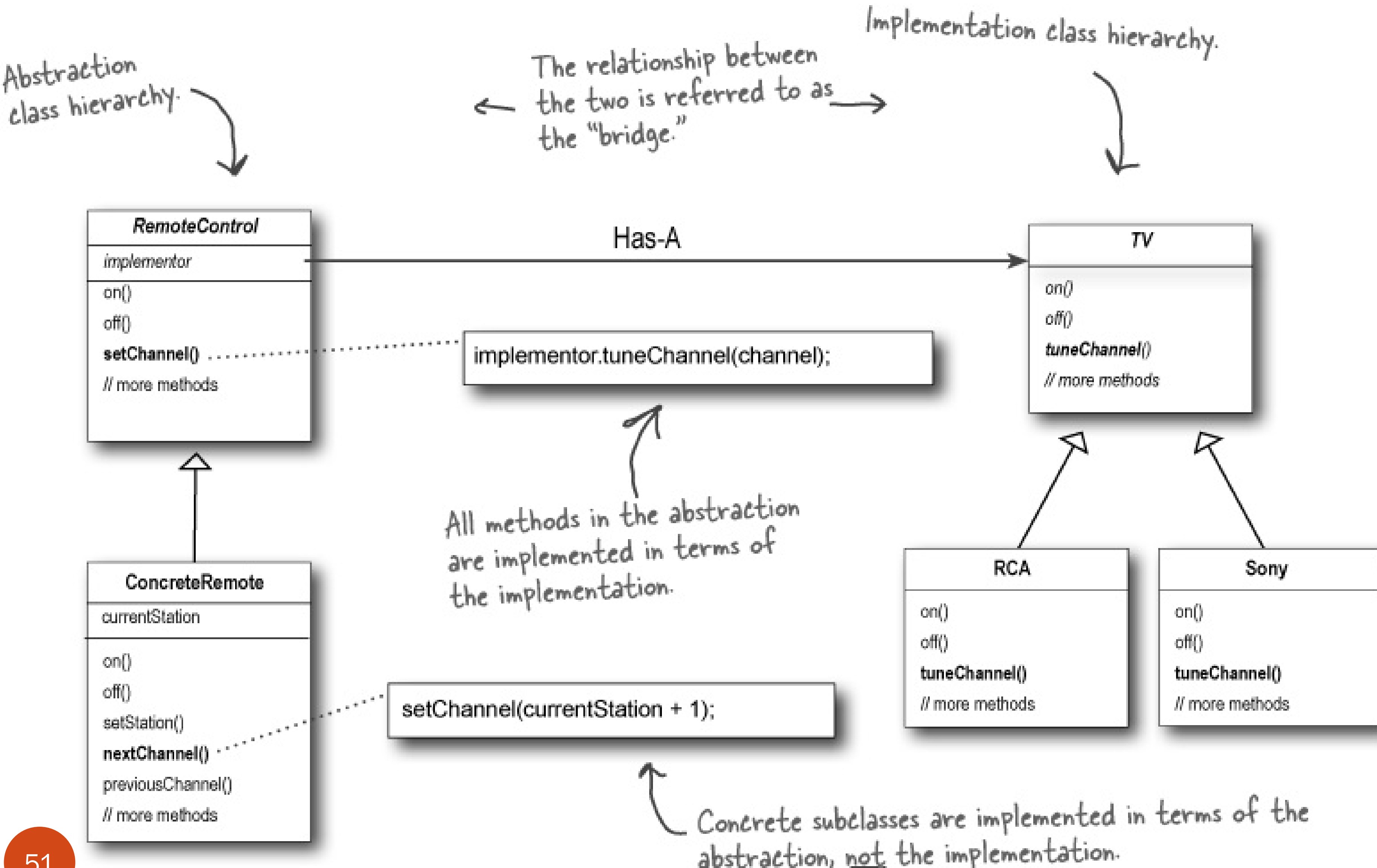
- Used to vary not only your implementations, but also your abstractions!
- Scenario:
  - you're writing the code for a new ergonomic and user-friendly remote control for TVs
  - there will be lots of implementations – one for each model of TV – use an abstraction (interface)
  - you know there will be many changes over time to the specification – needs to accommodate changes
  - Solution? Abstract the abstraction.

Scenario

- A bridge-less design
- Won't easily accommodate change



The Bridge Pattern



The Bridge Pattern

- Benefits:
  - Decouples an implementation so that it is not bound permanently to an interface.
  - Abstraction and implementation can be extended independently.
  - Changes to the concrete abstraction classes don't affect the client.
  - Useful in graphics and windowing systems that need to run over multiple platforms.
  - Useful any time you need to vary an interface and an implementation in different ways.
- Bridge Drawback:
  - Increases complexity

```java
/** "Abstraction" */
abstract class Shape {
    protected DrawingAPI drawingAPI;
    protected Shape(DrawingAPI drawingAPI){
        this.drawingAPI = drawingAPI;
    }
    public abstract void draw();                              // low-level
    public abstract void resizeByPercentage(double pct);    // high-level
}
/** "Refined Abstraction" */
class CircleShape extends Shape {
    private double x, y, radius;
    public CircleShape(double x, double y, double radius,
                DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;  this.y = y;  this.radius = radius;
    }
    // low-level i.e. Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= pct;
    }
}
```

```java
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}
/** "ConcreteImplementor"  1 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
     System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}
/** "ConcreteImplementor" 2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n",x,y,radius);
    }
}
/** "Client" */
public class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        };
        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }}}
```

Output:
API1.circle at 1.0:2.0 radius 7.5
API2.circle at 5.0:7.0 radius 27.5