

COMP 2511

Object Oriented Design & Programming

Week 07

Last week,

Generics, Collections

Use of anonymous inner classes

Iterator Pattern

This week

OO design principles

- Strive for loosely coupled objects

Design Patterns

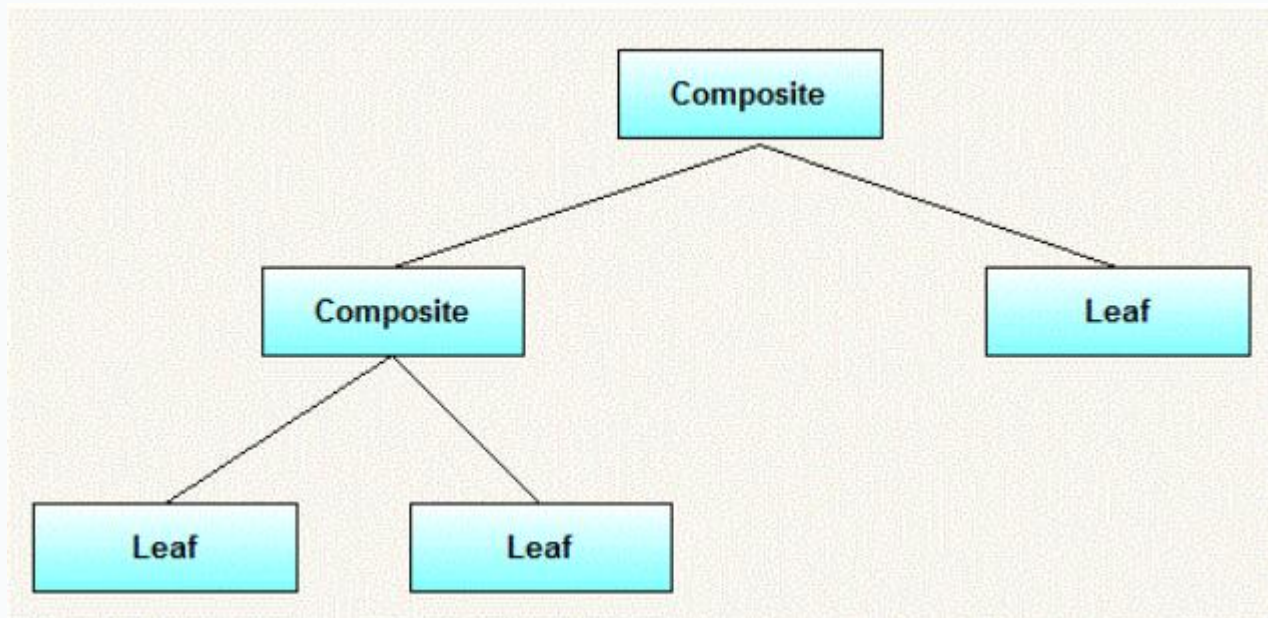
- Composite
- Decorator
- Observer

Pattern #5: Composite Pattern

- Motivation
 - Need a way to treat collections of objects as if they were a single (combined) whole e.g.,
File hierarchies:
 - Directories are composed of files and directories
 - Printing a directory requires printing each sub-directory
- Intent
 - To build structures of objects in the form of trees, that contains both individual objects and compositions of objects as nodes
 - The composite object that has the same behaviour as its parts i.e. we can apply the same operations over both composites and individual objects

The Composite Pattern

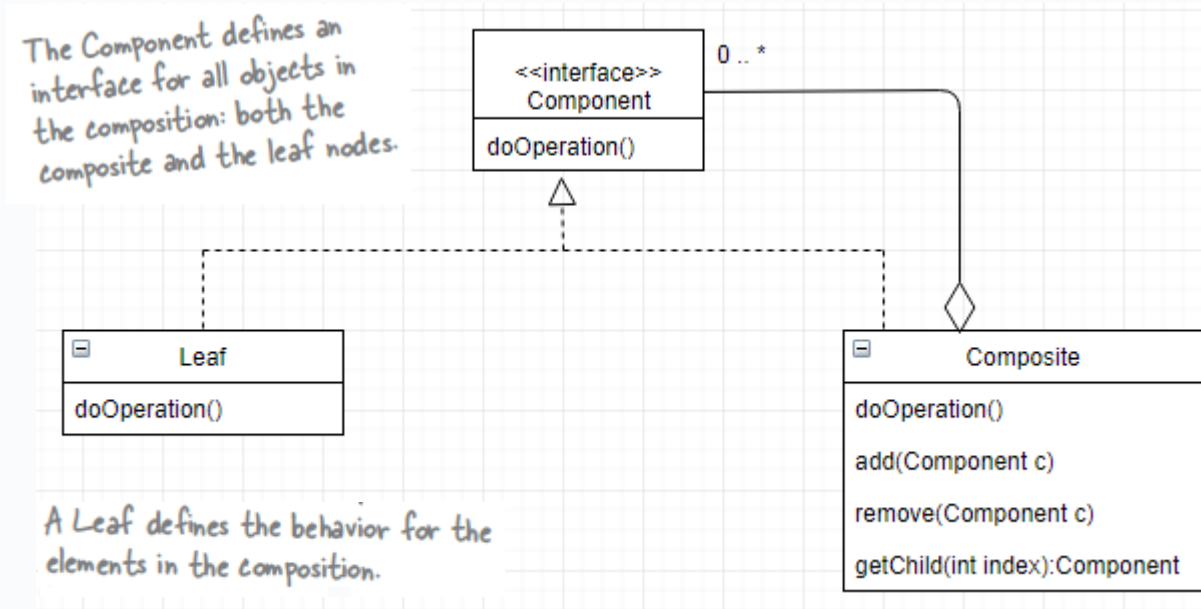
Design Pattern #5: The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



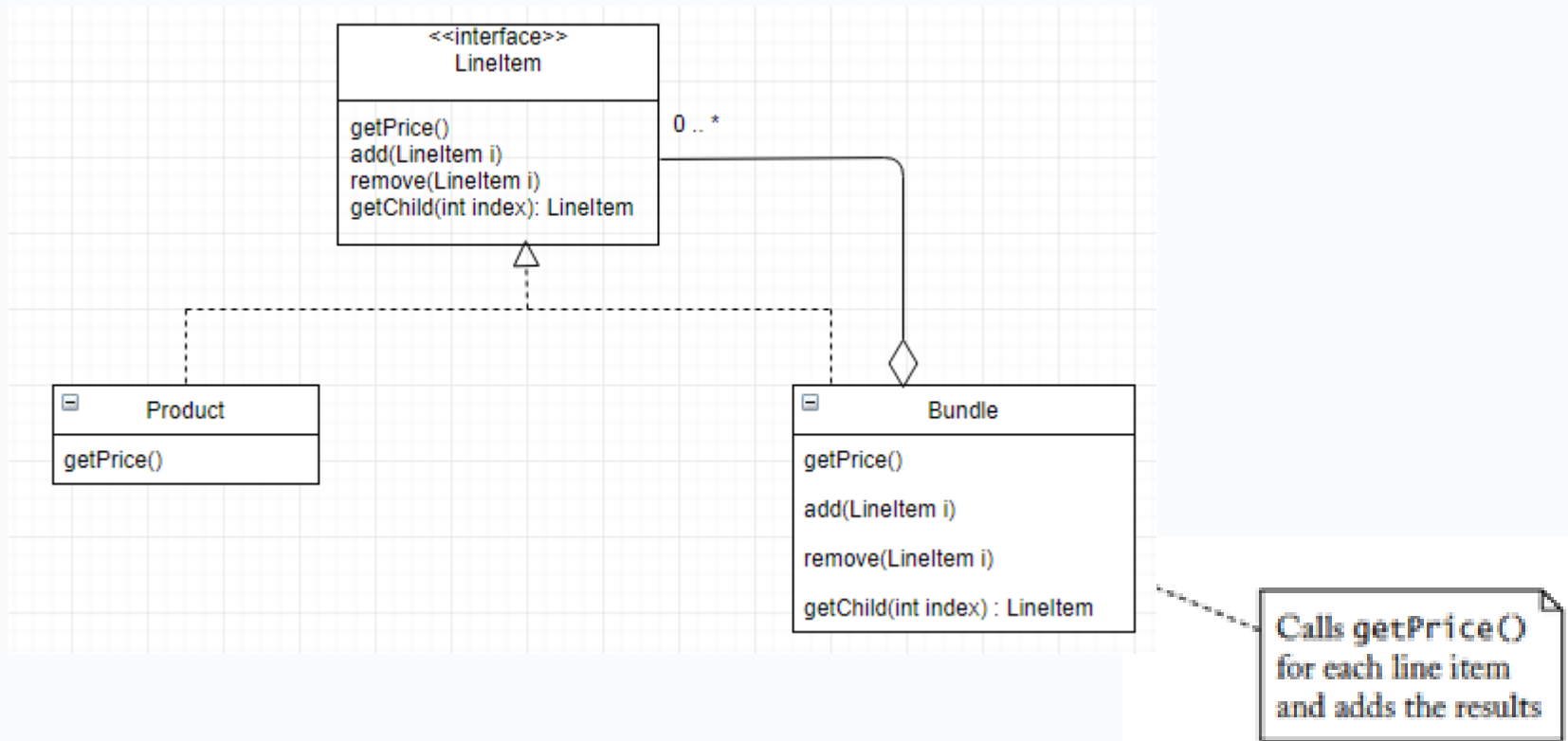
- A Leaf represents primitive objects in the composition and has no children
- A Composite stores child components

