COMP41670 Software Engineering

16. Design and Implementation including Design Patterns

Dr Avishek Nag

UCD School of Computer Science. **Scoil na Ríomheolaíochta UCD.**

Table of Contents

What Are Design and Implementation?

What Are Design and Implementation?

- **Software design** is the process of identifying the software components and their relationship based on customer requirements.
- **Software implementation** is the process of realising the design.

- Often it is possible to buy off-the-shelf systems that can be adapted or tailored to the users' requirements. In this case, design and implementation are about configuring the system.

Object-Oriented Design using UML

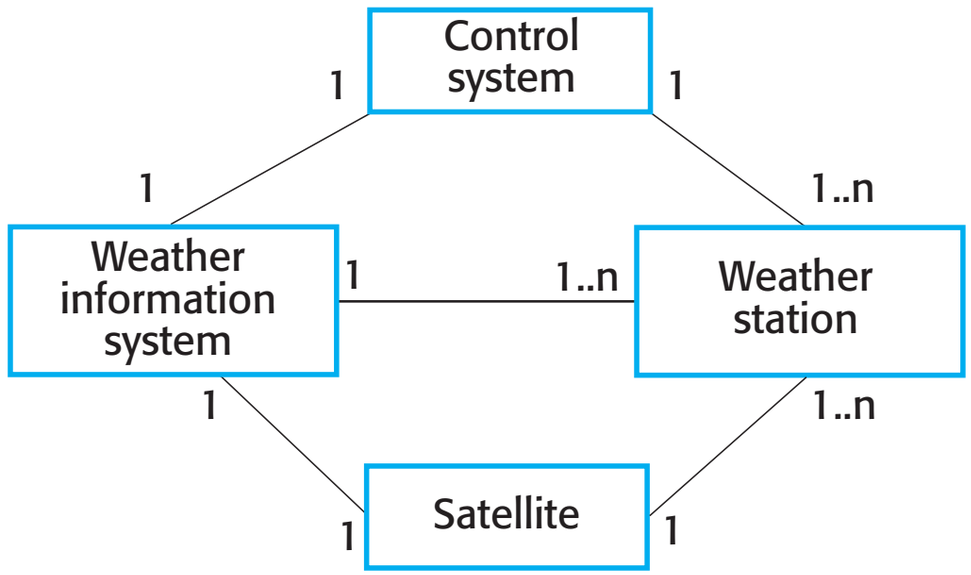
Object Oriented Design using UML

- The level of detail in object-oriented design depends on the size of the system.
- For small systems, the model is likely to be very simple.
- For a large systems, complete UML models may be worthwhile.

Design Process Stages

e.g. Wilderness Weather Station

1. **Context model**
2. Use case diagrams
3. Architectural design
4. Class diagram
5. Sequence diagrams
6. State diagram
7. Object interfaces

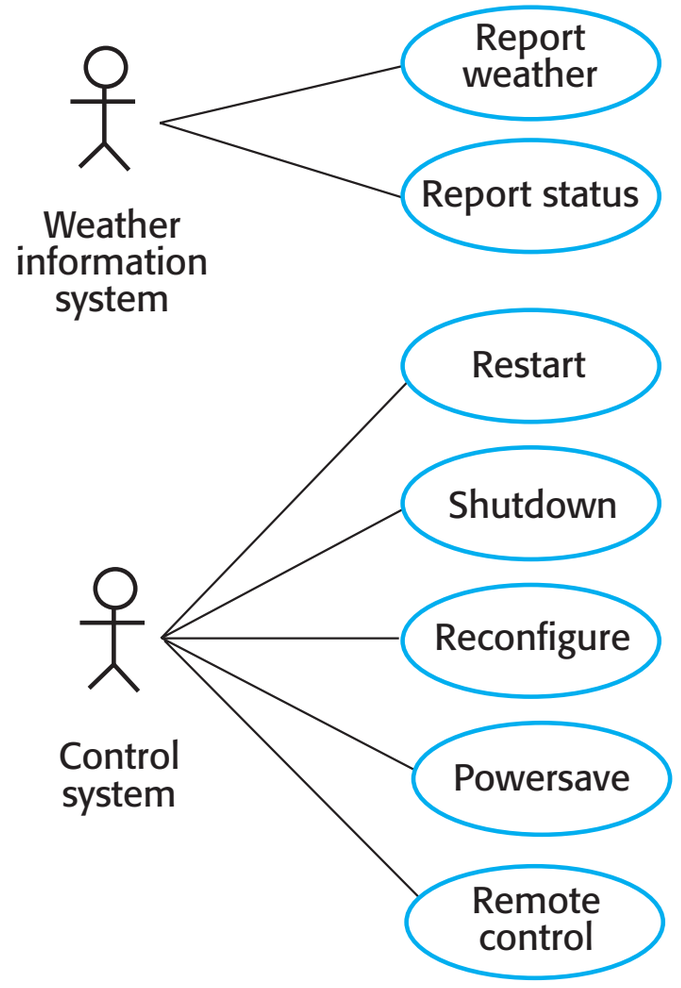


[Chapter 14: System Modelling with UML]

Design Process Stages

1. Context model
2. **Use case diagrams**
3. Architectural design
4. Class diagram
5. Sequence diagrams
6. State diagram
7. Object interfaces

e.g. Wilderness Weather Station

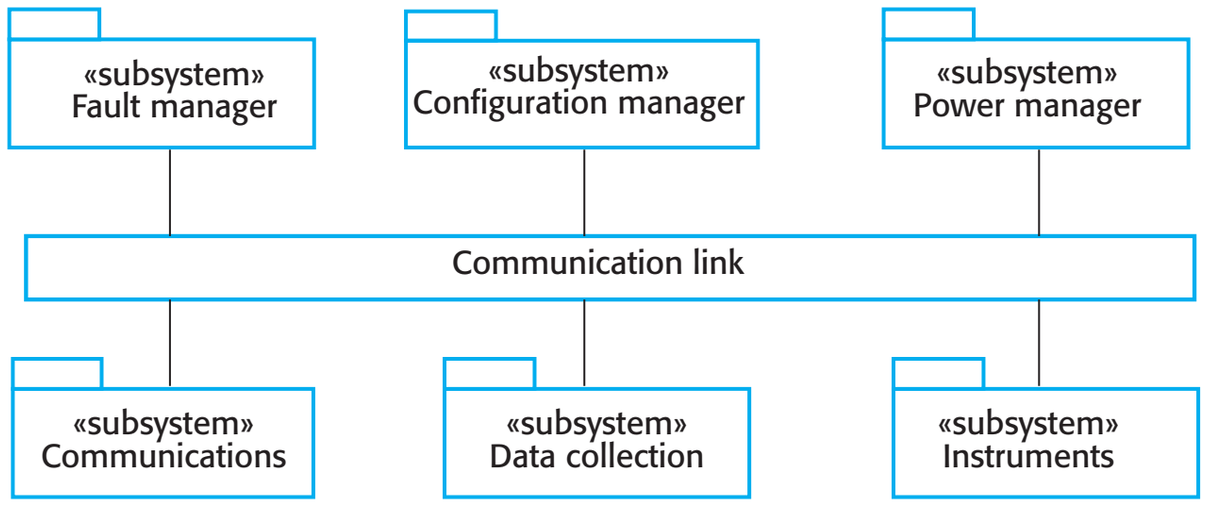


[Chapter 14: System Modelling with UML]

e.g. Wilderness Weather Station

Design Process Stages

1. Context model
2. Use case diagrams
3. **Architectural design**
4. Class diagram
5. Sequence diagrams
6. State diagram
7. Object interfaces

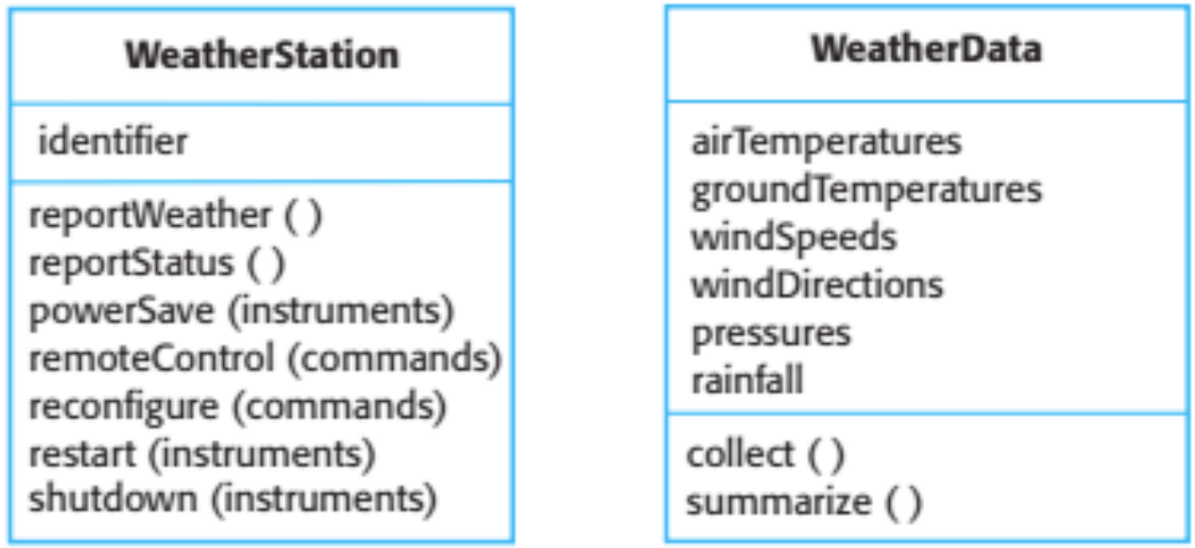


[Chapter 14: System Modelling with UML]

e.g. Wilderness Weather Station

Design Process Stages

1. Context model
2. Use case diagrams
3. Architectural design
4. **Class diagram**
5. Sequence diagrams
6. State diagram
7. Object interfaces

