Design Patterns

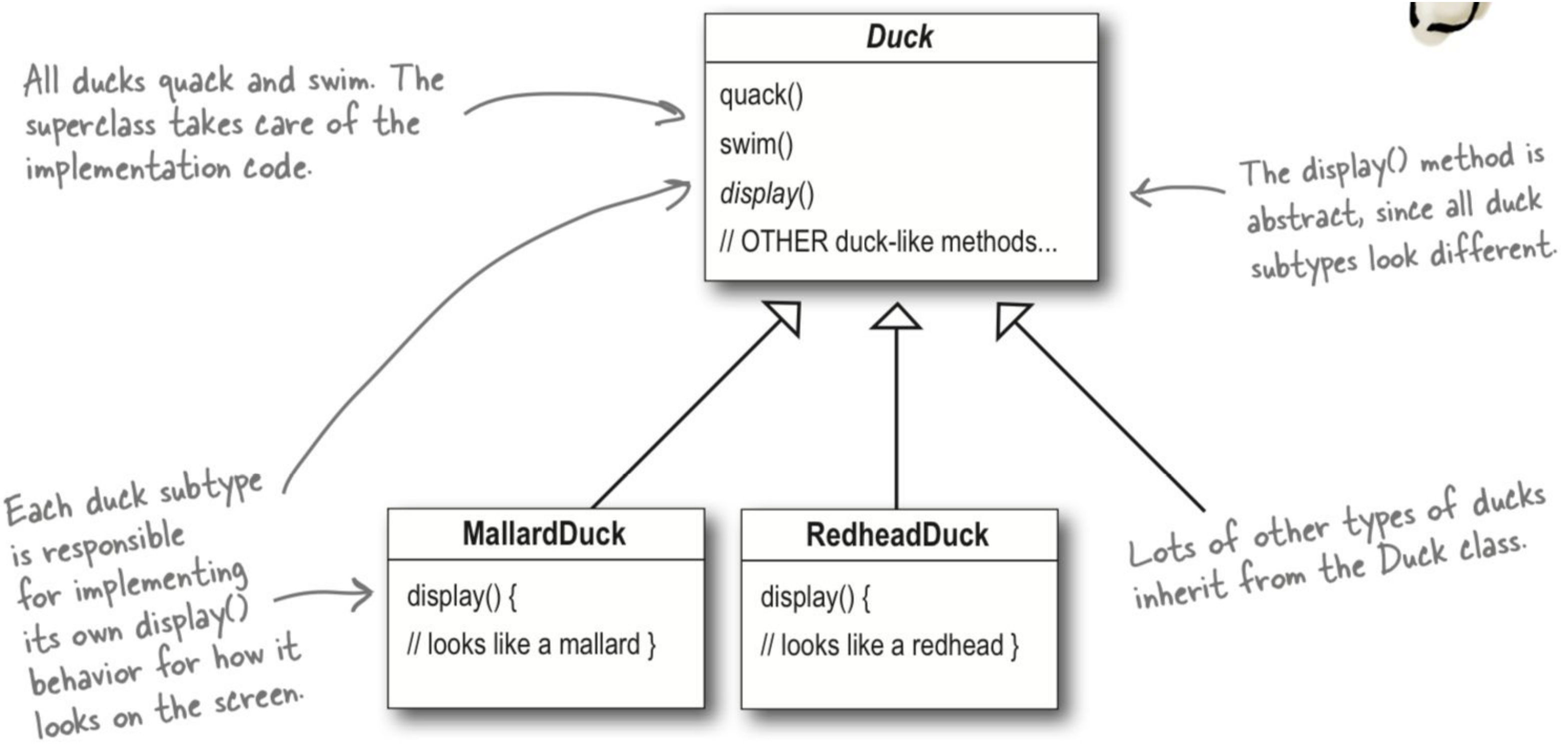
Design Patterns

Someone has already solved your problems !!

Duck Simulator Example (source: Head First Design Patterns, 2nd ed.)

- This simulator is for showing large variety of ducks quacking, flying etc.
- This is not for one type of duck. Many different variety of ducks
  - eg. Mallard duck, Redhead duck

Class Diagram: Design 1:



Duck

quack()

swim()

display()

// OTHER duck-like methods...

MallardDuck

display() {

// looks like a mallard }

RedheadDuck

display() {

// looks like a redhead }

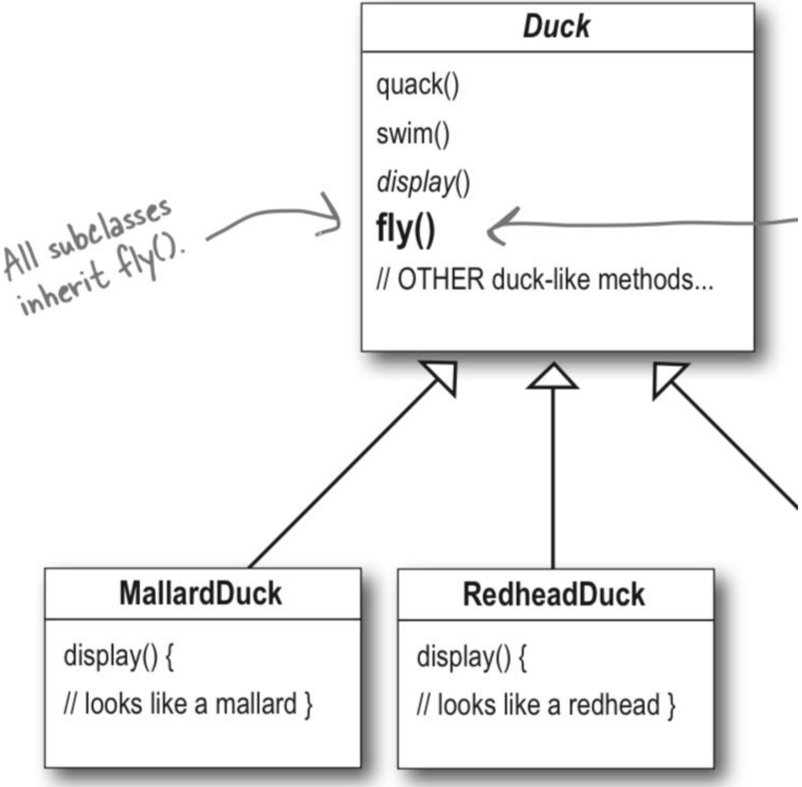
All ducks quack and swim. The superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

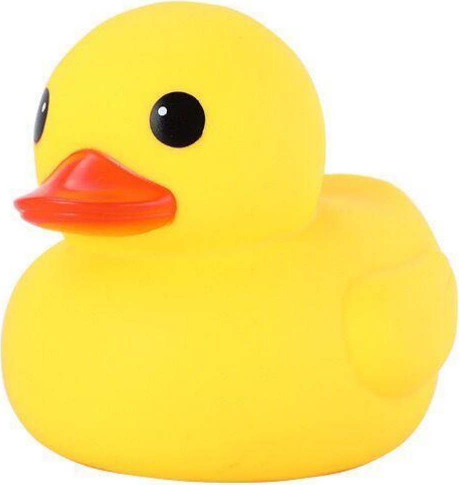
Lots of other types of ducks inherit from the Duck class.