Composite Pattern Implementation

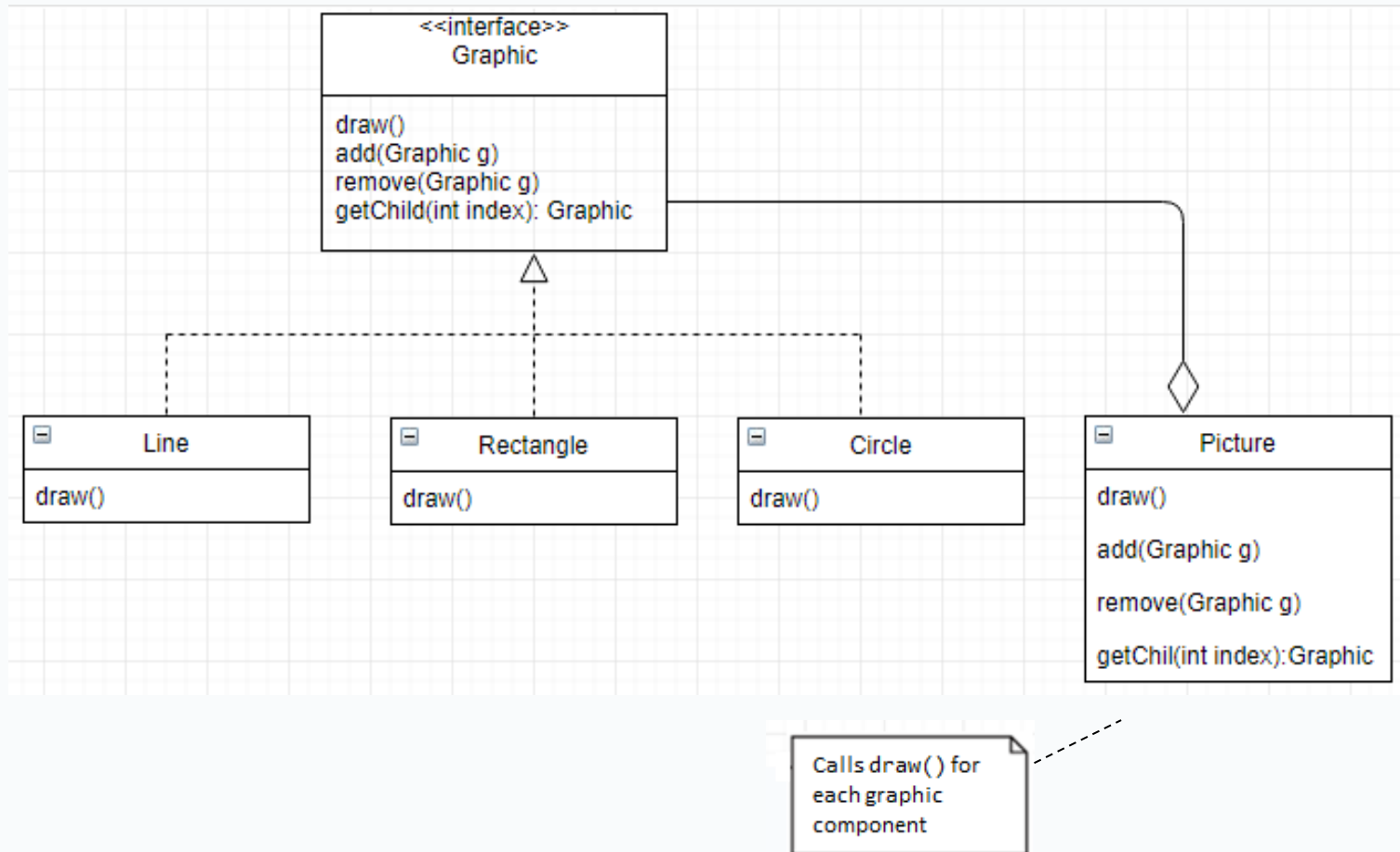
- Define an **interface type** that defines the default behaviour for both the **Leaf** and the **Composite**
- **Leaf** defines behaviour for the primitive objects in the composition
- **Composite** *contains* a collection of primitive objects
- Both the primitive and the composite *implement* the same interface type
- When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results.



Composite Pattern Example



Composite Pattern: Example

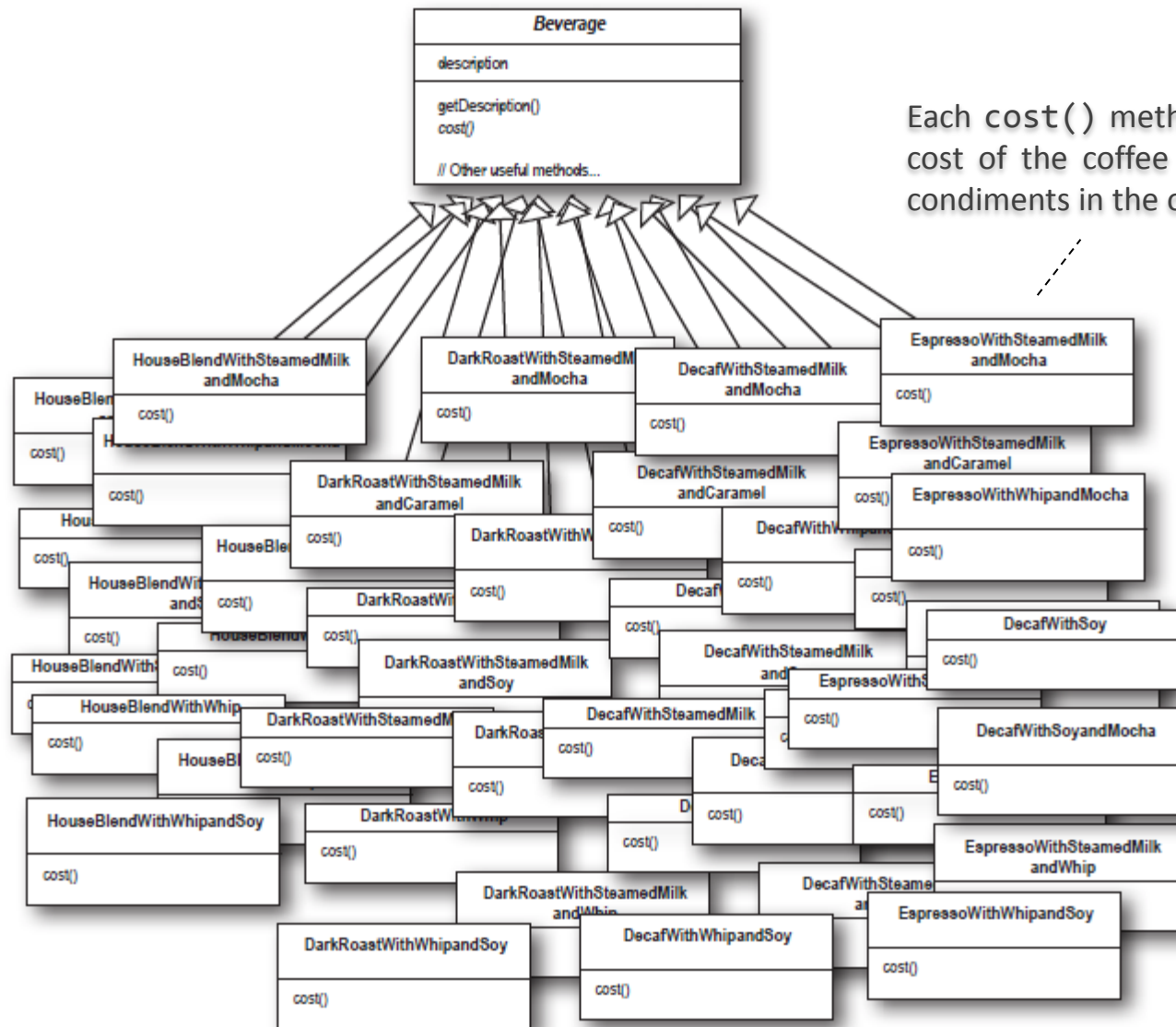


Decorator Pattern

Visit our Star Buzz café...

- Star Buzz wants to offer different variations of their beverage (e.g., soy, dark roast) and several kinds of condiments to add to the beverage e.g., Hazelnut, Vanilla, Whipped Cream
- Star Buzz charges an extra cost for each type of condiment added
- Star Buzz would like a flexible design that computes the total cost for different types of beverages with different condiments in the order
e.g. Dark Roast With Soy & Caramel
Decaf With Soy & Whipped Cream

First attempt at the design...



Each `cost()` method computes the cost of the coffee along with other condiments in the order

- An explosion of classes
 - What happens when the price of any condiment changes?
- ... A maintenance nightmare

Revise Open Closed Principle (OCP)