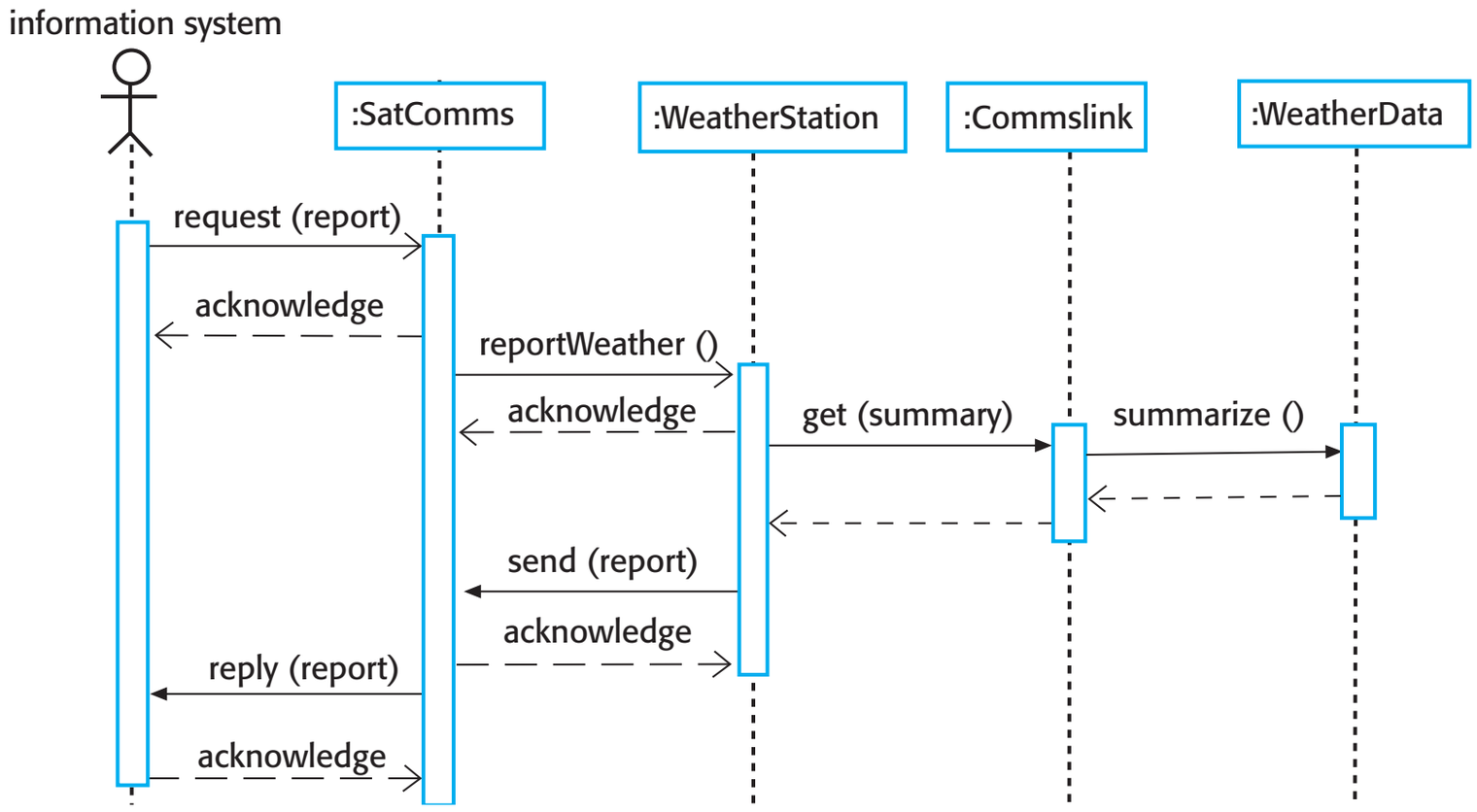


[Chapter 14: System Modelling with UML]

e.g. Wilderness Weather Station

Design Process Stages

1. Context model
2. Use case diagrams
3. Architectural design
4. Class diagram
5. **Sequence diagrams**
6. State diagram
7. Object interfaces

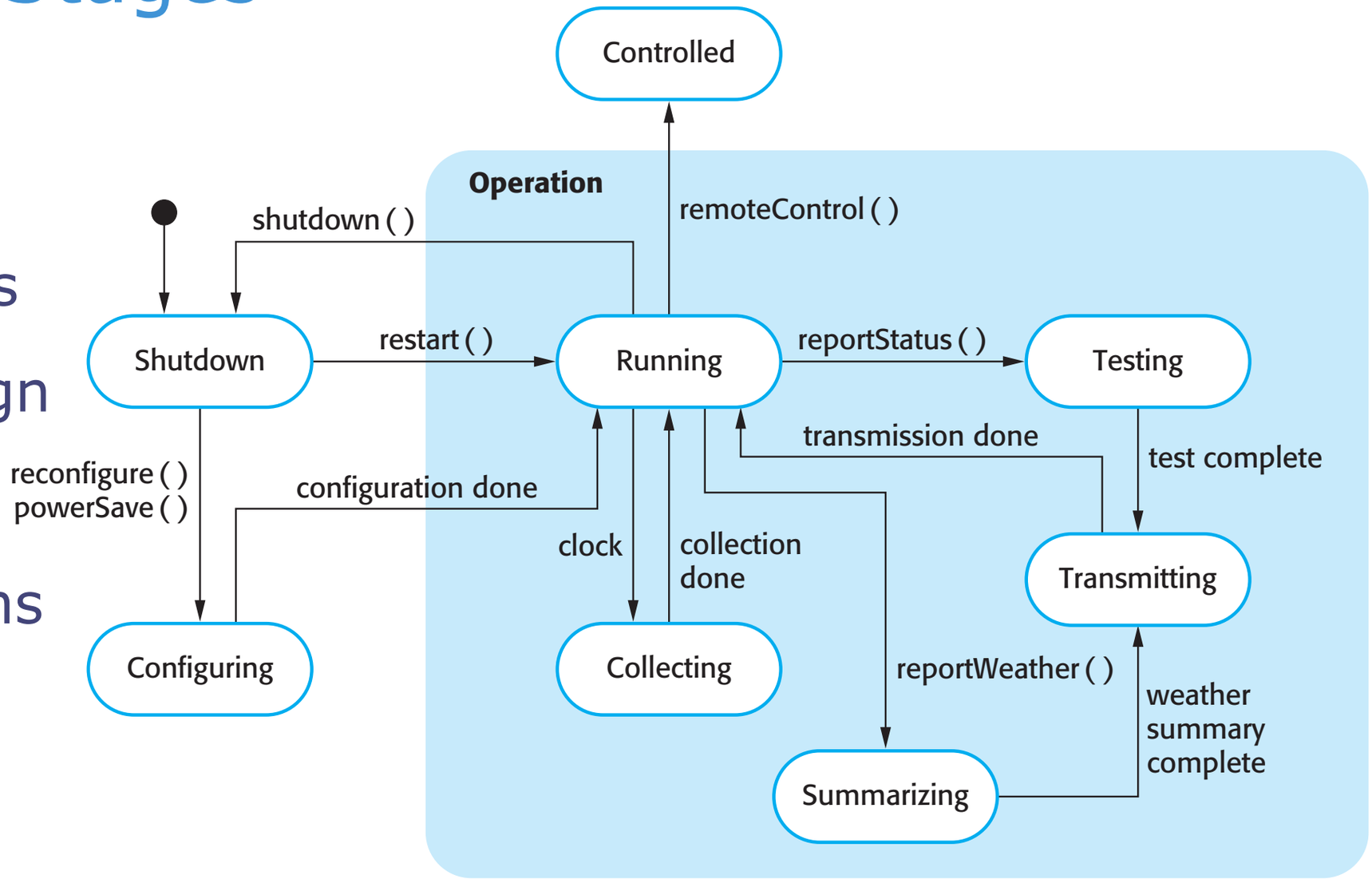


[Chapter 14: System Modelling with UML]

e.g. Wilderness Weather Station

Design Process Stages

1. Context model
2. Use case diagrams
3. Architectural design
4. Class diagram
5. Sequence diagrams
6. **State diagram**
7. Object interfaces