Design 1: Extensibility


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }

Design 1: Flaw

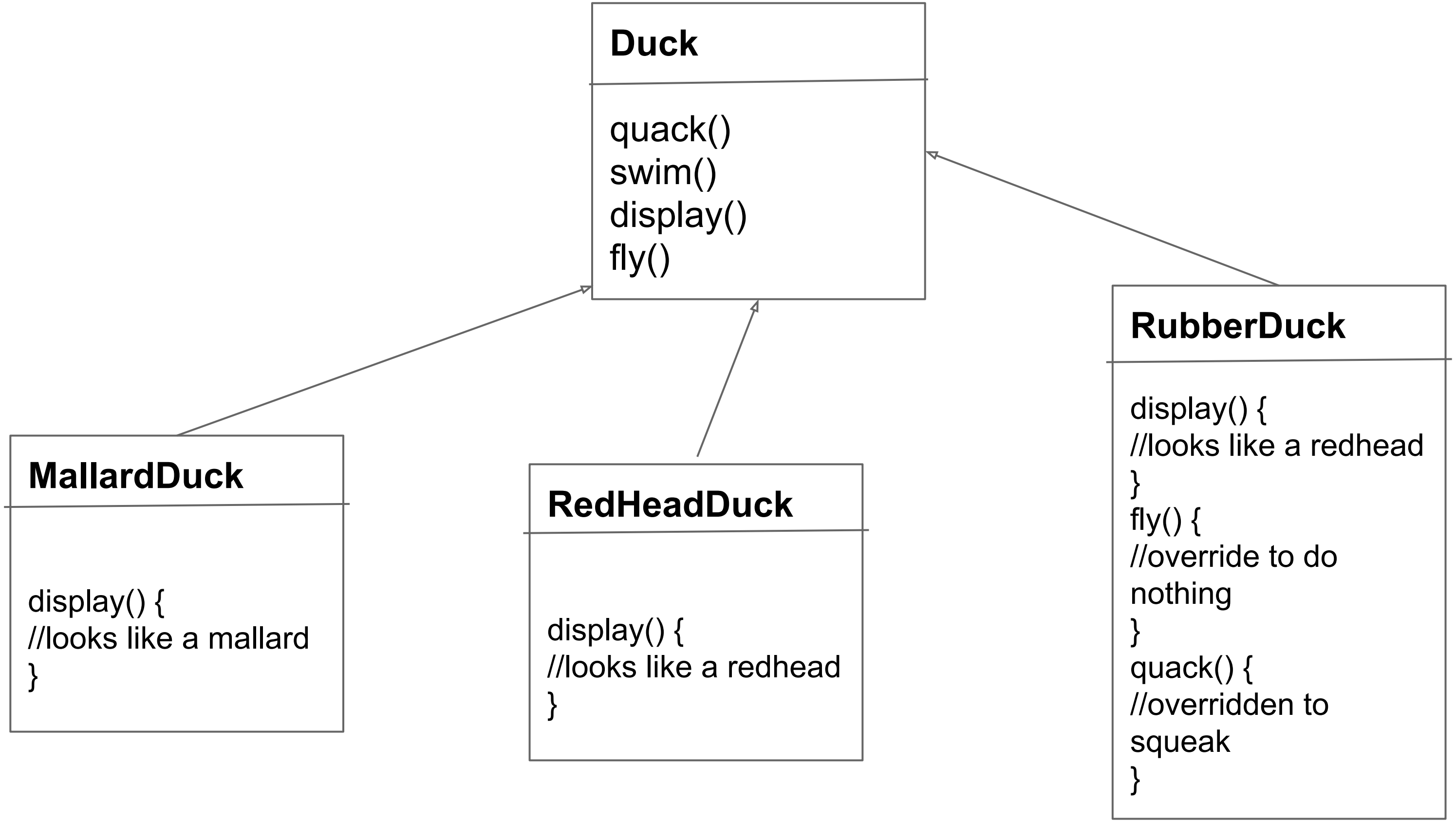
Can't fly



Rubber duck

Can Quack

Design 2: Solution to Design 1 Flaw



Duck

```
quack()
swim()
display()
fly()
```

MallardDuck

```
display() {
//looks like a mallard
}
```

RedHeadDuck

```
display() {
//looks like a redhead
}
```

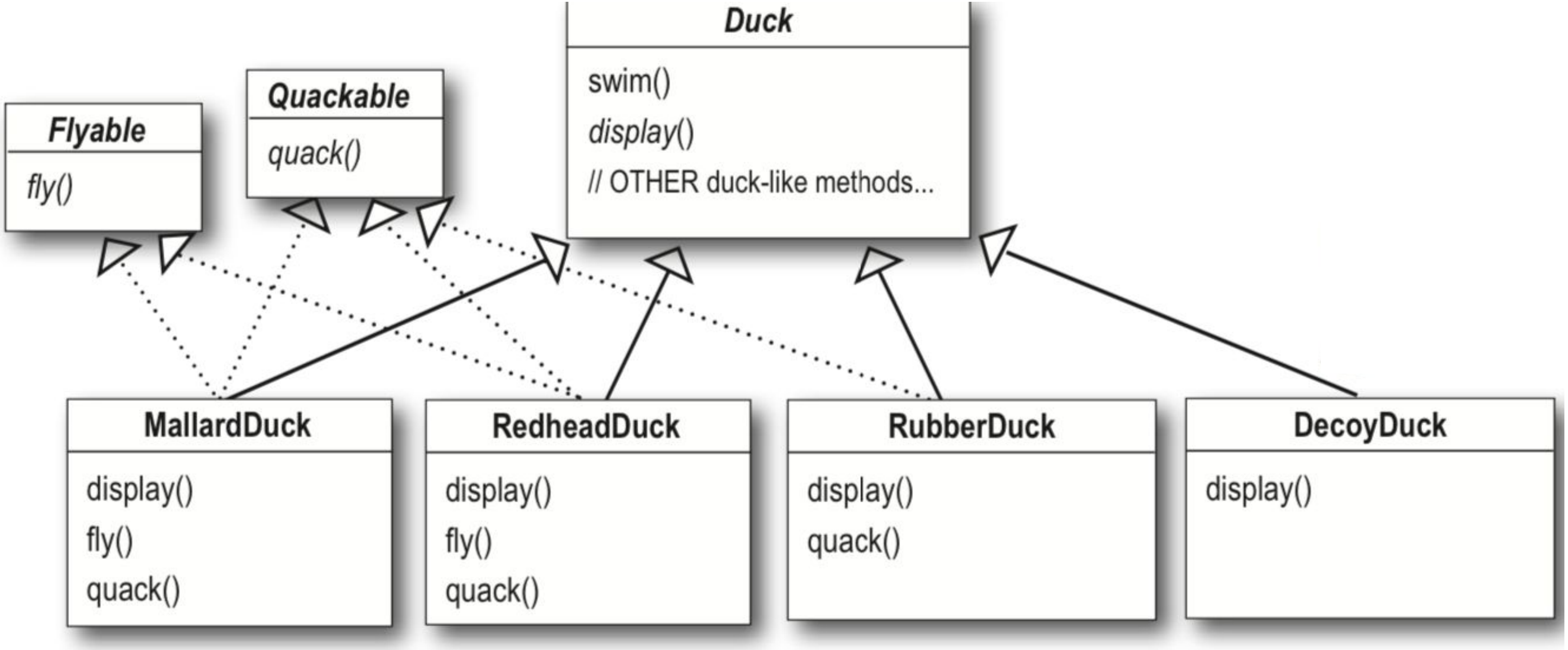
RubberDuck

```
display() {
//looks like a redhead
}
fly() {
//override to do
nothing
}
quack() {
//overridden to
squeak
}
```

Design 2: Flaw

- Overriding solution is not a cleaner way
- There will be issues with maintenance when there are modifications in the future.
- Whenever a new Duck subclass is added all the required overriding have to be done.

Design 3: Interfaces of Changing Behaviours



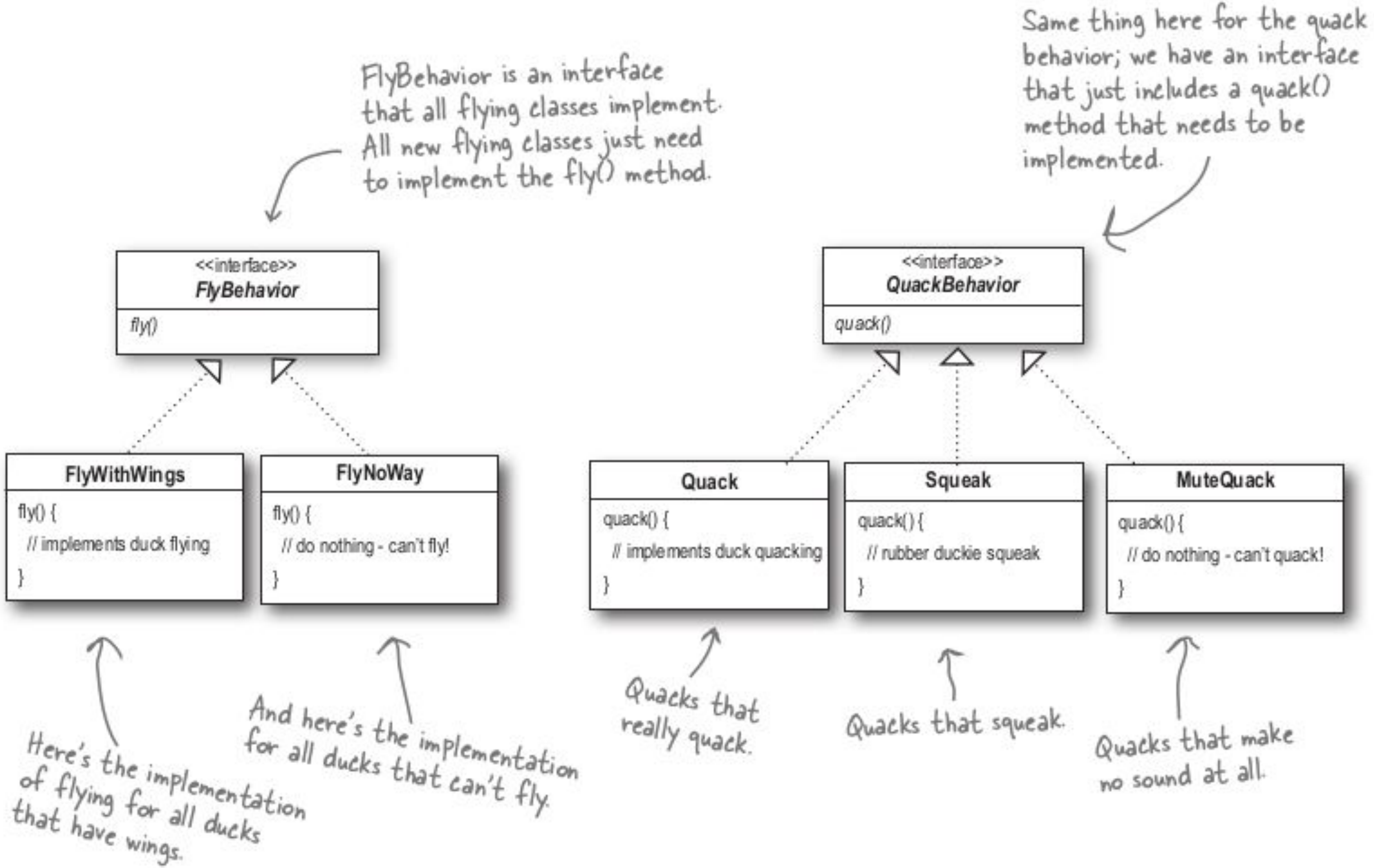
Design 3: Flaw

- No code reuse
- The fly method has to be implemented in all the subclasses even for the Ducks with the same fly method.
- For most of the modifications, we will have to make the modifications in multiple subclasses

The Final Solution

- The Duck subclasses will use interface for the behaviors (FlyBehavior, QuackBehavior)
- The concrete behavior is coded in the class that implements the above interfaces
- The actual behavior is not locked in the Duck subclasses

The Final Solution



FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly() method.