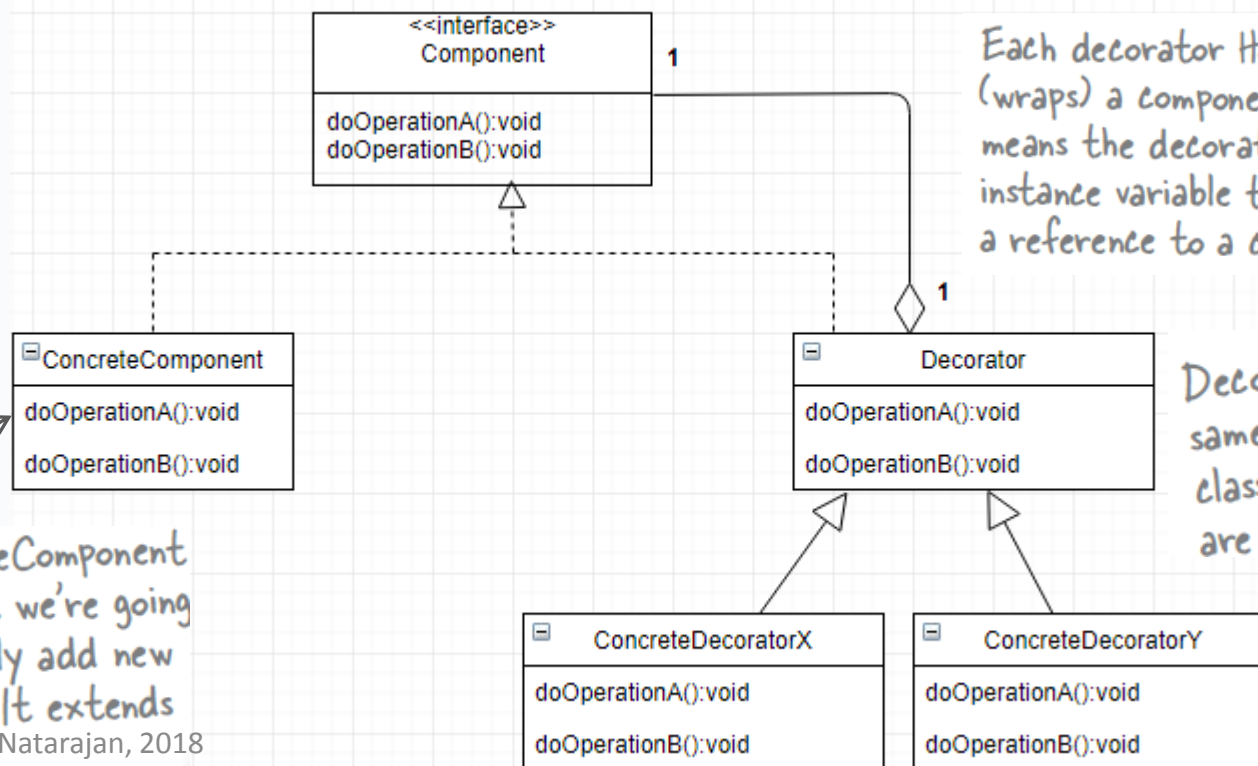
- Classes must be closed for modification
 - This reduces the possibility of breaking existing tried and tested code
- Classes should be open for extension
 - Allow classes to be easily extended to incorporate new behaviour
- OCP promotes designs that are resilient to change but flexible to take on new requirements
- But our previous design
 - Demonstrated an explosion of classes
 - Every time, the price of a condiment changed, all the affected classes need to be modified (Does not conform to OCP)

Pattern #6: Decorator Pattern

- Motivation
 - Need a way to enhance a component's functionality dynamically at runtime, yet use the enhanced component the same way as the plain component
 - The component class does not want to take responsibility for the decoration
 - There may be an open-ended set of possible decorations
e.g. add a border, scroll-bar etc... to any GUI component
- Intent
 - The Decorator Pattern attaches additional responsibilities to an object dynamically.
 - Decorators provide a flexible alternative to inheritance for extending functionality.

Decorator Pattern Implementation

- Define an **interface type** that is an abstraction for the component object
- Create **concrete component classes** and **decorator classes** that implement this interface
- The decorator class *manages* the decoration for the component object
- The component objects can be decorated dynamically at run-time with as many decorators as we like

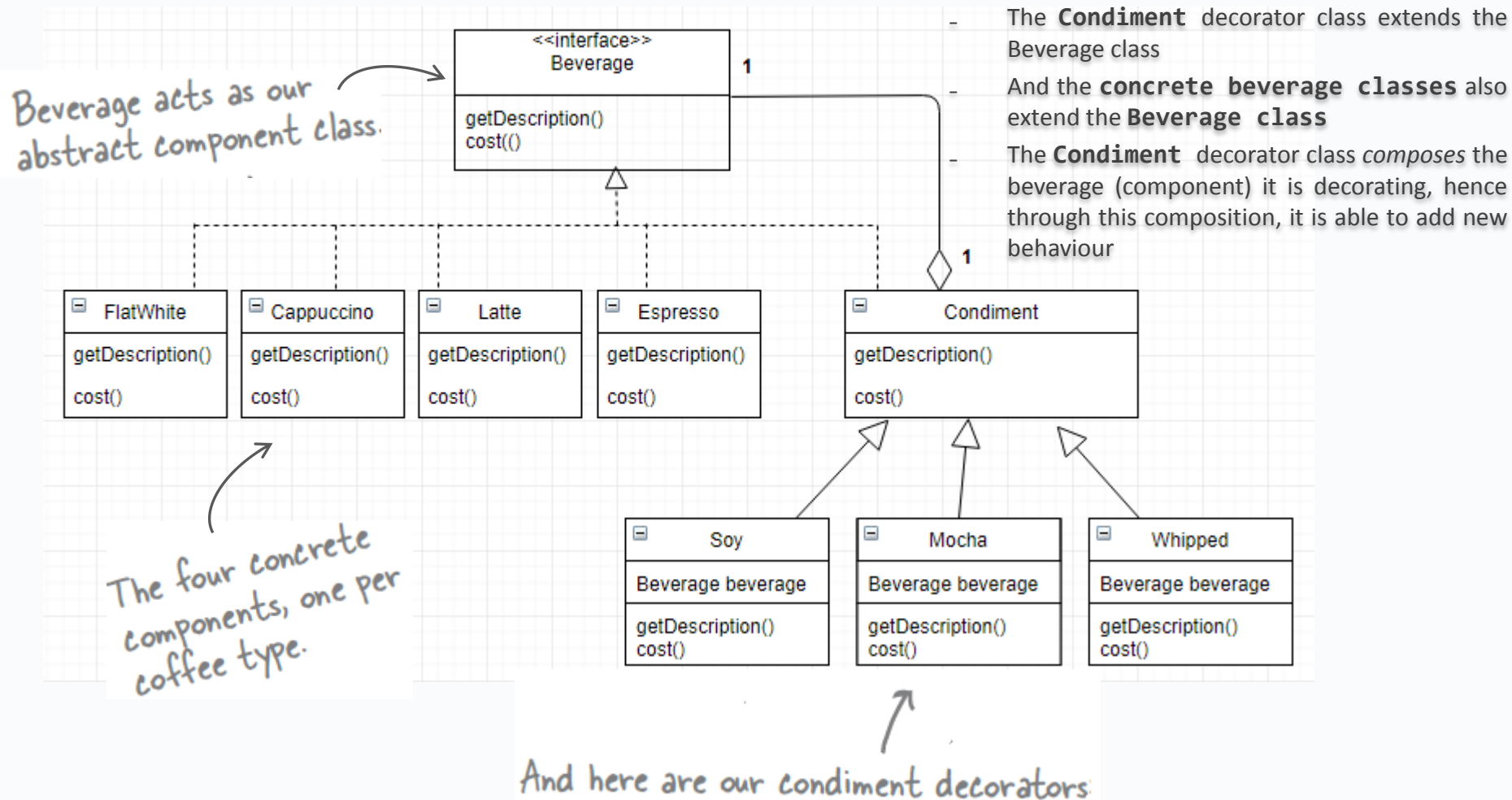


Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

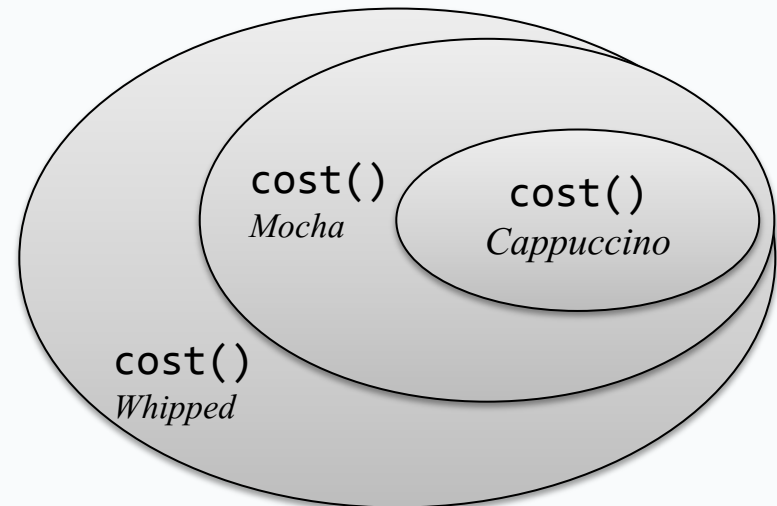
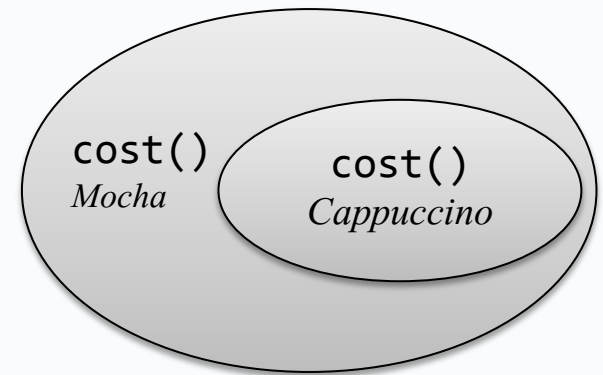
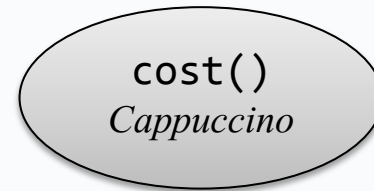
The **ConcreteComponent** is the object we're going to dynamically add new behavior to. It extends **Component**.