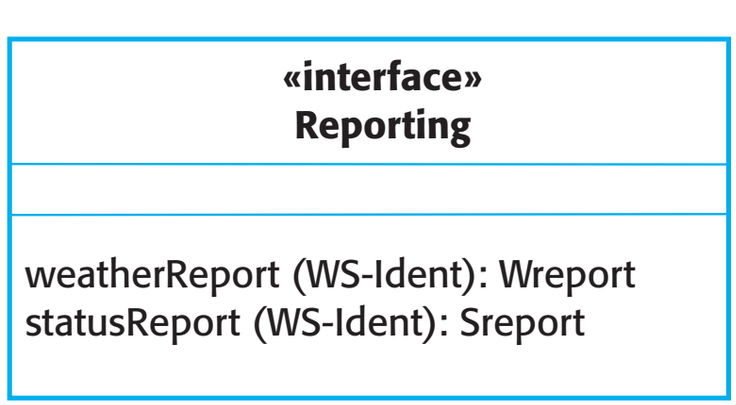


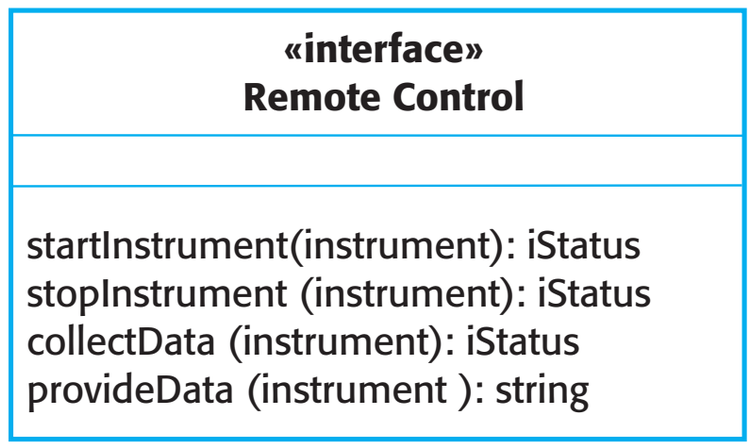
[Chapter 14: System Modelling with UML]

e.g. Wilderness Weather Station

Design Process Stages

1. Context model
2. Use case diagrams
3. Architectural design
4. Class diagram
5. Sequence diagrams
6. State diagram
7. **Object interfaces**





[Chapter 14: System Modelling with UML]

Design Patterns

Design Pattern

- A **pattern** is a high quality transferrable template solution for a commonly occurring design problem.
- A pattern can be documented and re-used for similar design problems.
- Pattern languages can be used to describe design patterns.

Design Patterns History

- The original concept of design patterns came from the building patterns proposed by Christopher Alexander in his 1977 book, *A Pattern Language: Towns, Buildings, Construction*.
- Idea was transferred to software engineering in 1995 in the famous book *Design Patterns: Elements of reusable object-oriented software* by the "Gang of Four", Gamma, Helm, Johnson, and Vlissides.
- The idea was further developed in a series of book by a group of authors from Siemens - Buschmann et al.

Pattern Elements

- Name
- Description
  - What the pattern achieves.
- Problem Description
  - A generic description of the design problem that the pattern solves.
- Solution Description
  - Not a concrete design but a template for a design solution that can be realised in different ways to suit the problem at hand.
- Consequences
  - The results and trade-offs of applying the pattern.

Types of Design Patterns

- Structural
  - Concerned with how classes and objects are composed to form larger structures
- Creational
  - Concerned with the everything about the creation of an object
- Behavioural
  - Concerned with algorithms and the assignment of responsibilities between objects.

Design Patterns

- We will examine the following commonly used structural design patterns:
  - Facade
  - Composite
  - Decorator
  - Singleton
  - Factory
  - Observer
- The following descriptions are taken from the book *Head First Design Patterns: A Brain-Friendly Guide: Building Extensible and Maintainable Object-Oriented Software* by Freeman and Robson (2nd edition, 2021).

Design Patterns: Facade

Facade

- A Facade pattern provides a unified interface to a set of interfaces in a subsystem.
- A Facade pattern defines a higher-level interface that can be used by a client.

Example Problem

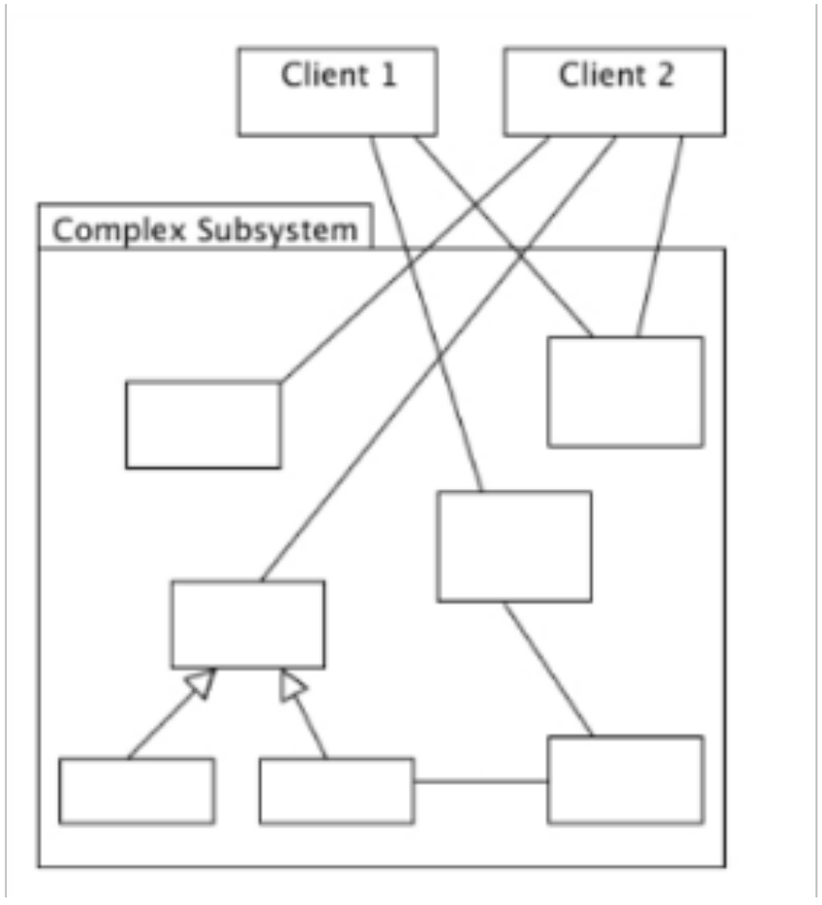
- You have created an awesome, but complicated, home theatre system.
- In order to watch a movie, you have to:
  - Dim the lights
  - Pull down the screen
  - Turn the projector on
  - Set the projector input to DVD
  - Put the projector on widescreen mode
  - Turn the sound amplifier on
  - Set the sound amplifier input to DVD
  - Set the volume
  - Turn the DVD player on
  - Start the DVD player

Example Solution

Dim the lights	LightController
Pull down the screen	ScreenController
Turn the projector on	ProjectorController
Set the projector input to DVD	
Put the projector on widescreen mode	
Turn the sound amplifier on	SoundController
Set the sound amplifier input to DVD	
Set the volume	
Turn the DVD player on	DVDController
Start the DVD player	