<<interface>>
FlyBehavior

fly()

FlyWithWings

```
fly() {
  // implements duck flying
}
```

FlyNoWay

```
fly() {
  // do nothing - can't fly!
}
```

Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.

<<interface>>
QuackBehavior

quack()

Quack

```
quack() {
  // implements duck quacking
}
```

Squeak

```
quack() {
  // rubber duckie squeak
}
```

MuteQuack

```
quack() {
  // do nothing - can't quack!
}
```

Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.

Final Solution: Duck class

```
public abstract class Duck {
  FlyBehavior flyBehavior;
  QuackBehavior quackBehavior;

public Duck() { }
public abstract void display();

public void performFly() {
    flyBehavior.fly();
}

public void performQuack() {
    quackBehavior.quack();
}
}

public void swim() {
    System.out.println("All ducks
float, even decoys!");
}
}
```

Final Solution: Mallard Duck

```java
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

Main Code in Simulator

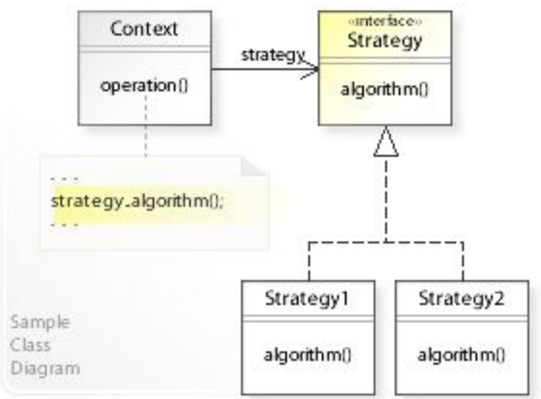
```
public class MiniDuckSimulator {

public static void main(String[] args) {

   Duck mallard = new MallardDuck();

   mallard.performQuack();

   mallard.performFly();

}
```

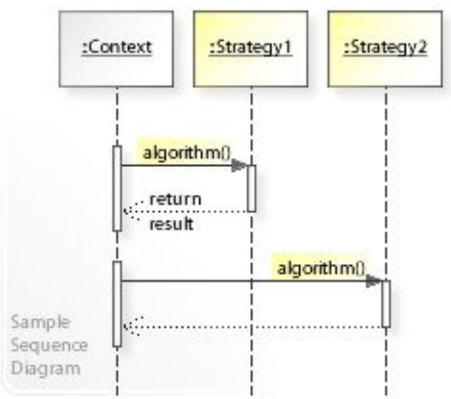
https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/strategy

What we were discussing till now?

Strategy Design Pattern

Is a Behavioral design pattern in which enables selecting the algorithm at runtime.





Source: https://en.wikipedia.org/wiki/Strategy_pattern

Strategy Design Pattern

The behaviors of a class should not be inherited. Instead, they should be encapsulated using interfaces.

Types of Design Patterns

- **Creational**

  Provides object or class creation mechanism

- **Structural**

  assemble object and classes into a structure ensuring that the structure should be flexible and efficient

- **Behavioural**

  Manages how one class communicates with another

Creational Design Pattern

Factory Method Pattern

Separates out the creation of objects/instances.

Making Pizza Example

```
Pizza orderPizza() {
  Pizza pizza = new Pizza();

  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza
}
```

```
Pizza orderPizza(String type) {
  Pizza pizza;

  if (type.equals("cheese")) {
    pizza = new CheesePizza();
   } else if (type.equals("greek") {
    pizza = new GreekPizza();
   } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
   }

  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza
}
```

Instantiation of different classes for different types of pizzas

There is pressure to add more pizza types !!

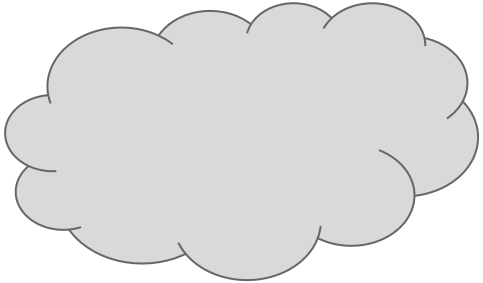
```
if (type.equals("cheese")) {
   pizza = new CheesePizza();
} else if (type.equals("greek") {
   pizza = new GreekPizza();
} else if (type.equals("pepperoni") {
   pizza = new PepperoniPizza();
} else if (type.equals("clam") {
   pizza = new ClamPizza();
} else if (type.equals("veggie") {
  pizza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

This keeps on growing and some get modified

Encapsulating Object Creation

```
Pizza orderPizza(String type) {
  Pizza pizza;

  if (type.equals("cheese")) {
    pizza = new CheesePizza();
   } else if (type.equals("greek") {
    pizza = new GreekPizza();
   } else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
   }

  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza
}
```

```
Pizza orderPizza(String type) {
  Pizza pizza;
```

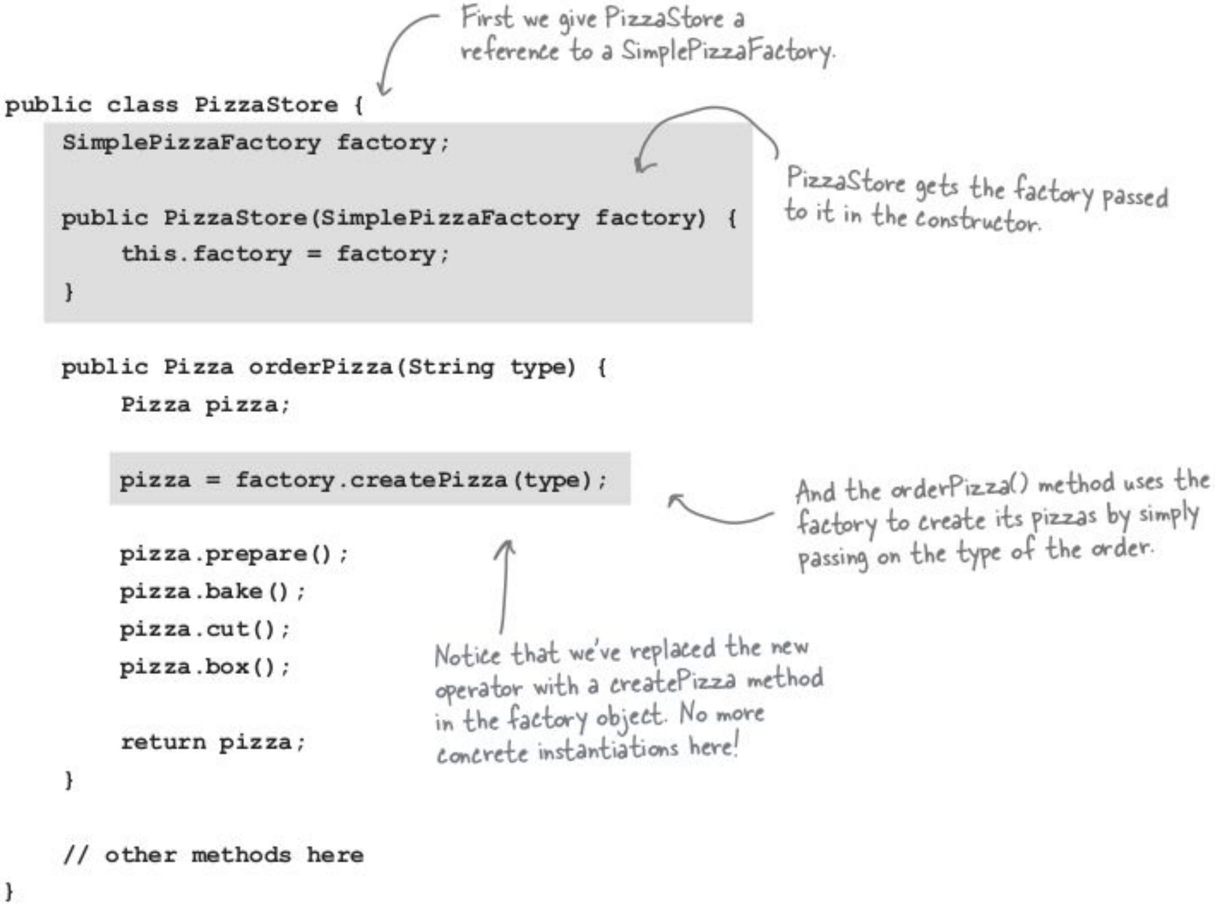


```
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza
}
```