Decorator Pattern Implementation for our Beverage example

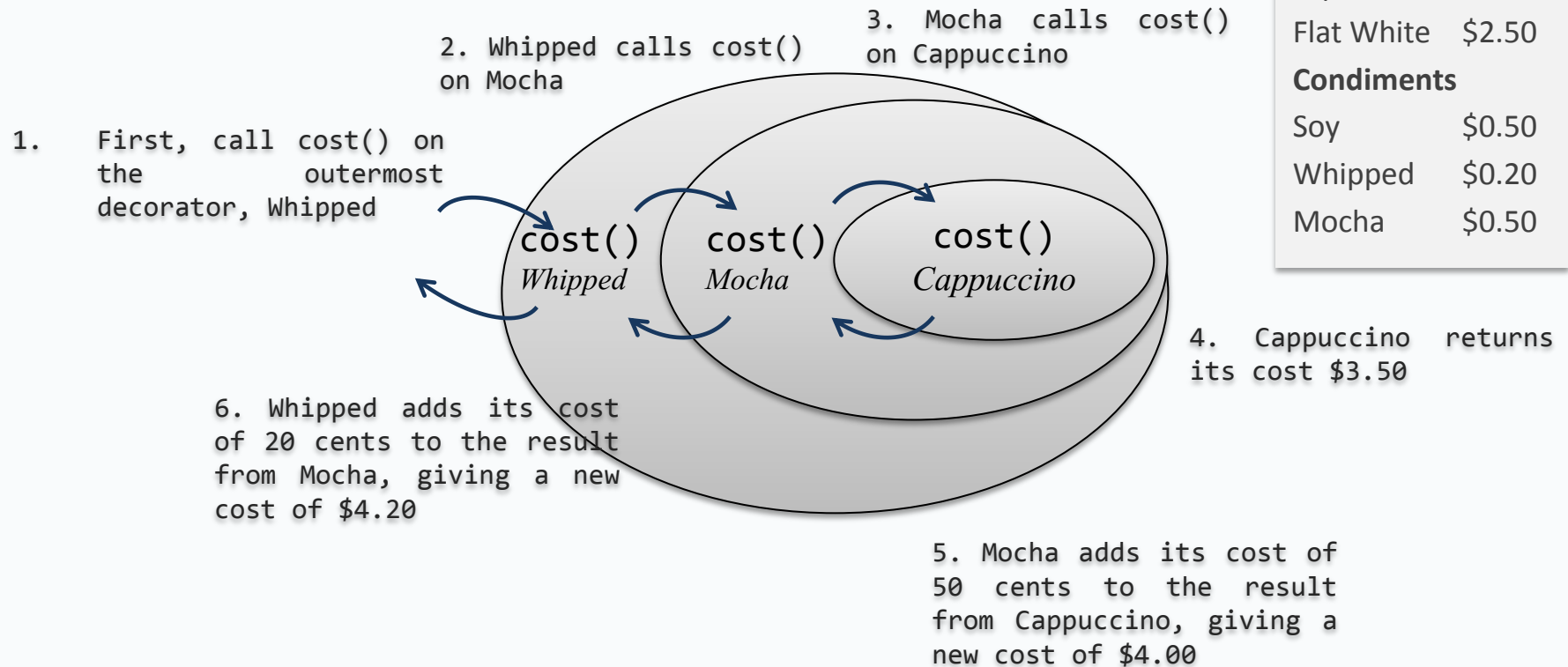


Using the Decorator Pattern

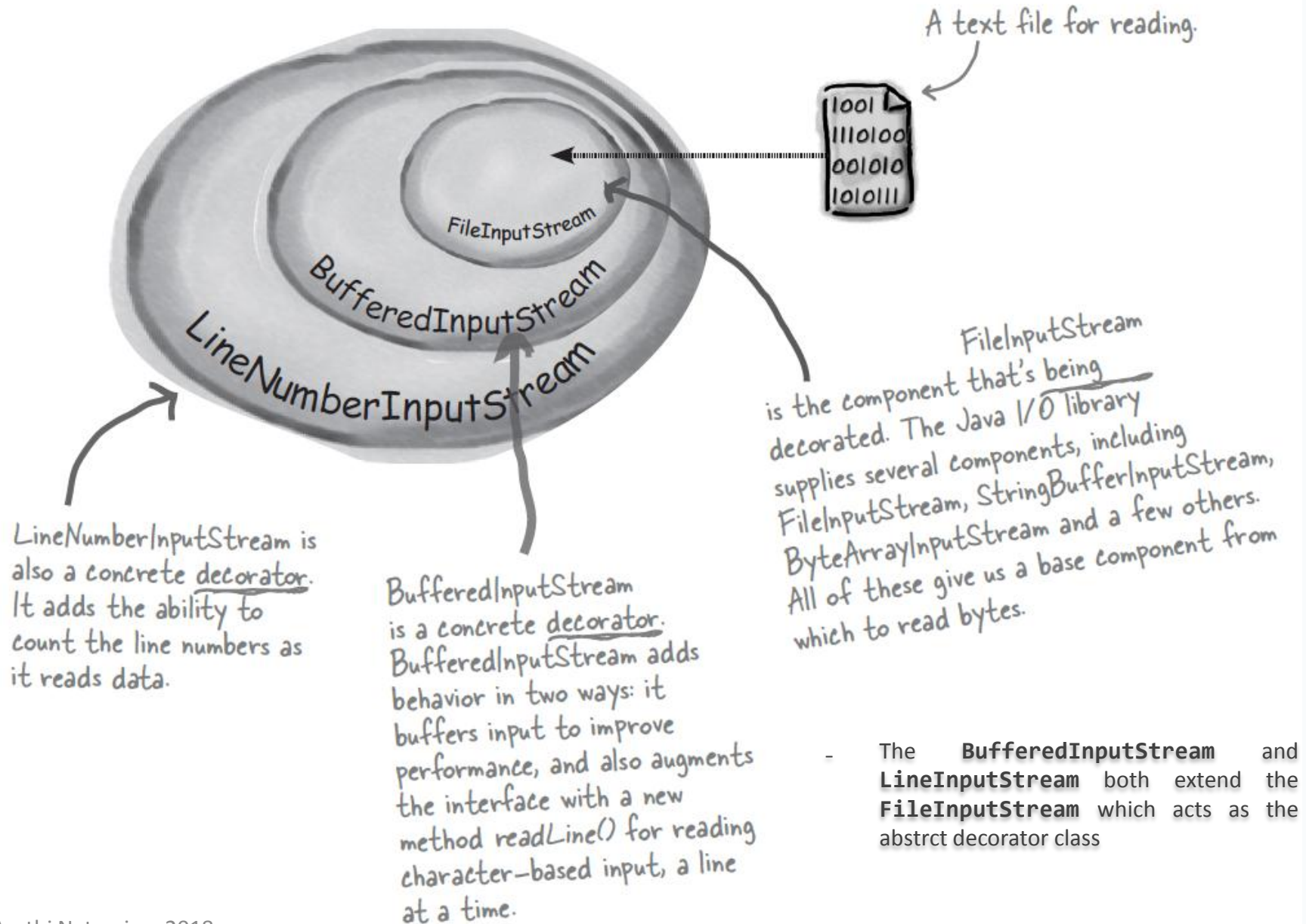
1. Start with a Cappuccino object
2. Customer wants to add mocha (chocolate), so create a Mocha object and wrap it around the Cappuccino object
3. Customer wants to add Whipped cream, so create a Whipped object and wrap it around Mocha with it