Without a Facade

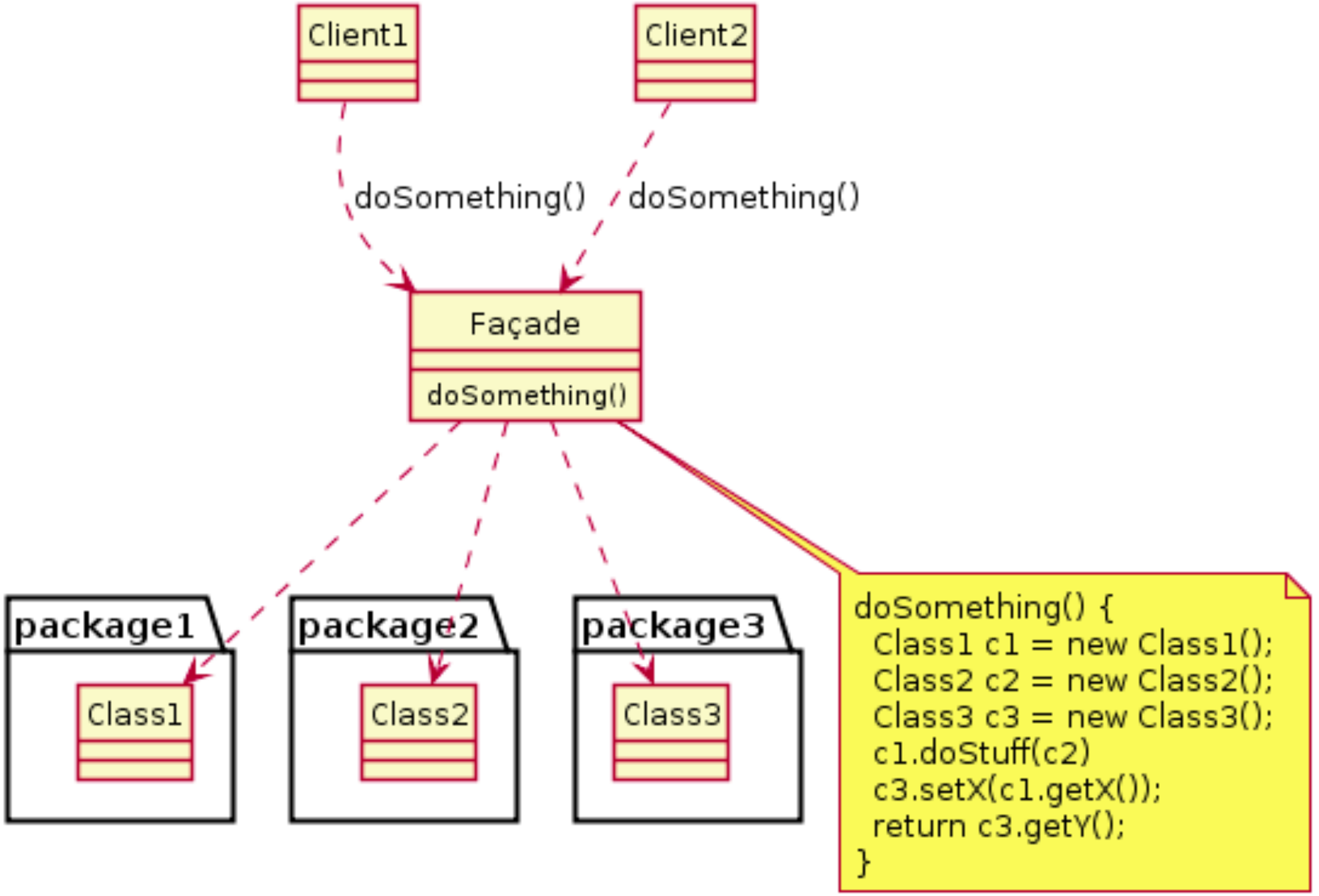
LightController



ScreenController

ProjectorController

With a Facade



https://upload.wikimedia.org/wikipedia/commons/5/57/Example_of_Facade_design_pattern_in_UML.png

```
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;
    }

    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();
    }

    public void listenToCd(String cdTitle) {
        System.out.println("Get ready for an audiopile experence...");
        lights.on();
        amp.on();
        amp.setVolume(5);
        amp.setCd(cd);
        amp.setStereoSound();
        cd.on();
        cd.play(cdTitle);
    }

    public void endCd() {
        System.out.println("Shutting down CD...");
        amp.off();
        amp.setCd(cd);
        cd.eject();
        cd.off();
    }

    public void listenToRadio(double frequency) {
        System.out.println("Tuning in the airwaves...");
        tuner.on();
        tuner.setFrequency(frequency);
        amp.on();
        amp.setVolume(5);
        amp.setTuner(tuner);
    }

    public void endRadio() {
        System.out.println("Shutting down the tuner...");
        tuner.off();
        amp.off();
    }
}
```

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        Amplifier amp = new Amplifier("Top-O-Line Amplifier");
        Tuner tuner = new Tuner("Top-O-Line AM/FM Tuner", amp);
        DvdPlayer dvd = new DvdPlayer("Top-O-Line DVD Player", amp);
        CdPlayer cd = new CdPlayer("Top-O-Line CD Player", amp);
        Projector projector = new Projector("Top-O-Line Projector", dvd);
        TheaterLights lights = new TheaterLights("Theater Ceiling Lights");
        Screen screen = new Screen("Theater Screen");
        PopcornPopper popper = new PopcornPopper("Popcorn Popper");

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

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

Design Patterns: Composite

Composite

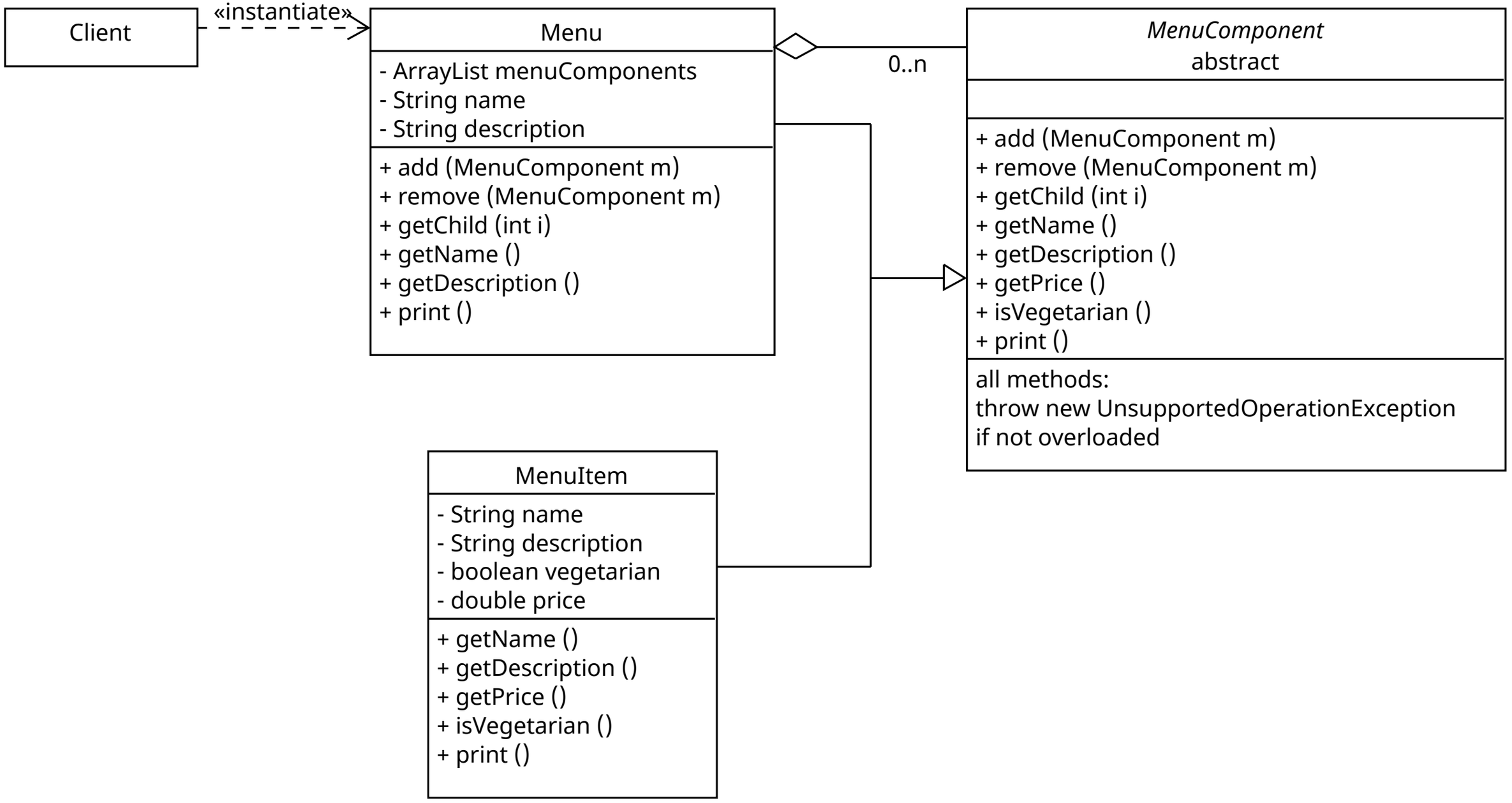
- A Composite pattern allows clients to treat individual objects and compositions of objects, or “composites”, in the same way.

Example Problem

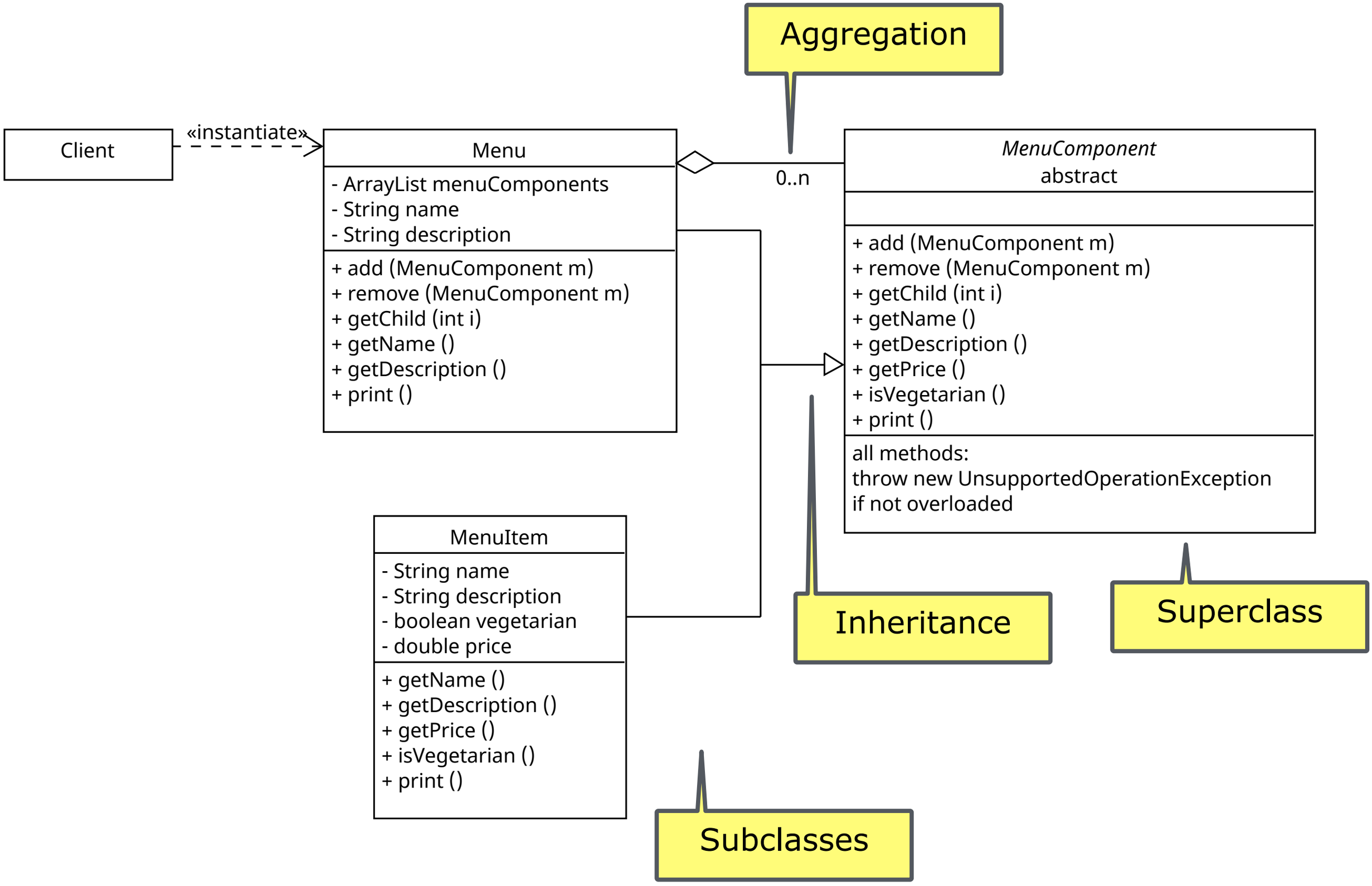
- Imagine that you want to model all of a resort hotel's menus.
- There are 3 restaurant menus: Pancake House, Diner, and Cafe.
- As well as menu items, the Diner menu has a sub-menu: Dessert.
- As well as menu items, the Cafe menu has a sub-menu: Coffe.
- The following should be stored for each menu item: name, description, vegetarian yes/no, price.