Simple 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;
}
```

How does PizzaStore looks like?



First we give PizzaStore a reference to a SimplePizzaFactory.

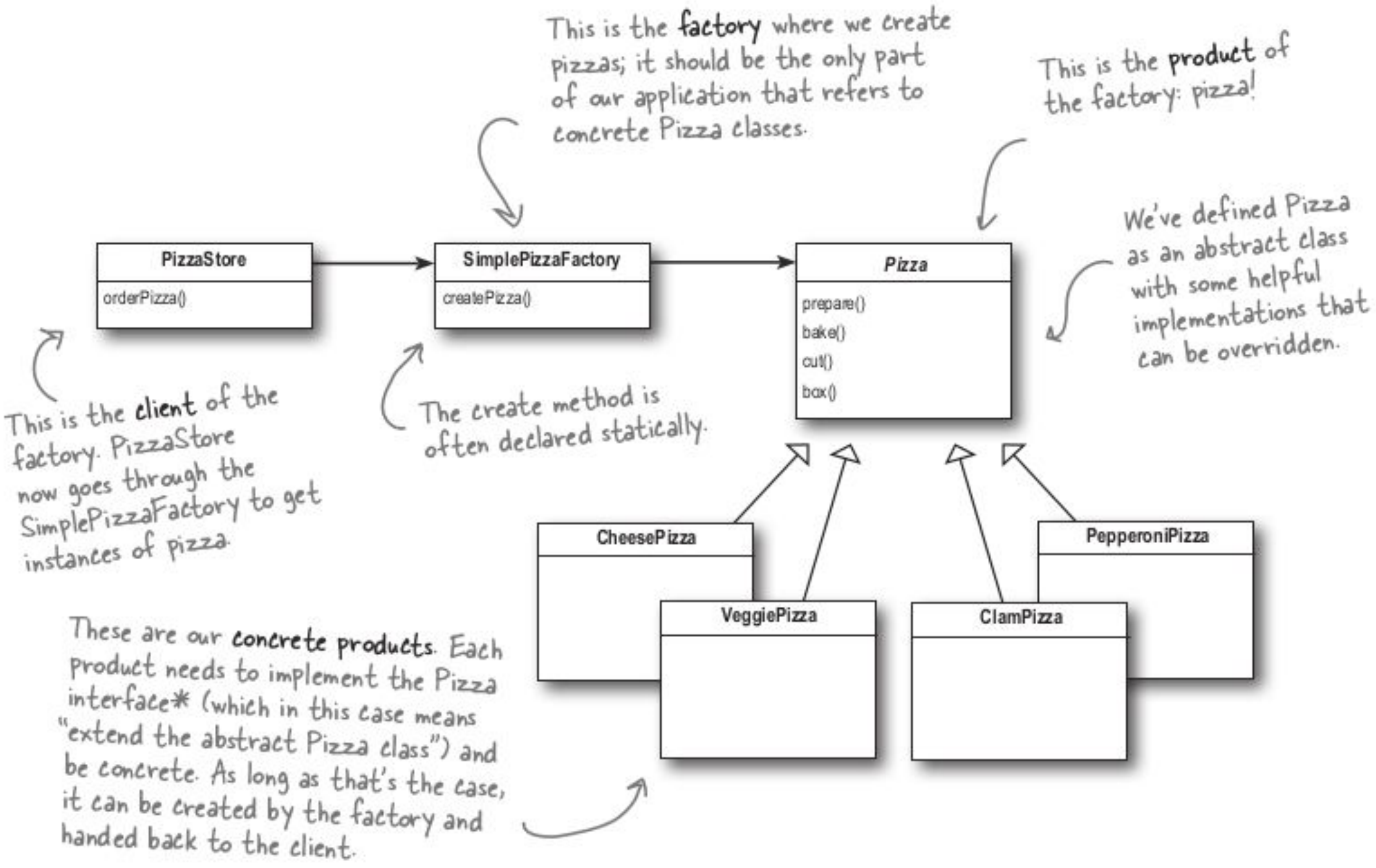
```
public class PizzaStore {
    SimplePizzaFactory factory;

    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;
    }

    // other methods here
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the new operator with a createPizza method in the factory object. No more concrete instantiations here!

Class Diagram of Pizza Store



This is the **factory** where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes.

This is the **product** of the factory: pizza!

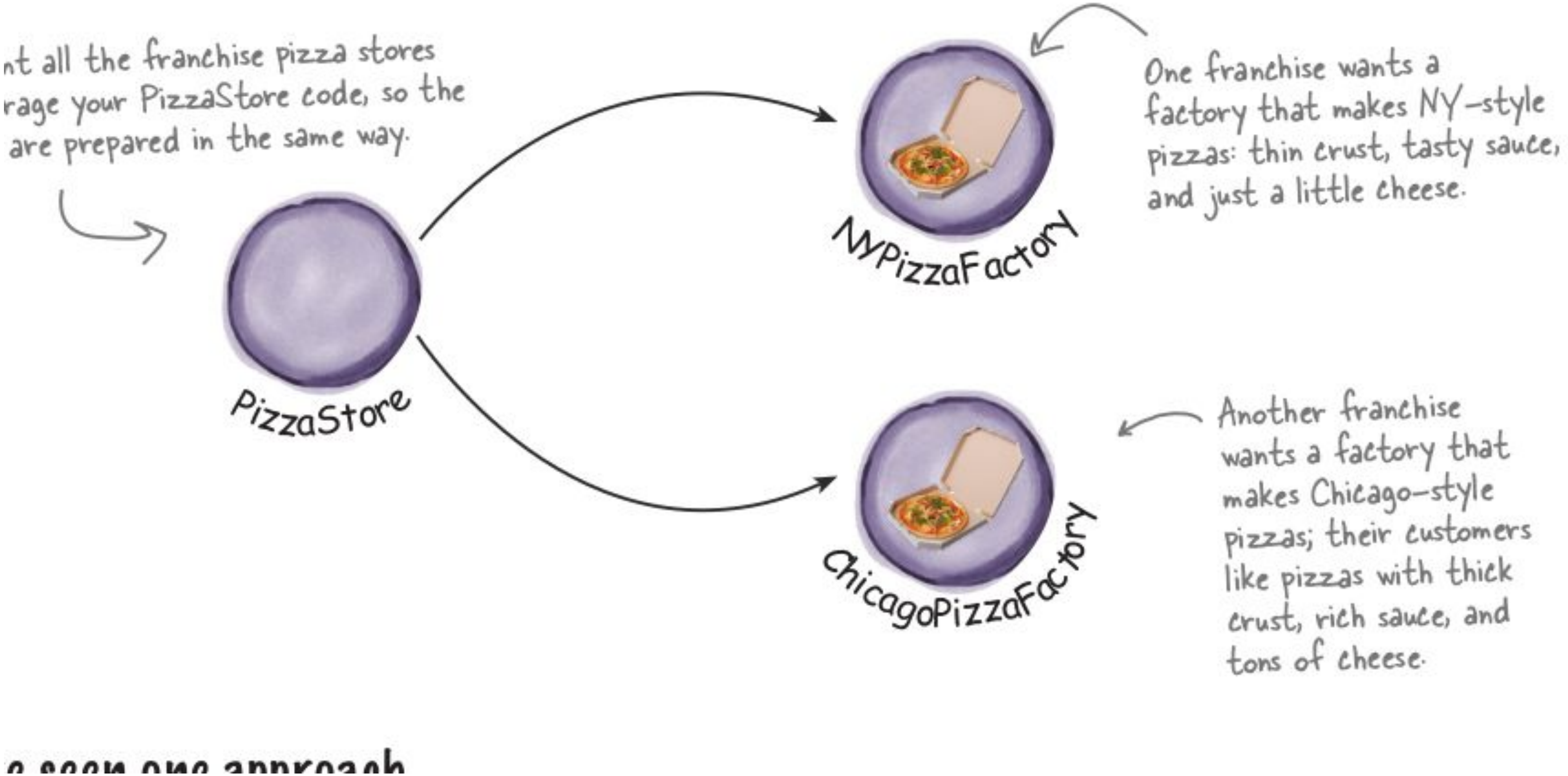
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

This is the **client** of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

These are our **concrete products**. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Different Style Pizza