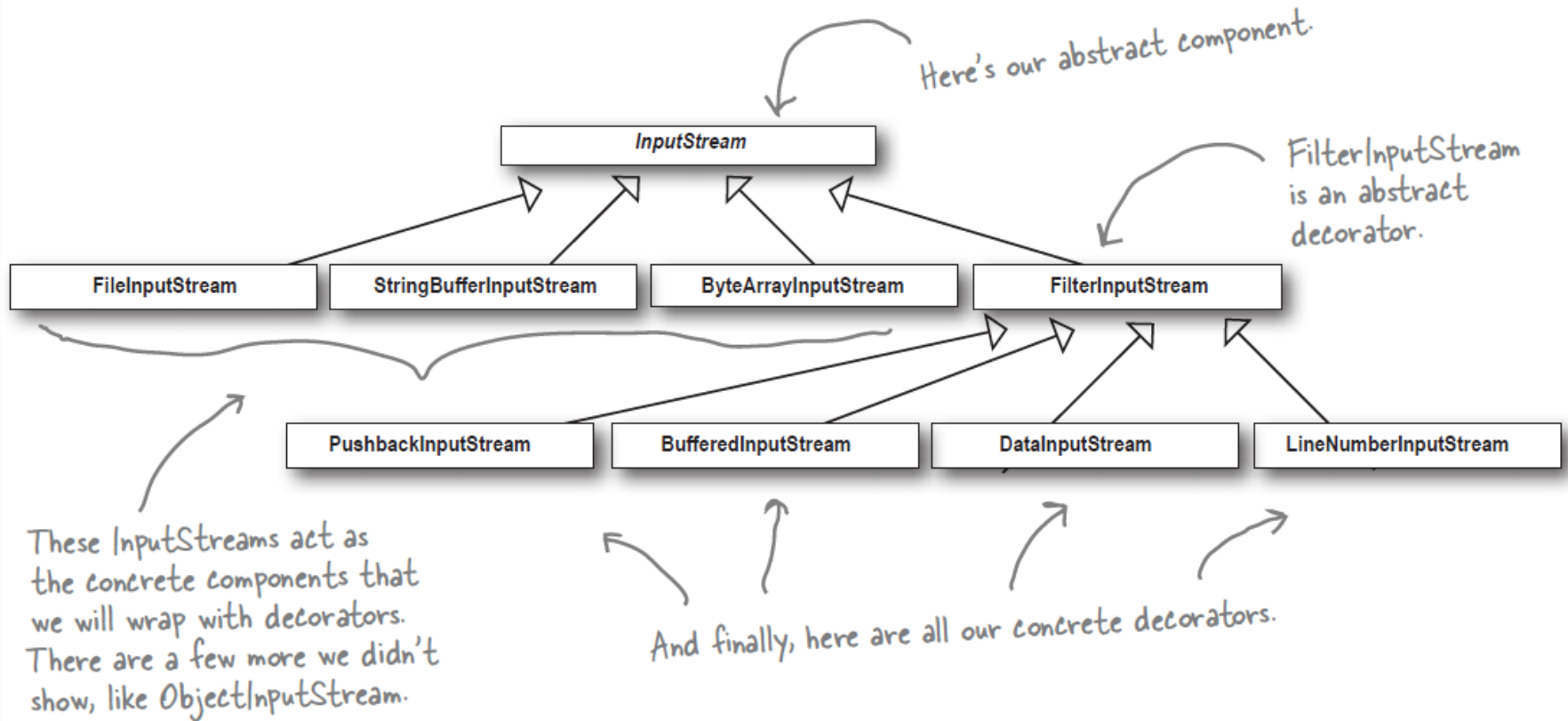
Using the Decorator Pattern



Use of Decorators in the Java API



Use of Decorators in the Java API



Implementation

```
public abstract class Beverage {
    private String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String desc) {
        this.description = desc;
    }

    public abstract double cost();
}

public class Cappuccino extends Beverage {

    public Cappuccino () {
        super.setDescription("Cappuccino");
    }

    public double cost() {
        return 3.00;
    }
}
```

```
public class Mocha extends Condiment {

    // Mocha decorator composes the beverage
    // it needs to decorate
    private Beverage beverage;
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

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

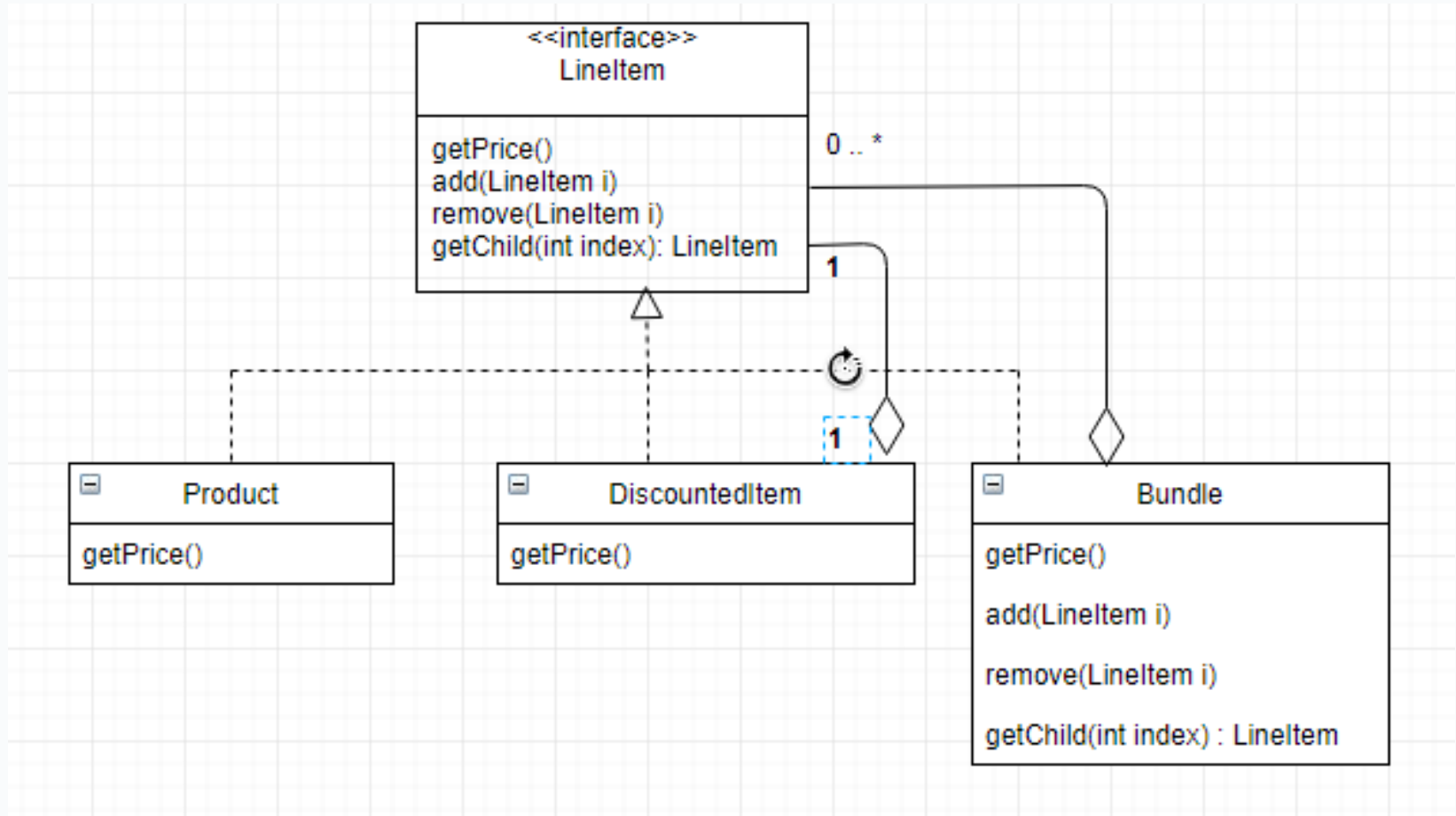
    @Override
    public double cost() {
        return 0.50 + beverage.cost();
    }
}
```

```
public class StarBuzzCoffee {

    public static void main(String[] args) {
        Beverage b1 = new Cappuccino();
        System.out.printf
        ("Cost of %s is: %.2f \n", b1.getDescription(), b1.cost());

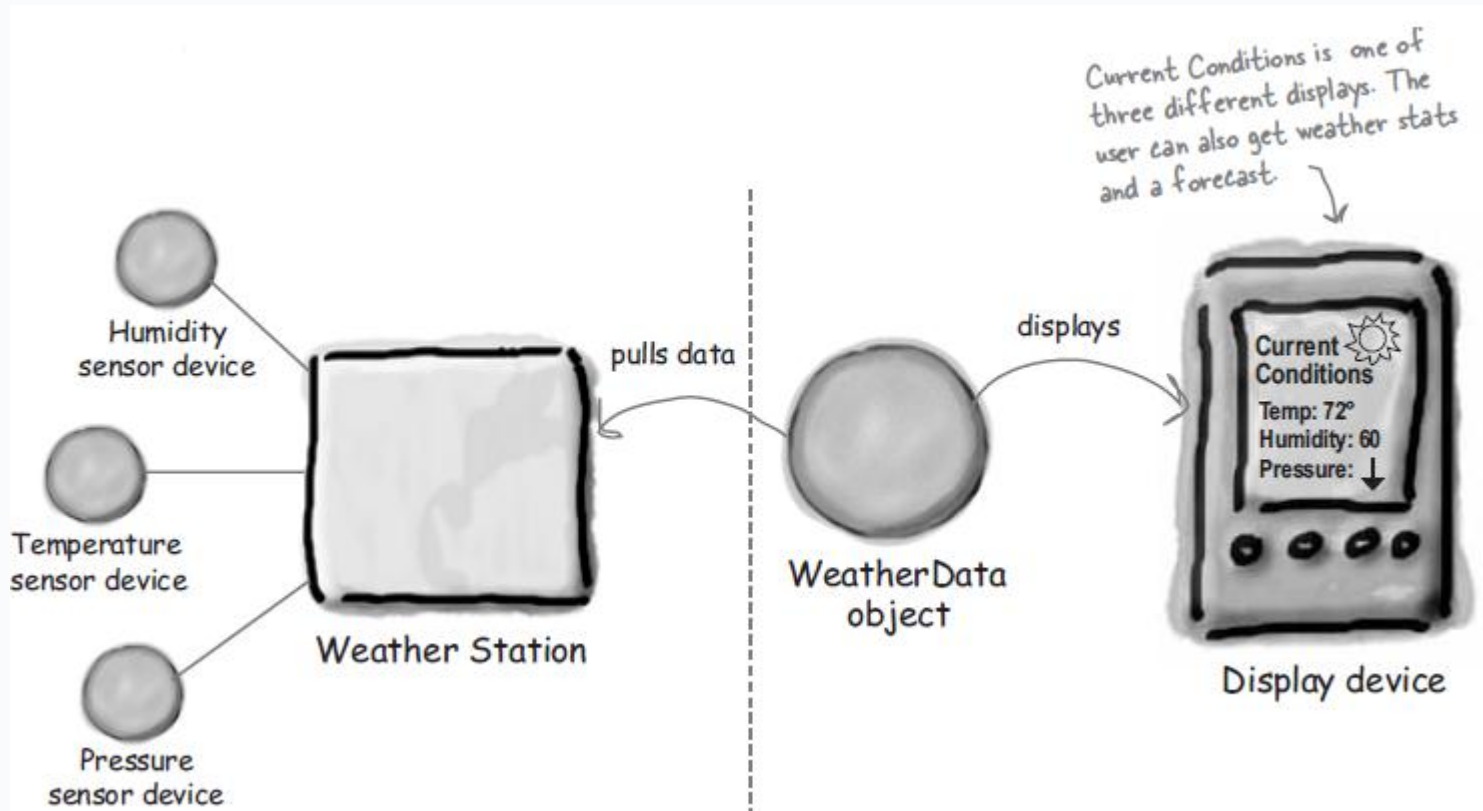
        Beverage b2 = new Cappuccino();
        // Add mocha
        b2 = new Mocha(b2);
        // Add whipped
        b2 = new Whip(b2);
        System.out.printf
        ("Cost of %s is: %.2f \n", b2.getDescription(), b2.cost());
    }
}
```

Combining composite and decorator patterns



Observer Pattern

A weather monitoring system



- A Weather Monitoring Application supports three different display devices: (1) Current Condition (2) Weather Statistics (3) Weather forecast
- Display is updated on the three devices by a Weather Data Object (WDO) which collects data from the weather station