Hierarchy of menus and menu items.

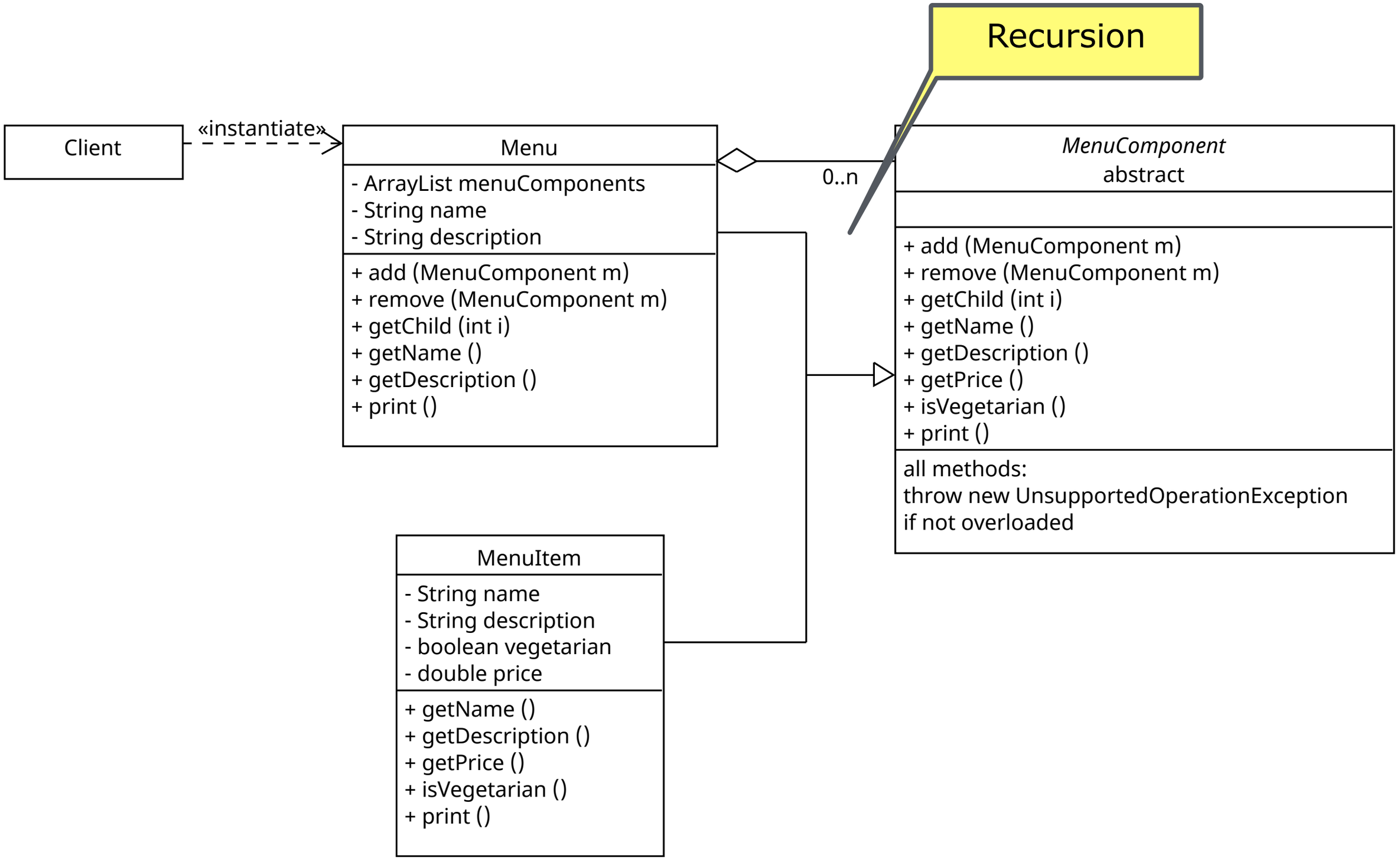
Example Solution



Example Solution



Example Solution



```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");
        MenuComponent coffeeMenu =
            new Menu("COFFEE MENU", "Stuff to go with your afternoon coffee");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem(
            "K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99));
…

        dinerMenu.add(new MenuItem(
            "Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat",
            true,
            2.99));
…
```

```
        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true,
            1.59));
…

        cafeMenu.add(new MenuItem(
            "Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true,
            3.99));
…

        cafeMenu.add(coffeeMenu);

        coffeeMenu.add(new MenuItem(
            "Coffee Cake",
            "Crumbly cake topped with cinnamon and walnuts",
            true,
            1.59));
…

        Waitperson waitperson = new Waitperson(allMenus);

        waitperson.printMenu();
    }
}
```

Add an item

Add an menu

```java
package headfirst.composite.menu;

import java.util.*;

public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Abstract superclass

Methods that will be overloaded by the subclasses