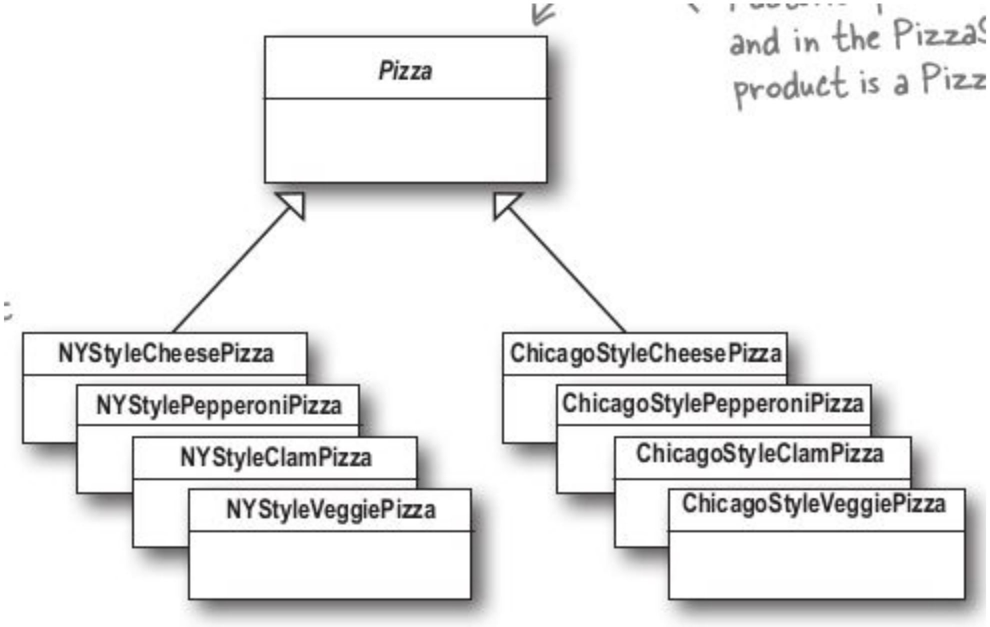
nt all the franchise pizza stores
rage your PizzaStore code, so the
are prepared in the same way.

One franchise wants a factory that makes NY-style pizzas: thin crust, tasty sauce, and just a little cheese.

Another franchise wants a factory that makes Chicago-style pizzas; their customers like pizzas with thick crust, rich sauce, and tons of cheese.

e seen one approach

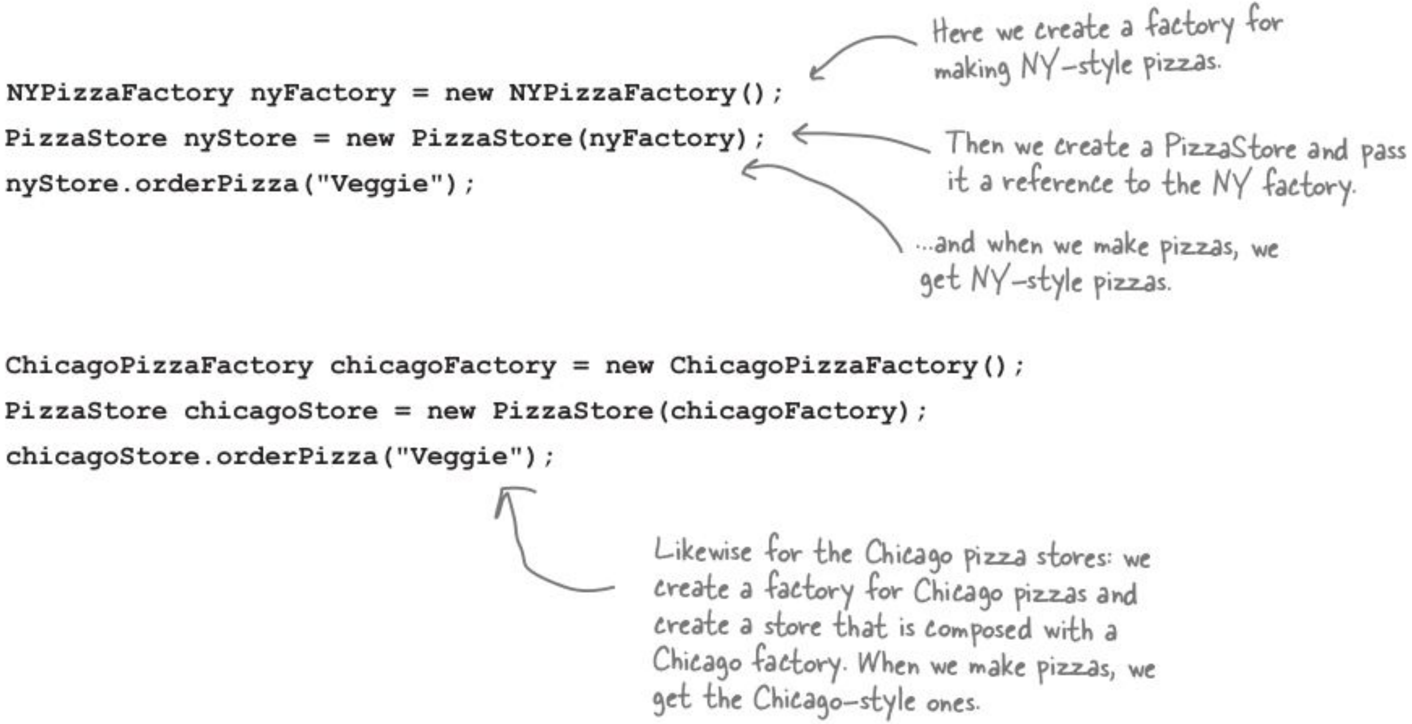
Pizza Class Diagram



New York Style

Cheese

Factory Classes for Different Styles



```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("Veggie");
```

Here we create a factory for making NY-style pizzas.

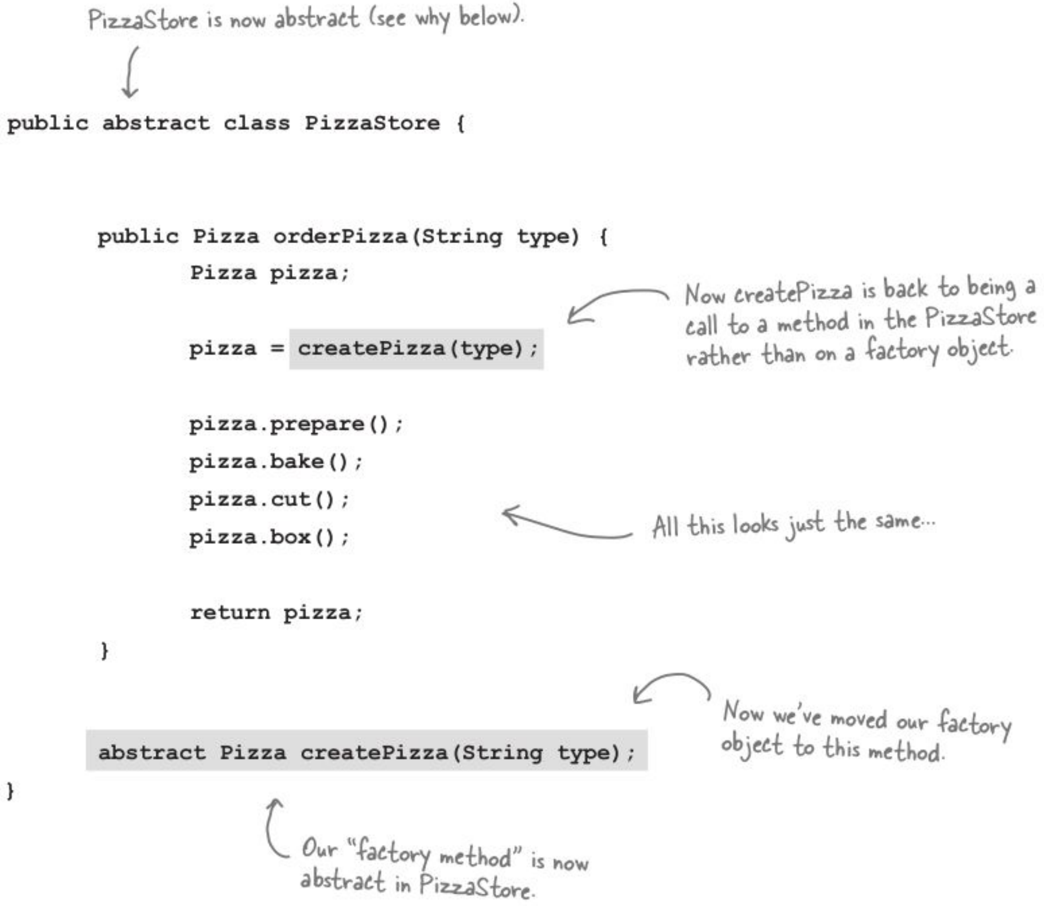
Then we create a PizzaStore and pass it a reference to the NY factory.