The Weather Data Object class

```
public class WDO {  
  
    public float getTemperature() {  
    public float getHumidity() {  
  
    public float getPressure() {  
  
    /*  
    * This method updates three different display devices for current conditions,  
    * weather statistics and simple forecast  
    * This method is called any time new weather measurement is available  
    * We do not need to know, how or when this method will be called  
    */  
    public void measurementsChanged() {  
    .
```

Three getter methods class receive three measurement values: temperature, humidity and pressure

A method `measurementsChanged()` that will:


- be called whenever any of the above values change
- update the three display devices

- Our task is to implement the method `measurementsChanged()`
- System needs to be extensible in the future to accommodate more display devices
- Users should be able to dynamically add or remove display devices


What is the problem with this implementation?

```
public void measurementsChanged() {  
    float temp = getTemperature();  
    float pressure = getPressure();  
    float humidity = getHumidity();  
  
    //Now update the displays  
    currentConditionsDisplay.update(temp, humidity, pressure);  
    statisticsDisplay.update(temp, humidity, pressure);  
    forecastDisplay.update(temp, humidity, pressure);  
}  
}
```

1. Area of change that we need to encapsulate



2. By coding to concrete implementation we have no way to add or remove other display elements without making changes to the program

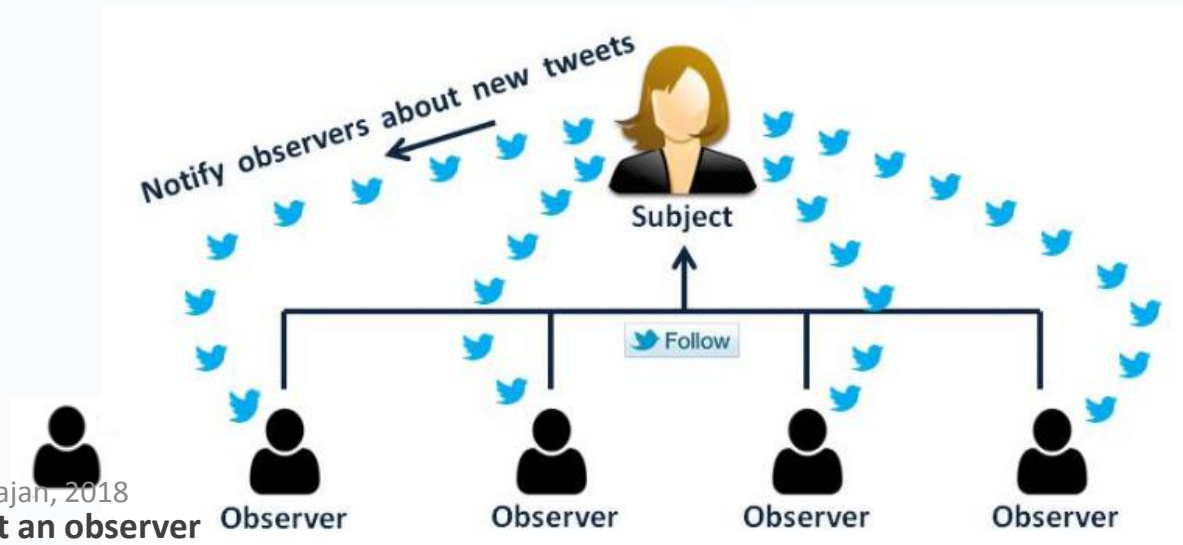


3. We can have a common interface to update the display elements, they all have the update() method that take in the same parameters temp, humidity and pressure

Introduce the Observer Pattern

A few real world examples:

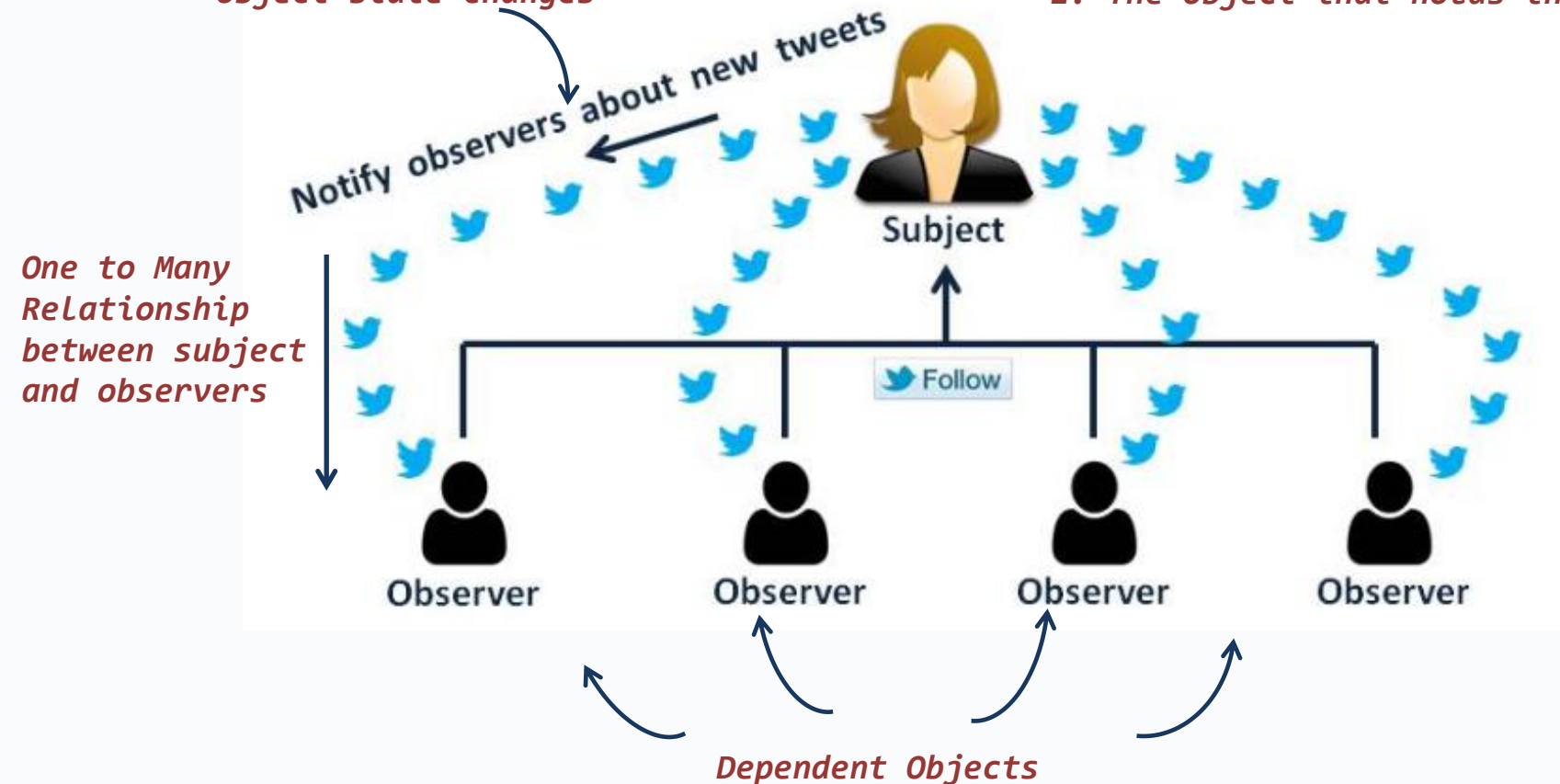
- A magazine subscription
 - A publisher produces magazines (e.g. Better Homes & Gardens)
 - A user subscribes to the magazine and is delivered copies each month
 - The user can unsubscribe when they want
 - Similarly, multiple users subscribe and unsubscribe
- Facebook messages
- Twitter posts
- Publisher + Subscriber => Observer Pattern
 - Publisher often known as the Subject
 - Subscribers referred to as the Observer