```java
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

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

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

Only overloads the relevant methods

```java
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

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

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

List of components. Each component could be an item or a menu.

Only overloads the relevant methods

Iterator deals with listing all components of this menu

Design Patterns: Decorator

Decorator

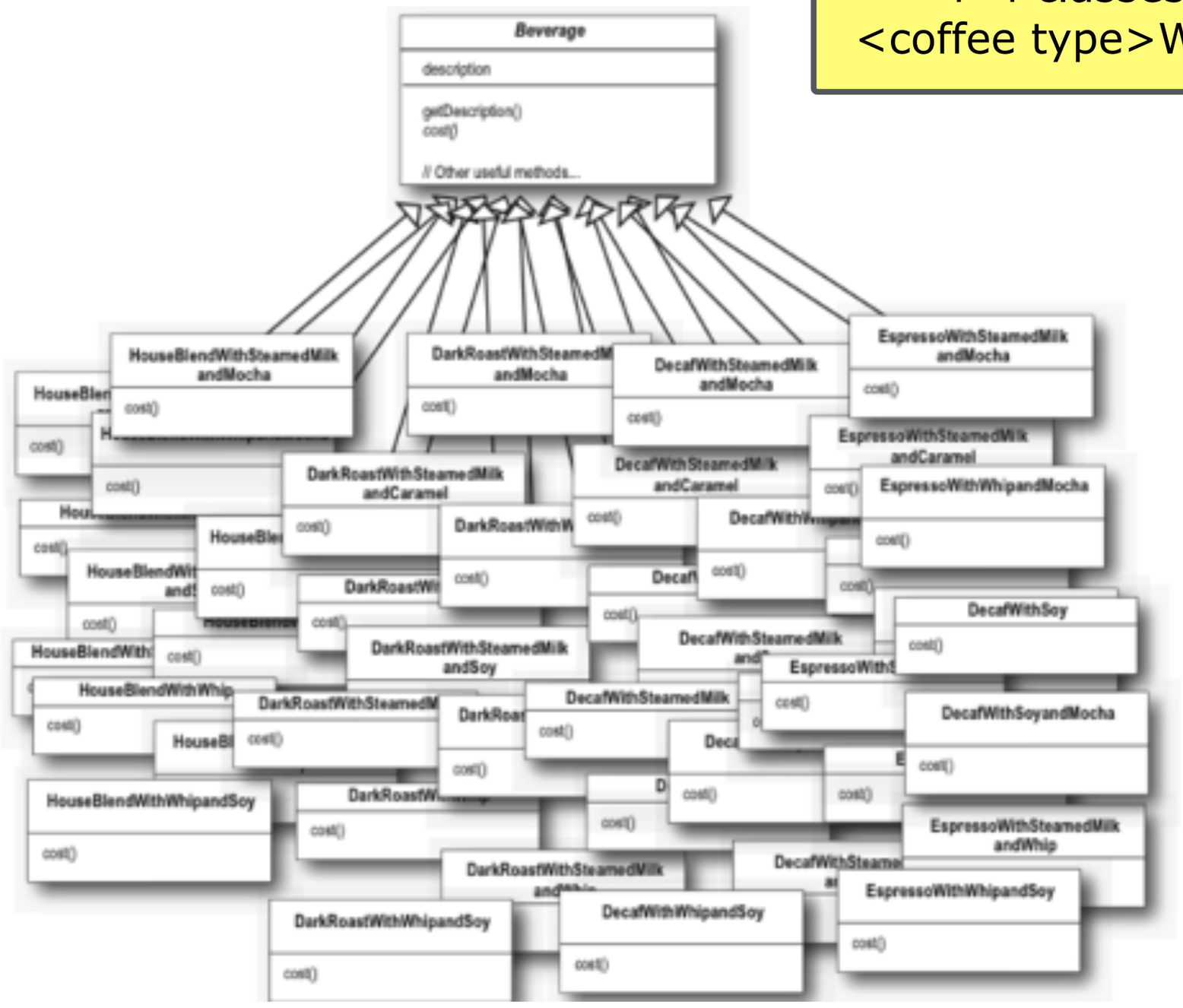
- A Decorator pattern allows behaviour to be added to an individual object without affecting the behaviour of other objects from the same class.

Example Problem

- Imagine that you want to model the beverages sold by the coffee shop.
- There are 4 basic coffee types with different costs: house blend, dark roast, expresso and decaf.
- There are 4 condiments that can be added to any of the basic coffee types: milk, mocha, soy, and whip.
- Your model should allow for additional condiments to be added.

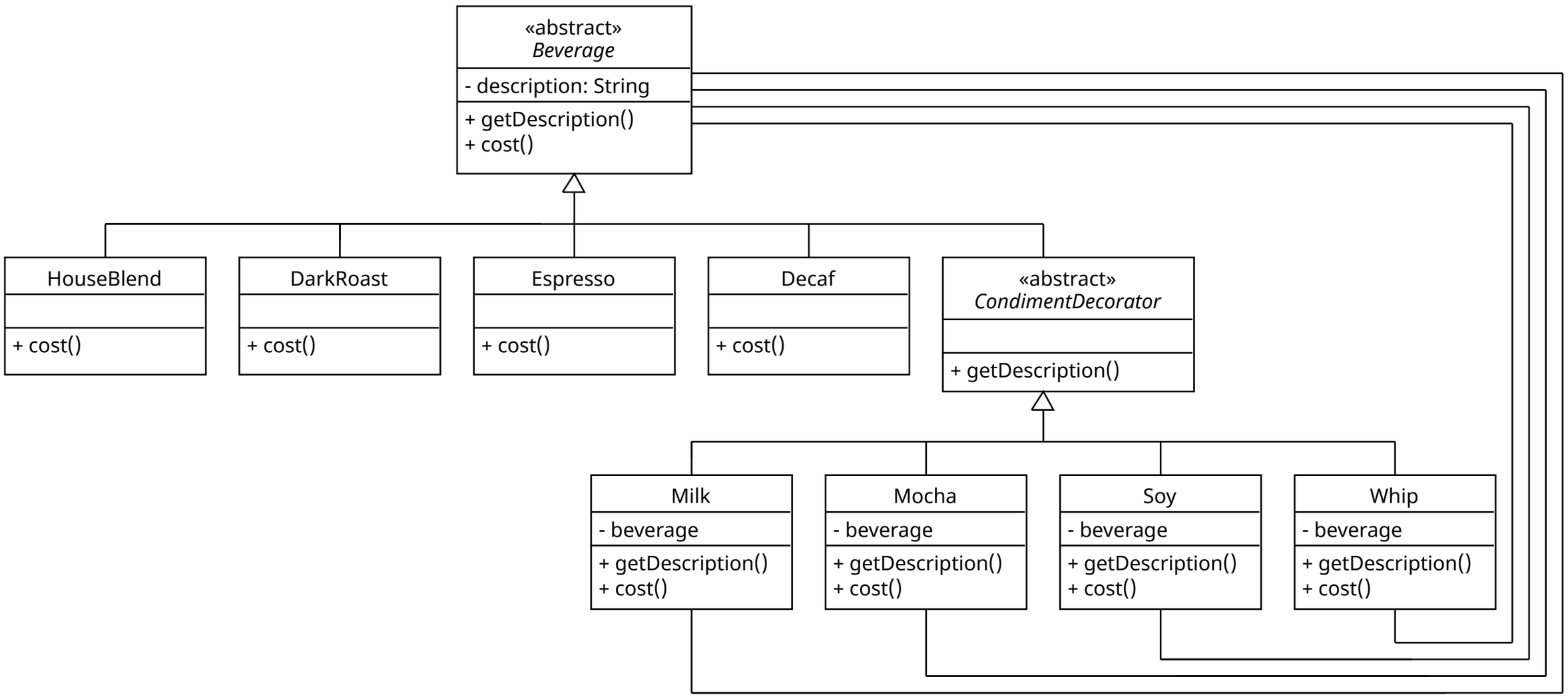
Without Decorators

4*4 classes each named
<coffee type>With<condiment>



[Freeman, et al. “Design Patterns, Head First”]

With Decorators



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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract parent class

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

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

HouseBlend is an concrete child class of Beverage

Similar concrete classes are implemented for DarkRoast, Decaf and Expresso

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

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

The abstract class declares the field.

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

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

The concrete class constructor overrides the field value

Fixed value. No field needed.

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

The CondimentDecorator is an abstract child class of Beverage

Milk is a grandchild class of Beverage so Milk can be treated as a Beverage

```java
public class Milk extends CondimentDecorator {
    Beverage beverage;

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

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

    public double cost() {
        return .10 + beverage.cost();
    }
}
```

Keeps a reference to a Beverage instantiation which this object builds on

Concatenates the previous description

Increases the previous cost

Similar concrete classes are implemented for Mocha, Soy and Whip

```
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());
    }
}
```

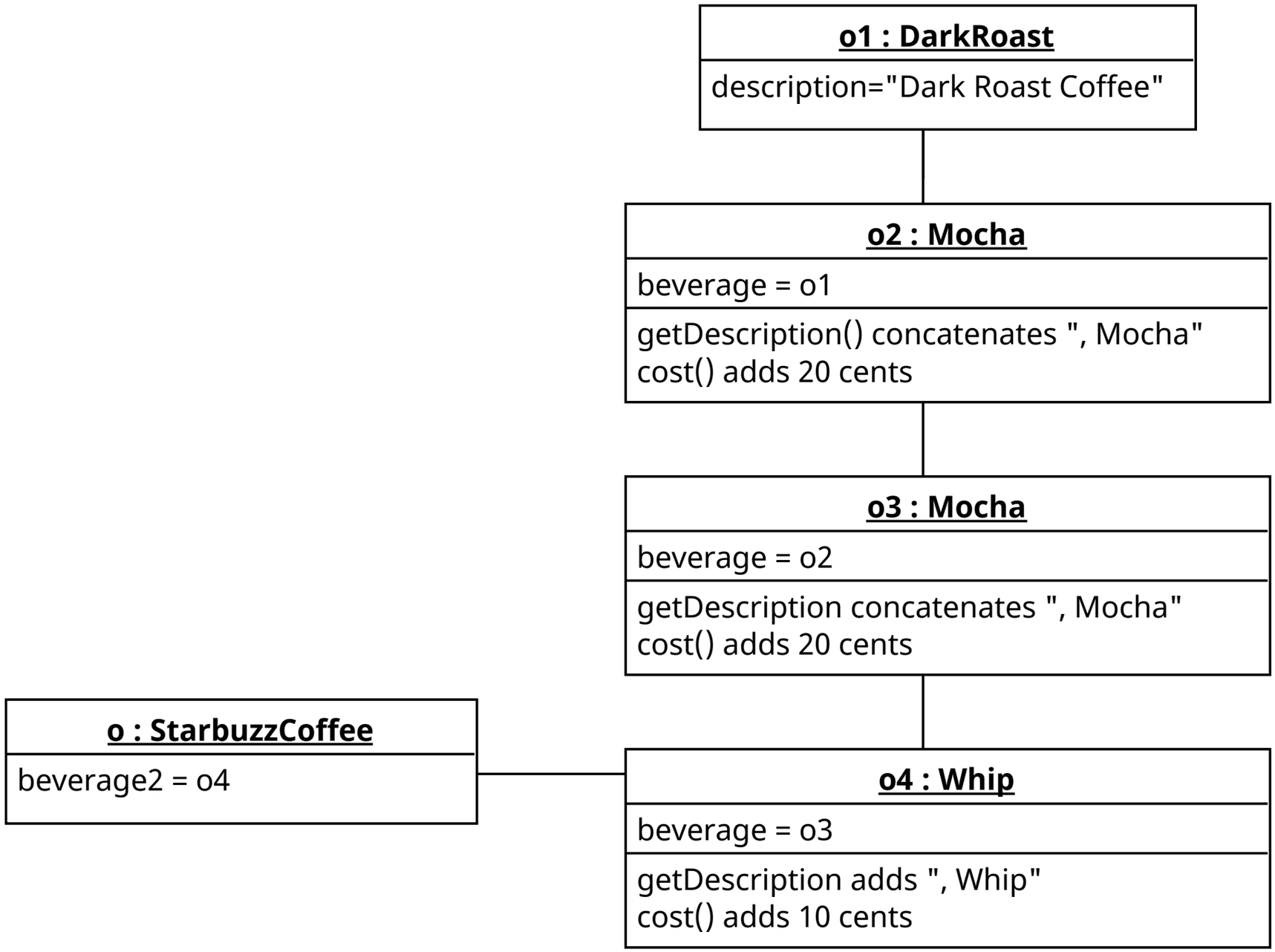
```
public class StarbuzzCoffee {

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

Basic beverage

Creates a new object building on the previous

Beverage with selection of condiments

UML Object Diagram



Design Patterns: Singleton

Singleton

- A Singleton pattern ensures that a class can only be instantiated once within a program.
- IMHO, the Singleton pattern should only be used when a class must be instantiated only once, not when a class is only instantiated once.