...and when we make pizzas, we get NY-style pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.orderPizza("Veggie");
```

Likewise for the Chicago pizza stores: we create a factory for Chicago pizzas and create a store that is composed with a Chicago factory. When we make pizzas, we get the Chicago-style ones.

A Different Design to Manage Styles



```
public abstract class PizzaStore {


        public Pizza orderPizza(String type) {
                Pizza pizza;

                pizza = createPizza(type);

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

                return pizza;
        }


        abstract Pizza createPizza(String type);
}
```

PizzaStore is now abstract (see why below).

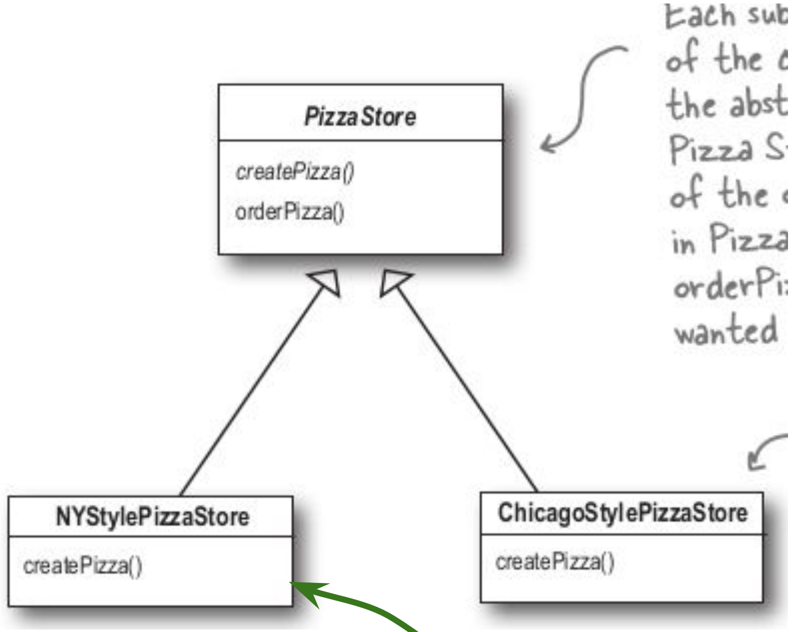
Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

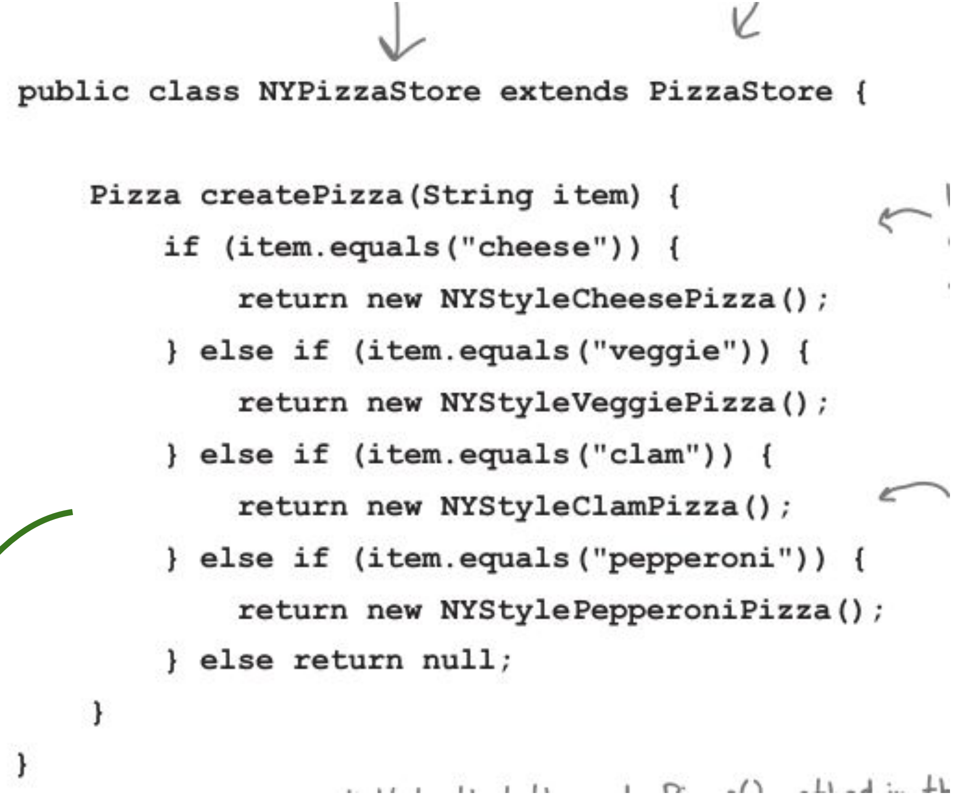
All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

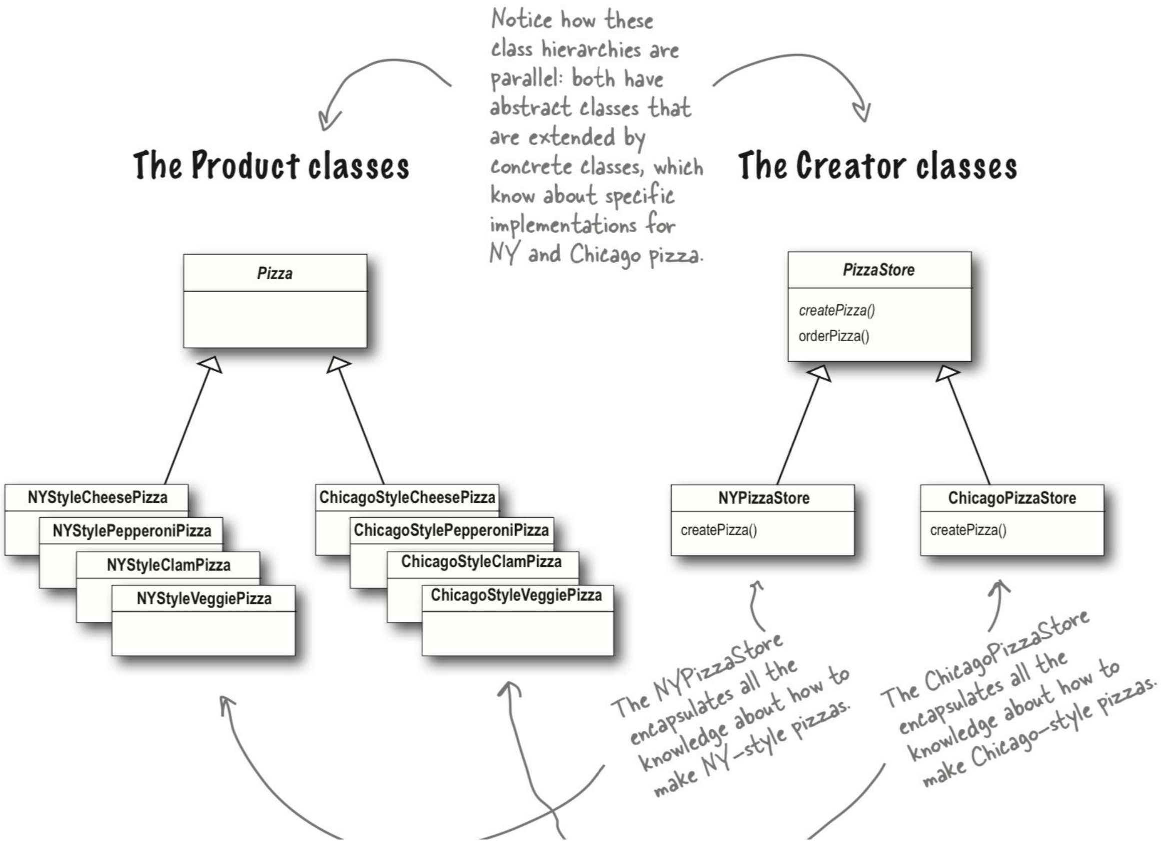
Allow Subclasses to Decide





```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

Product and Creator Classes



Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago pizza.