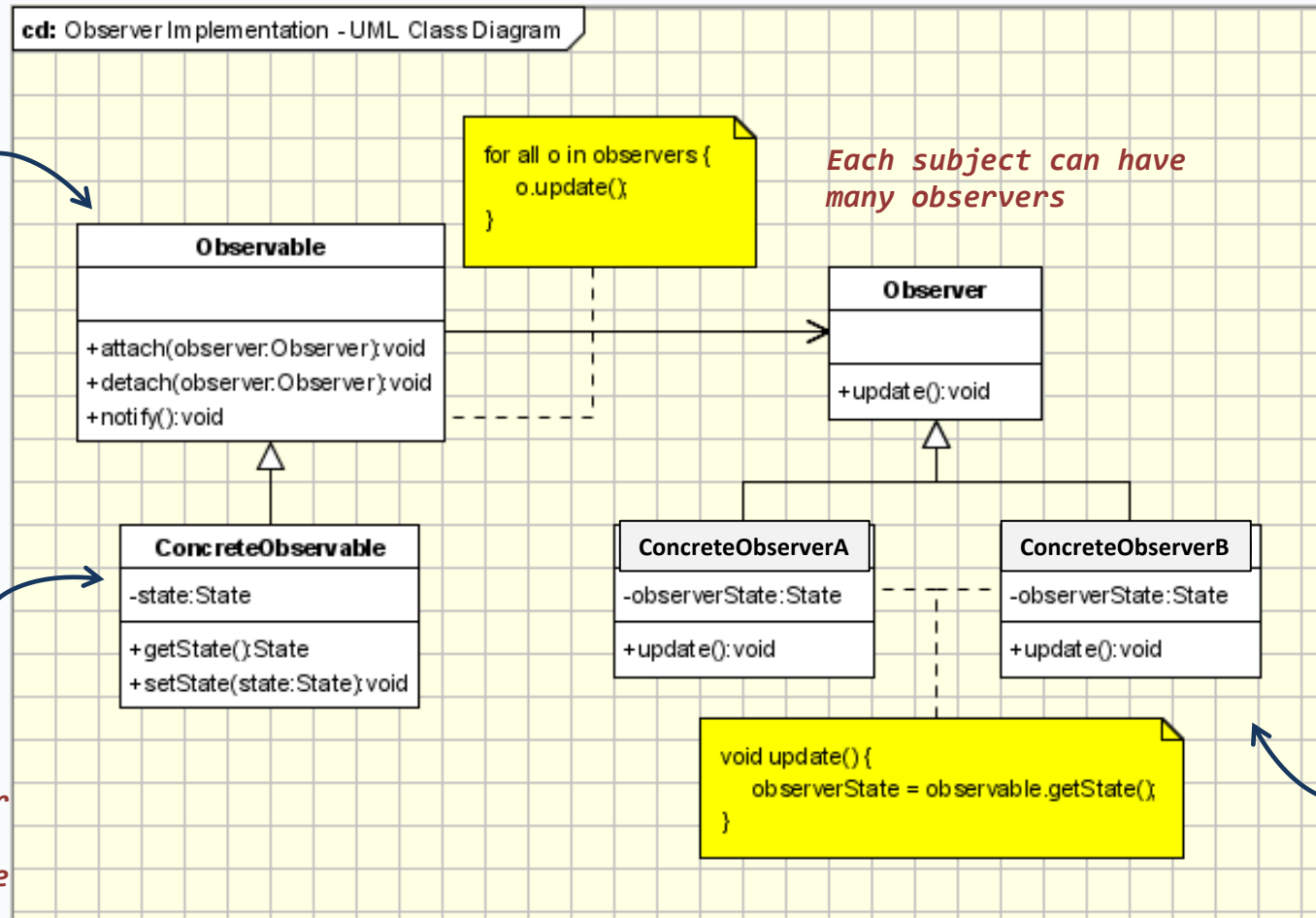
Introduce the Observer Pattern

2. Automatic update
to observers when
object state changes

1. The object that holds the state



Observer Pattern Implementation



Subject interface - objects use this interface to register as observers and also to remove themselves as observers

A concrete subject implements `attach()`, `detach()` to register and unregister observers and `notifyObserver()` that is used to update all current observers

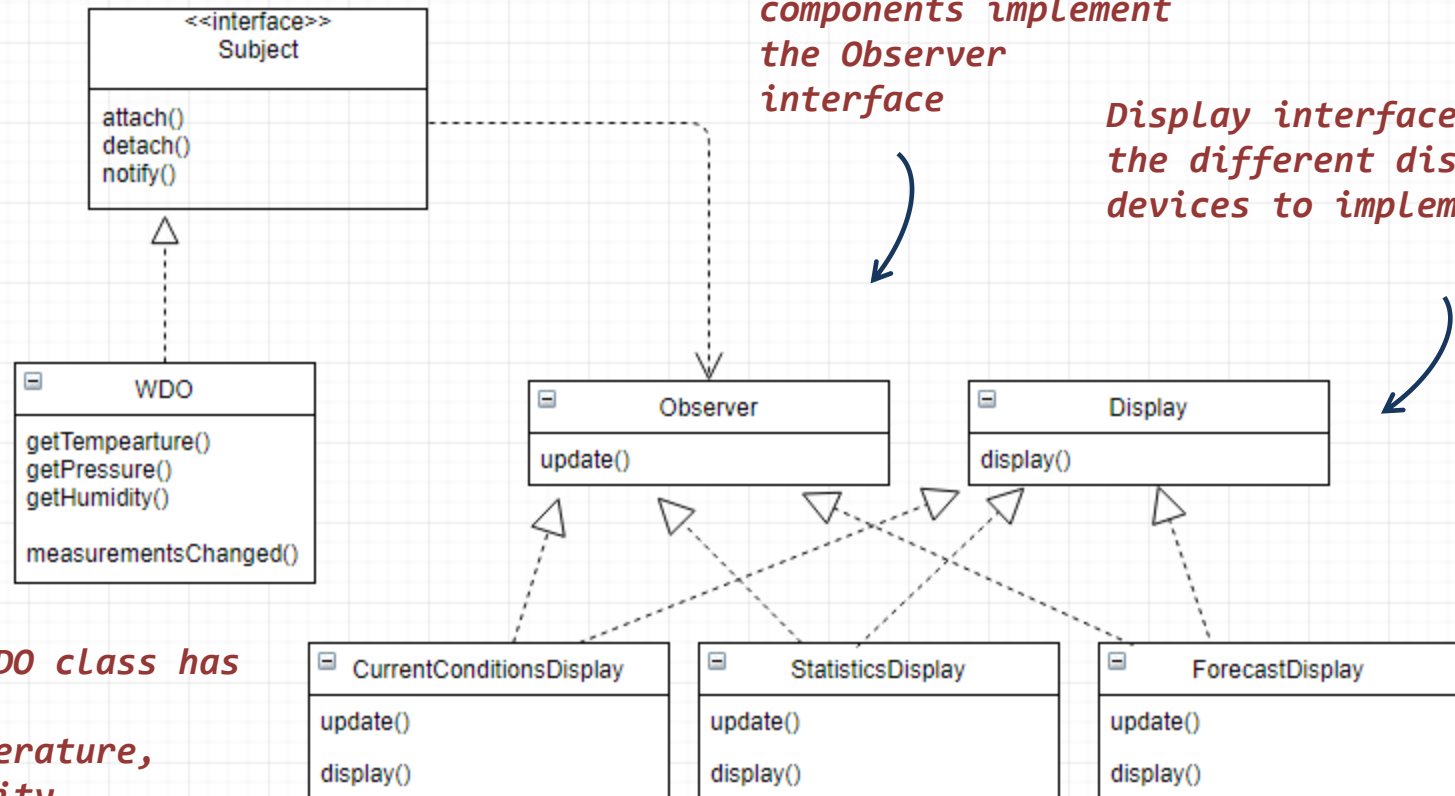
Each subject can have many observers

A concrete observer that implements the **Observer** interface and registers with the concrete subject to receive updates

Back to our Weather Station App

All Observer components implement the Observer interface

Display interface for the different display devices to implement

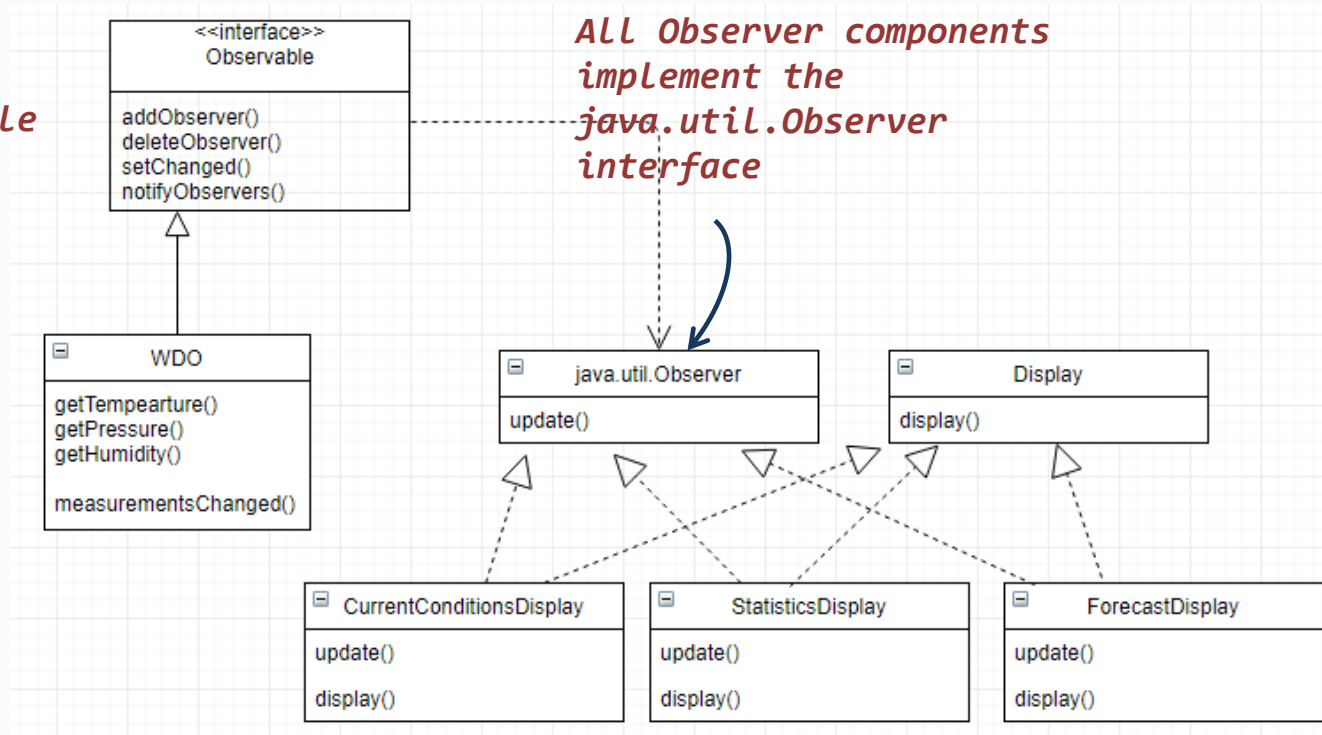


The WDO class has state (temperature, humidity, pressure) which change

The various display elements are our concrete observers

Using Java's built-in Observer Pattern

The *java.util.Observable* interface



- WDO extends `java.util.Observable` and inherits `addObserver()`, `deleteObserver()` and `notifyObservers()` methods
- For the `Observable` to send notifications
 - Call the `setChanged()` method to signal that a change has been made
 - Then call the `notifyObservers()` method