Design Patterns: Factory

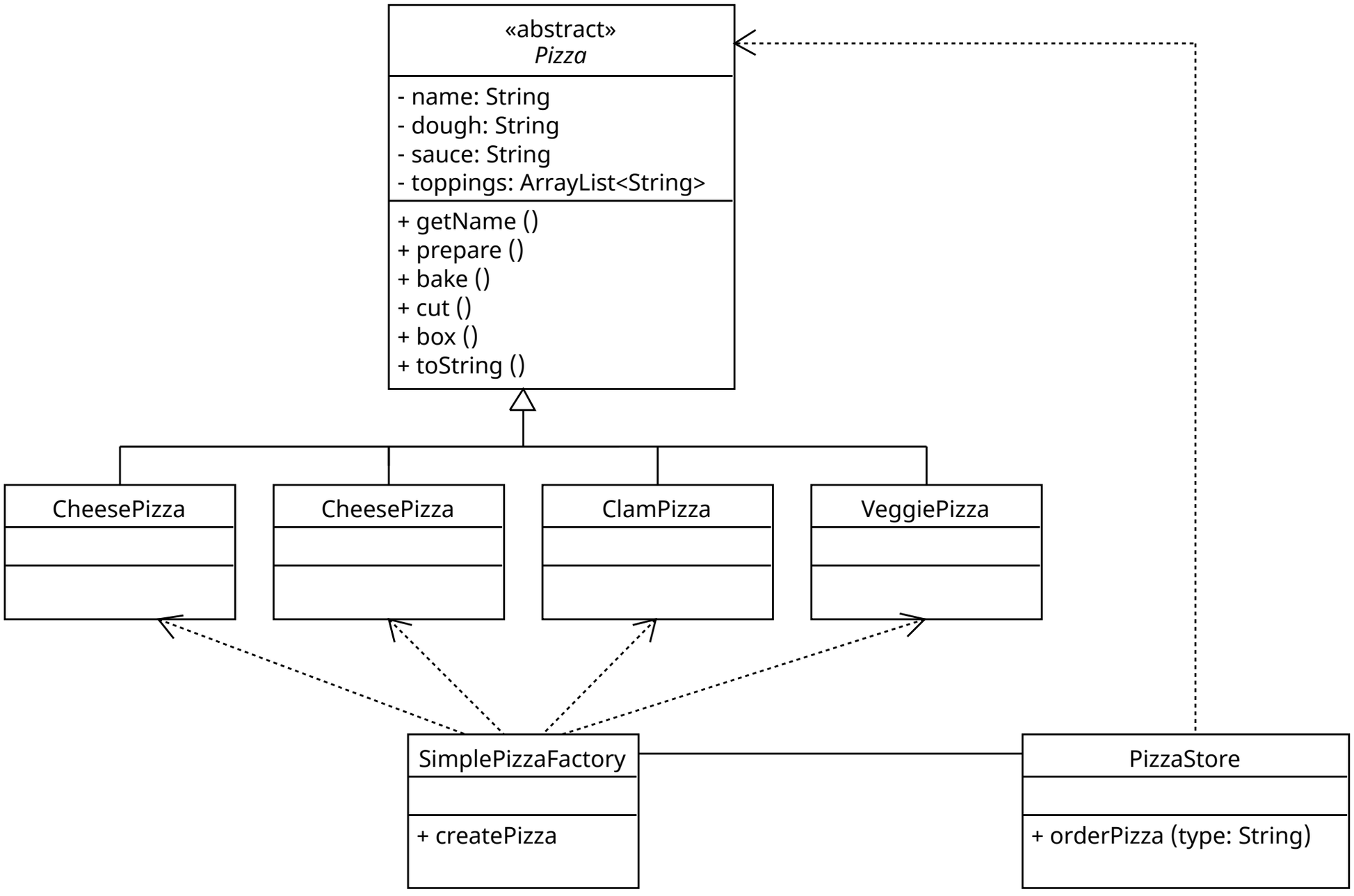
Factory

- A Factory pattern moves the specifics of class instantiation from the client class to a factory classes. It is typically used when the client must select the class to instantiate from a number of alternative subclasses.
- The main advantage is the the factory hides the details of class instantiation from the client.

Example Problem

- Imagine that you need to model pizzas being made in a pizza store.
- There are 4 types of pizza: cheese, clam, pepperoni, veggie.
- There may be more types of pizza in future.
- For each pizza type, the following information must be captured: name, dough, sauce, toppings.
- To fulfil a pizza order, the following 4 steps must be performed: prepare, bake, cut, box.

With Factory



Pizza superclass

```
abstract public class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();

    public String getName() {
        return name;
    }

    public void prepare() {
        System.out.println("Preparing " + name);
    }

    public void bake() {
        System.out.println("Baking " + name);
    }

    public void cut() {
        System.out.println("Cutting " + name);
    }

    public void box() {
        System.out.println("Boxing " + name);
    }

    public String toString() {
        // code to display pizza name and ingredients
        StringBuffer display = new StringBuffer();
        display.append("---- " + name + " ----\n");
        display.append(dough + "\n");
        display.append(sauce + "\n");
        for (int i = 0; i < toppings.size(); i++) {
            display.append((String )toppings.get(i) + "\n");
        }
        return display.toString();
    }
}
```

Concrete subclass of Pizza

Similar for CheesePizza, ClamPizza, and VeggiePizza

```
public class PepperoniPizza extends Pizza {
    public PepperoniPizza() {
        name = "Pepperoni Pizza";
        dough = "Crust";
        sauce = "Marinara sauce";
        toppings.add("Sliced Pepperoni");
        toppings.add("Sliced Onion");
        toppings.add("Grated parmesan cheese");
    }
}
```

Pizza Factory

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Depending on the string argument, a different Pizza subclass with be instantiated.

Returns the instantiated pizza

Client

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}
```

Uses the factory to instantiate the pizza subclass

Cooks the pizza

Test program

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza pizza = store.orderPizza("cheese");
        System.out.println("We ordered a " + pizza.getName() + "\n");

        pizza = store.orderPizza("veggie");
        System.out.println("We ordered a " + pizza.getName() + "\n");
    }
}
```