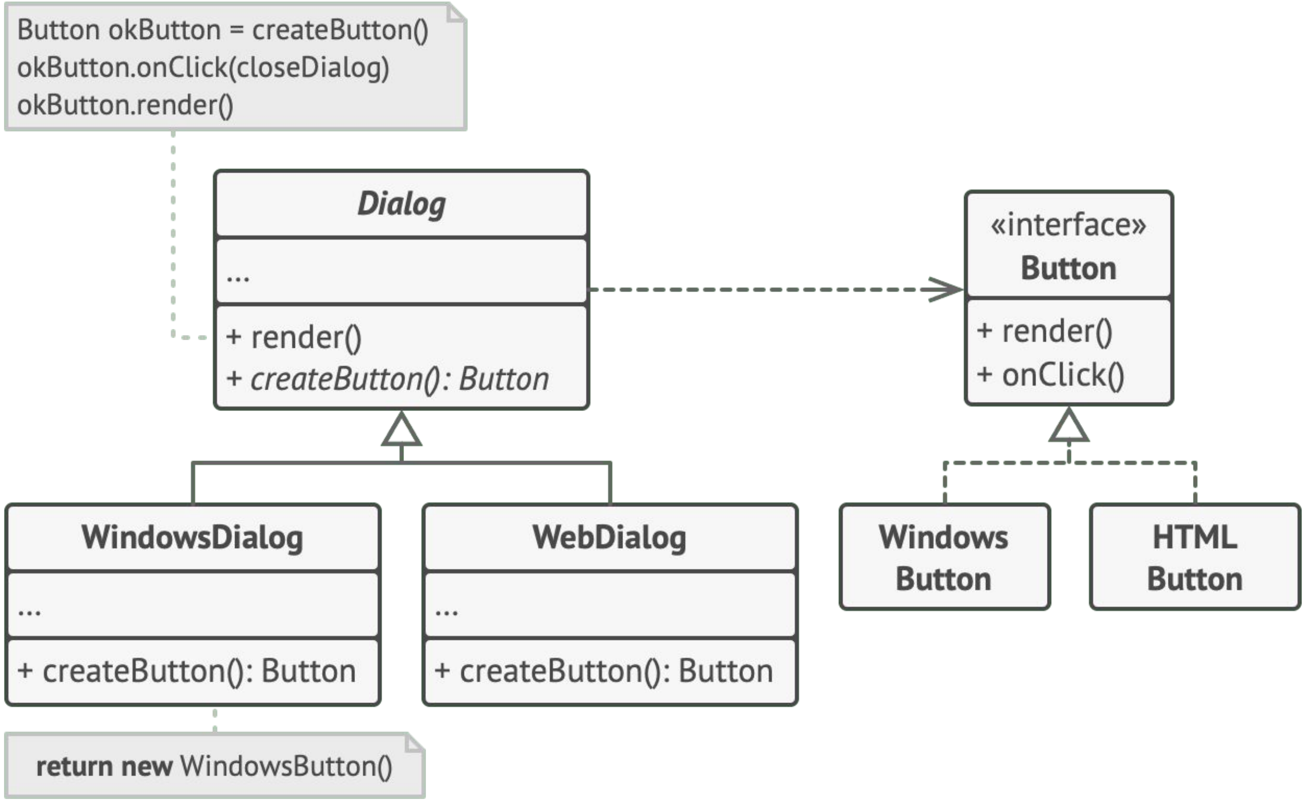
The Product classes

The Creator classes

The NYPizzaStore encapsulates all the knowledge about how to make NY-style pizzas.

The ChicagoPizzaStore encapsulates all the knowledge about how to make Chicago-style pizzas.

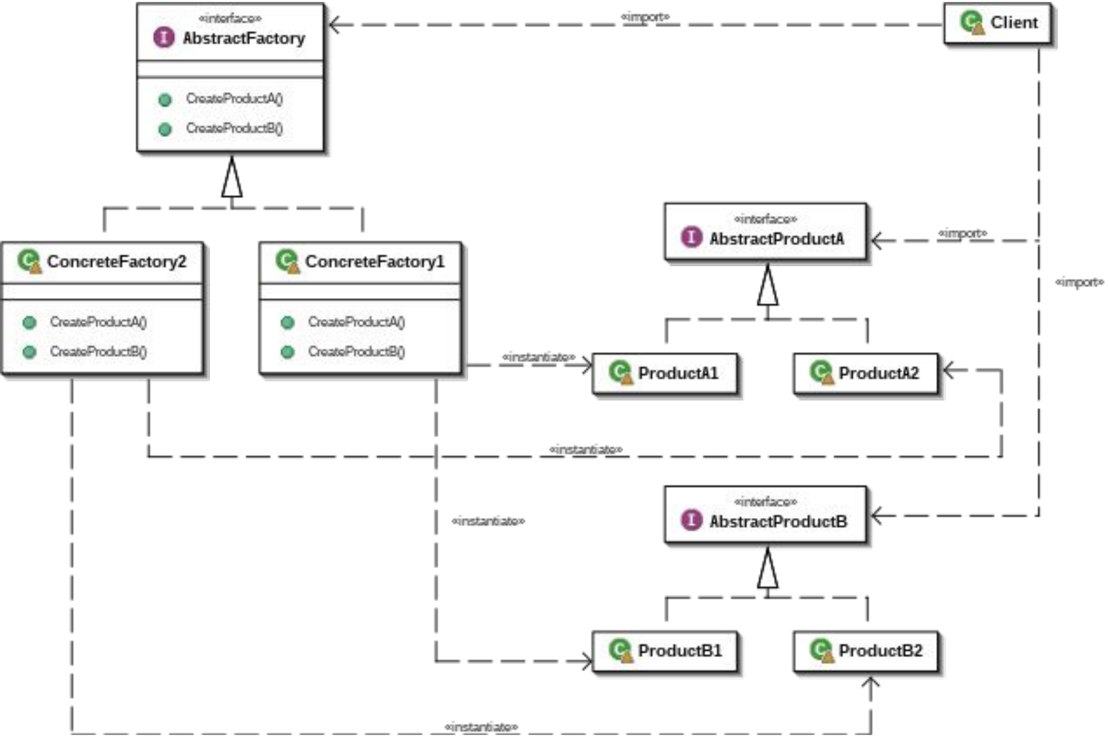
Another Example: Shapes

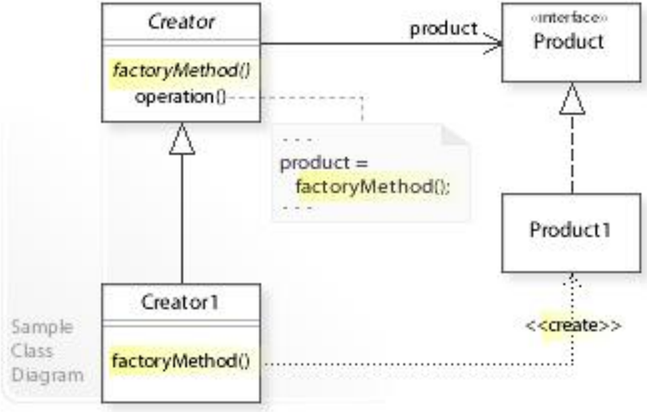


Source: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Abstract Factory Pattern

Factory Method pattern is responsible for creating products that belong to one family, while Abstract Factory pattern deals with multiple families of products.



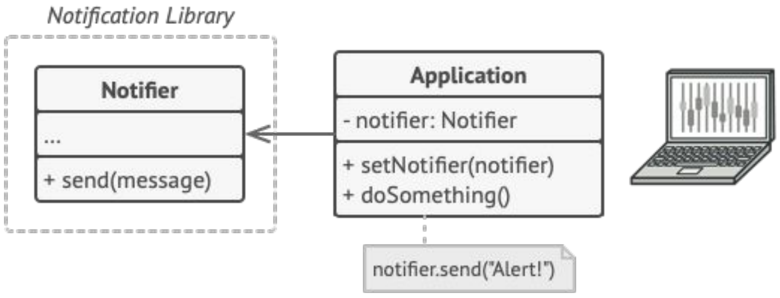


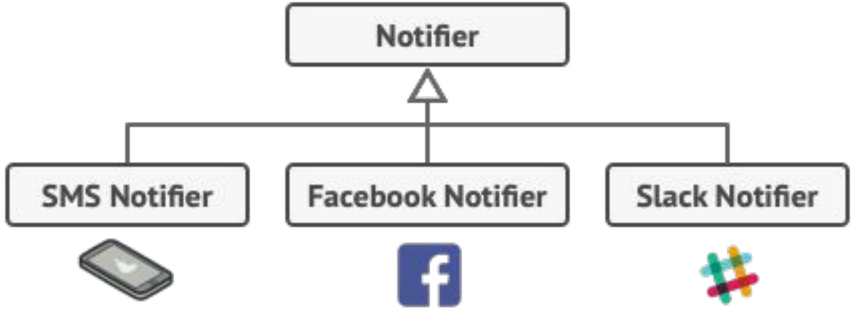
Structural Pattern

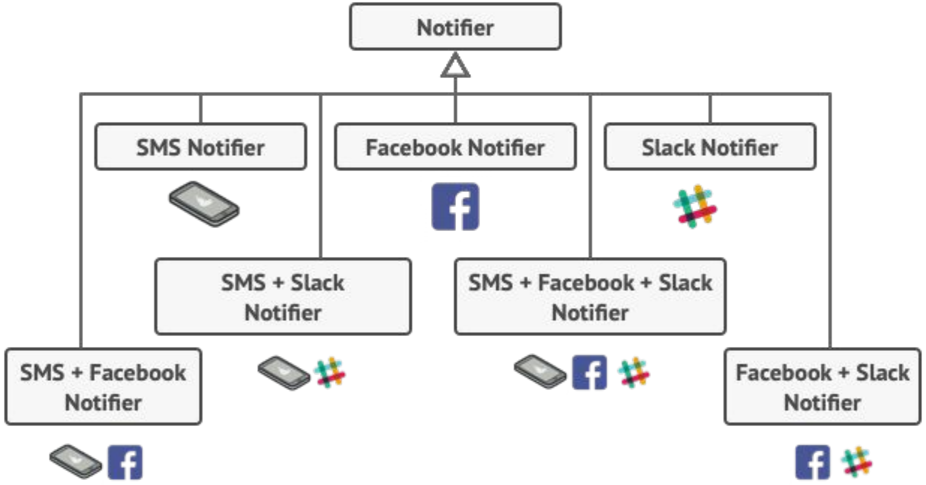
Decorator Pattern

- Allows addition of functionalities to a class without altering its structure.
- Also without inheriting the class.