- For an `Observer` to receive notification, implement the `update()` method

Observer Pattern Benefits/Drawbacks

- The observer pattern provides a design where subjects and observers are **loosely coupled**
 - The only thing the subject knows about the observer is the interface that it implements
 - New observers can be added any time, as the subject only depends on a list of observers that implement the interface
 - Changes to either the subject or the observer do not affect each other
- Loosely coupled designs minimize interdependency between objects and allows us to build flexible OO systems

Design Principle #7: Strive for loosely coupled designs between objects that interact

- Note, if you extend `java.util.Observable`, then you cannot sub-class any other class

Design Toolbox

OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

OO Patterns

- *Strategy*
- *State*
- *Template Method*
- *Iterator*
- *Composite*
- *Decorator*
- *Observer*

OO Design Principles

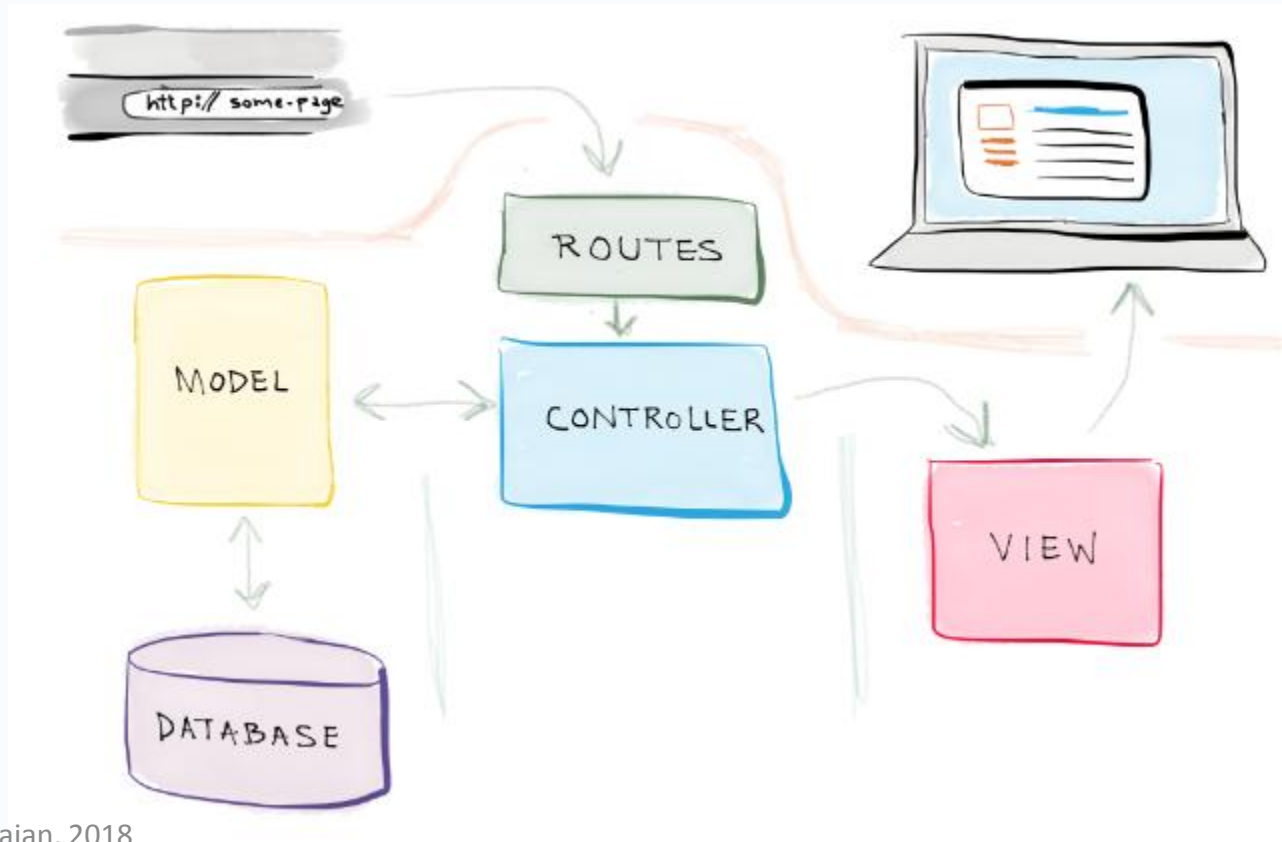
- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*
- *Classes should be open for extension and closed for modification*
- *Don't call us, we'll call you*
- *A class should have only one reason to change*
- *Strive for loosely coupled designs between objects that interact*
- *Depend on abstractions, do not depend on abstract classes*

Model View Controller

Separation of Concerns Using MVC (1)

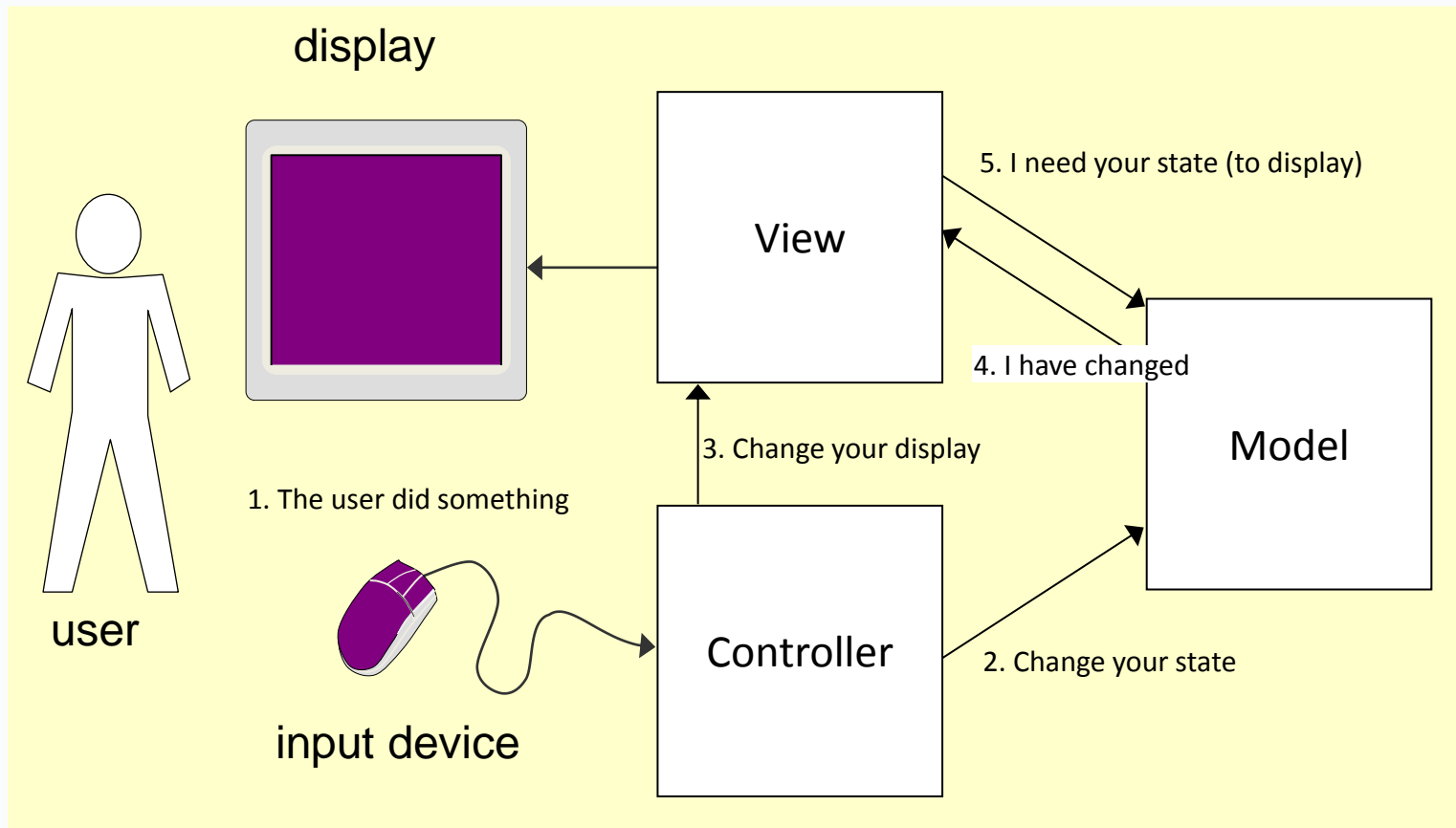
(Model-View-Controller)

A software architectural pattern that decouples data access, application logic and user interface into three distinct components



Separation of Concerns Using MVC (2)

(Model notifies View)

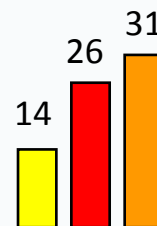


View

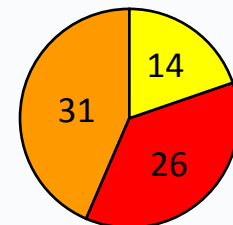
- This is the **presentation layer** provides the interaction that the user sees (e.g a web page).
- View component takes inputs from the user and sends actions to the **controller** for manipulating data.
- View is responsible for displaying the results obtained by the controller from the model component in a way that user wants them to see or a pre-determined format.
- The format in which the data can be visible to users can be of any 'type' such as HTML or XML depending upon the presentation tier.
- It is responsibility of the controller to choose a view to display data to the user.

Model: array of numbers [14, 26, 31]

➔ Different Views for the same Model:



versus



Model

- Holds all the data, state
- Responds to instructions to change of state (from the controller)
- Responds to requests for information about its state (usually from the view),
- Sends notifications of state changes to “observer” (view)

Controller

- Glue between user and processing (Model) and formatting (View) logic
- Accepts the user request or inputs to the application, parses them and decides which type of Model or View should be invoked