Instantiates the factory

Provides the factory to the store

The store instantiates the pizza subclass

Without Factory

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}
```

Design Patterns: Observer

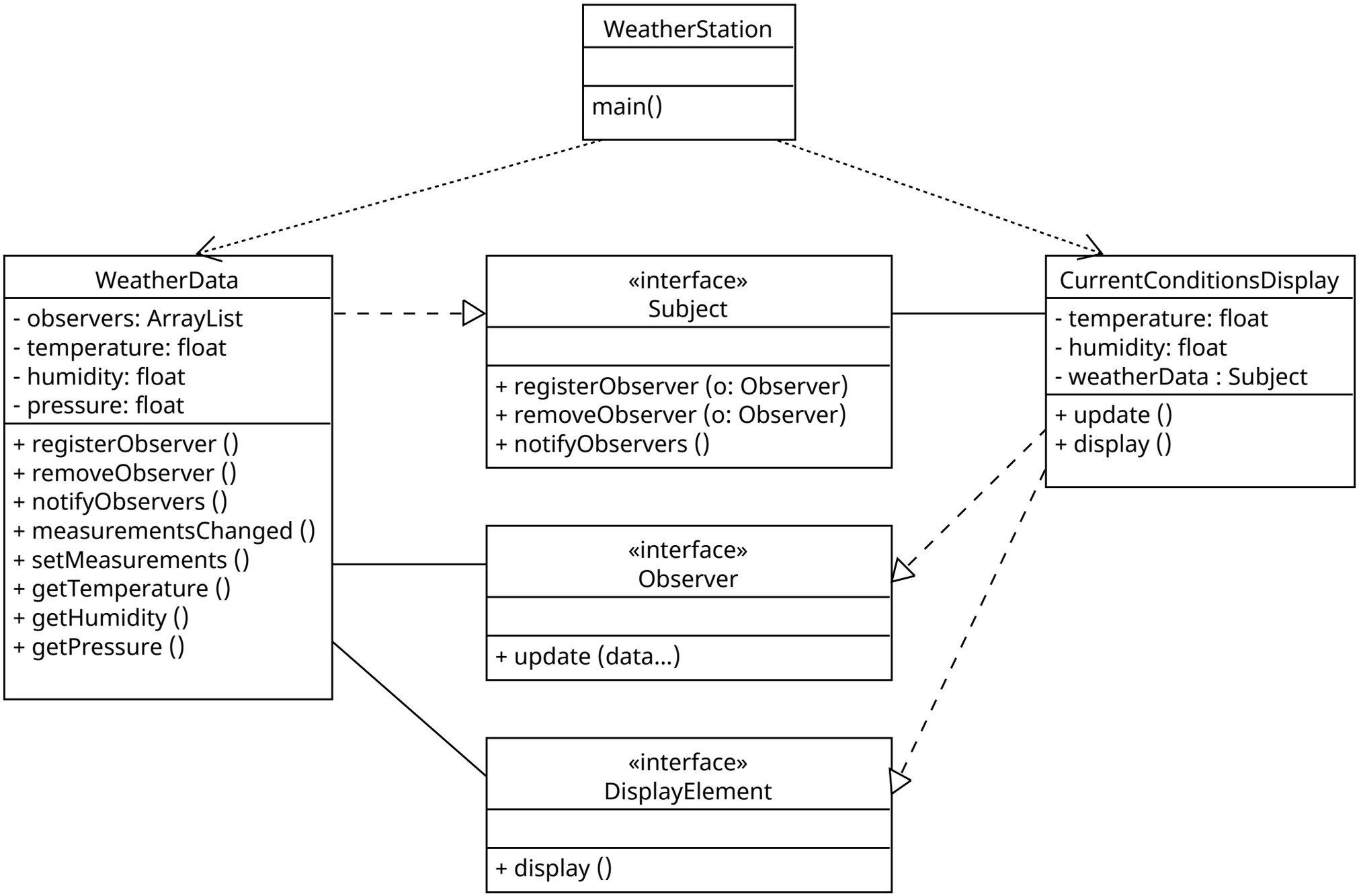
Observer

- An Observer pattern ensures that, when an object changes state (the subject), all its dependents (the observers) are notified and updated automatically.
- Commonly used in Module-View-Controller architectures to allow asynchronous View updates when the state of the Model changes.

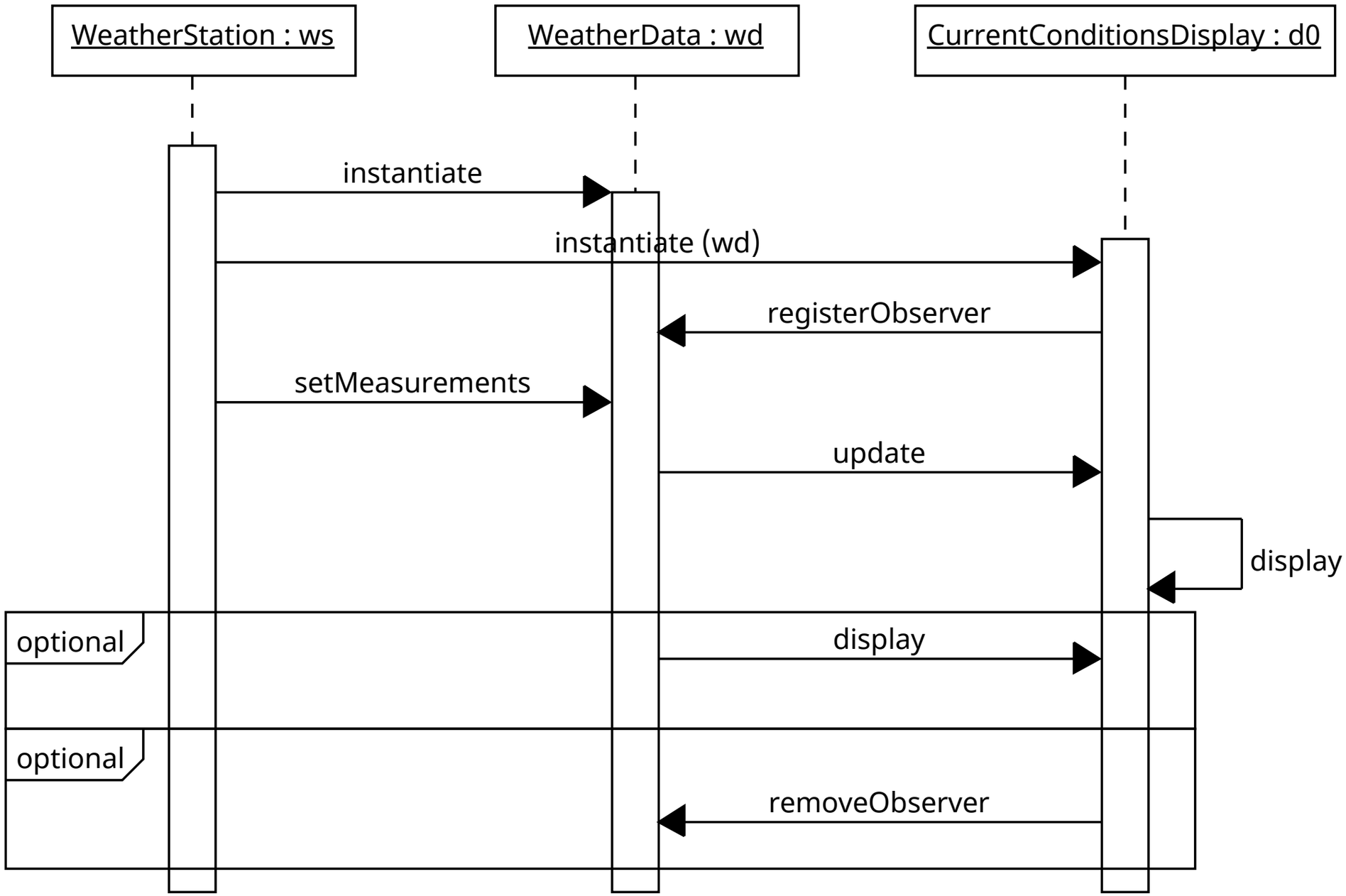
Example Problem

- Imagine that you have to implement a weather station app. The app reads current weather data from sensors and displays the current conditions, the resulting forecast, and the long term statistics.
- Using the MVC architecture, there should be separate classes for the weather data and the 3 display classes.
- How are the display classes updated when new sensor measurements are made?

With Observer



With Observer



Main

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

Instantiate subject

Instantiate observers providing a reference to the

New measurements cause the subject to notify the observers. The observers then display the new data or new results based on the new data.

Interfaces

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

```
public interface DisplayElement {
    public void display();
}
```

```java
public class WeatherData implements Subject {
     private ArrayList observers;
     private float temperature;
     private float humidity;
     private float pressure;

     public WeatherData() {
          observers = new ArrayList();
     }

     public void registerObserver(Observer o) {
          observers.add(o);
     }

     public void removeObserver(Observer o) {
          int i = observers.indexOf(o);
          if (i >= 0) {
               observers.remove(i);
          }
     }

     public void notifyObservers() {
          for (int i = 0; i < observers.size(); i++) {
               Observer observer = (Observer)observers.get(i);
               observer.update(temperature, humidity, pressure);
          }
     }

     public void measurementsChanged() {
          notifyObservers();
     }

     public void setMeasurements(float temperature, float humidity, float pressure) {
          this.temperature = temperature;
          this.humidity = humidity;
          this.pressure = pressure;
          measurementsChanged();
     }

     public float getTemperature() {
          return temperature;
     }

     public float getHumidity() {
          return humidity;
     }

     public float getPressure() {
          return pressure;
     }
}
```

Subject

Allows observer registration

The argument is the observer object

Allows observer removal

Notifies registered observers by calling their update methods

```java
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

Receives reference to the subject

Registers as an observer

The subject calls update. The method updates local fields and displays the latest data.

Similar in StatisticsDisplay and ForecastDisplay

Open Source Software

Open Source

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open Source

- Important open source systems:
  - Linux: open source implementation of the Unix operating system
  - Java: open source programming language, platform and tools. Originally developed by Sun and now by Oracle.
  - Eclipse: open source integrated development environment (IDE) for multiple programming languages
  - Git: open source version control system
  - Apache: open source web server
  - mySQL: open source database system

Open Source Licensing

- Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
- Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
- Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

Open Source Licensing

- The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

Pair Programming

Pair Programming

- This is an Agile technique.
- Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair). The programmer at the keyboard is usually called the *driver*, the other, also actively involved in the programming task but focusing more on overall direction is the *navigator.* it is expected that the programmers swap roles every few minutes or so.

Pair Programming

- Benefits
  - Higher quality: code review as you go
  - Better sharing of knowledge
  - Better transfer of skills
- Possible pitfalls:
  - Lazy navigator
  - Driver doesn’t talk
  - Interpersonal relationship needs to work
- In experiments:
  - - productivity
  - +++ quality
  - + team relationships
  - + knowledge sharing

Pair Programming

- Tips:
  - Share everything (about the program!). “We” not “I”.
  - Play fair
    - ❖Take turns driving
    - ❖The navigator should be reviewing and planning (not resting)
  - Both programmers needs to stay focused
    - ❖Take planned breaks from pair programming
    - ❖Resync individual work after a break
  - Treat it as a discussion by peers
    - ❖Don’t be defensive
    - ❖Don’t be superior
  - Tidy up as you go
  - Don’t take things too seriously
  - Sit side-by-side

Pair Programming

- Recommended reading:
  - Williams, Kessler, Cunningham and R. Jeffries, Strengthening the Case for Pair Programming, IEEE Software, 17, 4, 2000.
  - Hannay, Dybå, Arisholm, and Sjøberg, The Effectiveness of Pair Programming: a Meta Analysis, Information and Software Technology, 51, 7, 2009.
  - Böckeler and Siessegger, On Pair Programming, https://martinfowler.com/articles/on-pair-programming.html, 2020.

Summary

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- Design patterns can be re-used to improve code quality and to assist in communicating design intent.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.