Notifier Example

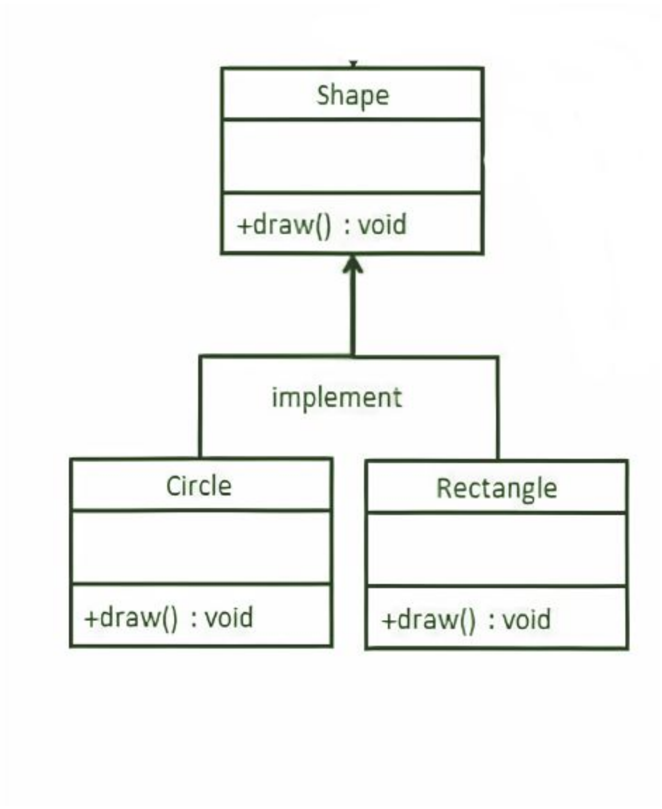






source: https://refactoring.guru/design-patterns/decorator

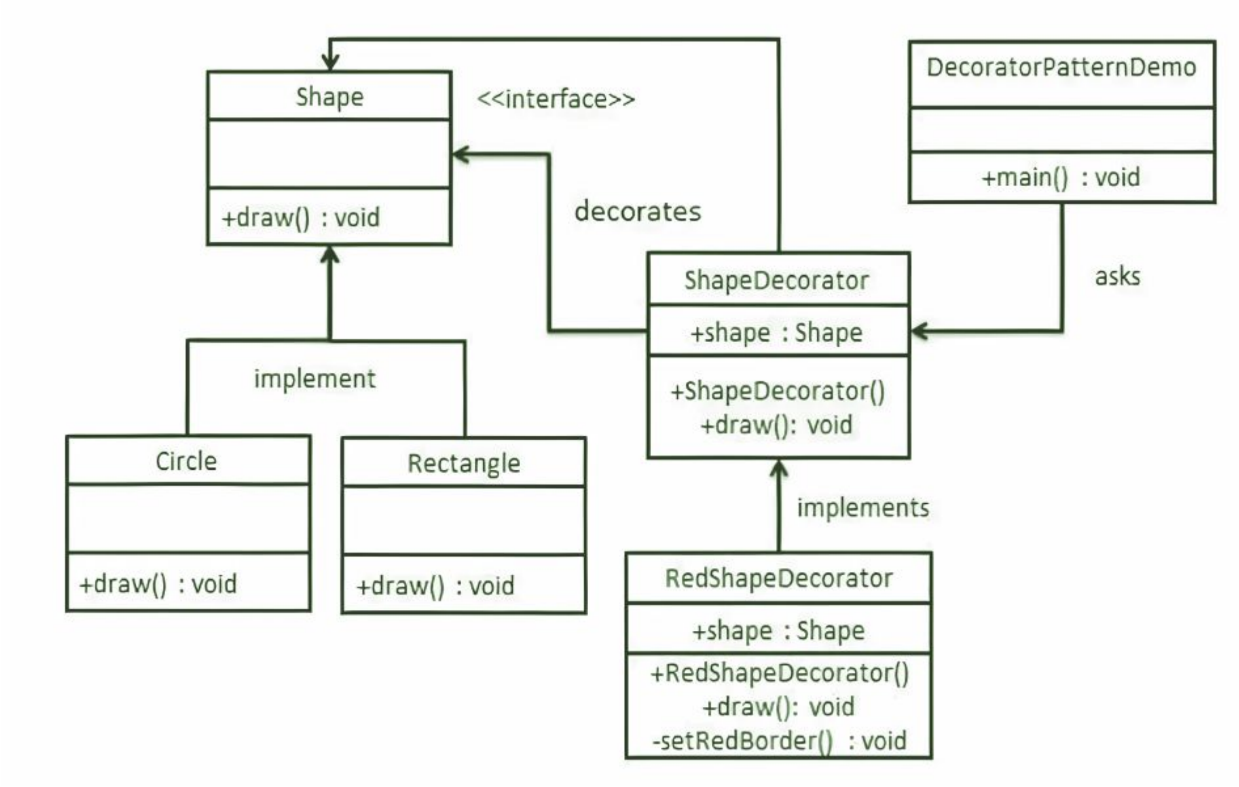
Shape Example



Rectangle with red border

Solution: Another subclass which can draw rectangle with red border?

Shapes: Solution with Decorator Pattern



source: https://www.geeksforgeeks.org/decorator-design-pattern-in-java-with-example/

Shapes: Decorator Pattern

```
// Interface named Shape
public interface Shape {

    // Method inside interface
    void draw();
}
```

```
public class Rectangle
implements Shape {

  @Override public void
draw()   {
    System.out.println("Sha
pe: Rectangle");
   }
}
```

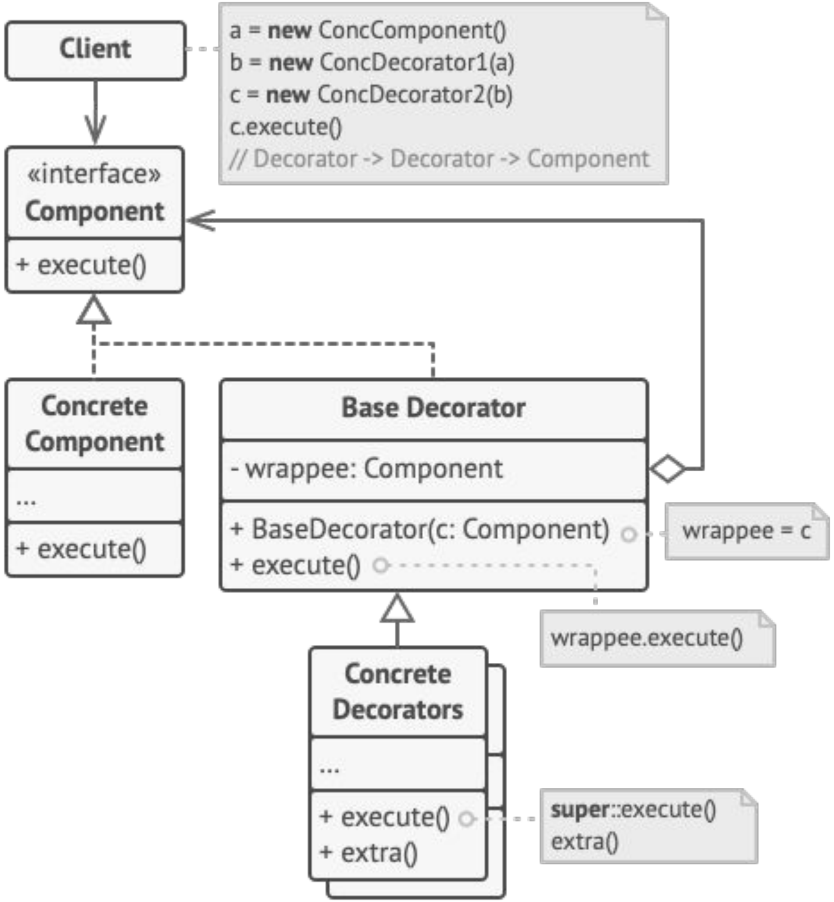
```
public abstract class ShapeDecorator implements Shape {
        protected Shape decoratedShape;
        public ShapeDecorator(Shape decoratedShape)
        {
                this.decoratedShape = decoratedShape;
        }

        public void draw() { decoratedShape.draw(); }
}
```

```
public class RedShapeDecorator extends ShapeDecorator
{
   public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape); }

   @Override public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);  }

   private void setRedBorder(Shape decoratedShape) {
      System.out.println("Border Color: Red");  }
}
```

Shapes Main Demo

```
public class DecoratorPatternDemo {
   public static void main(String[] args)
    {
      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      redRectangle.draw();
    }
}
```

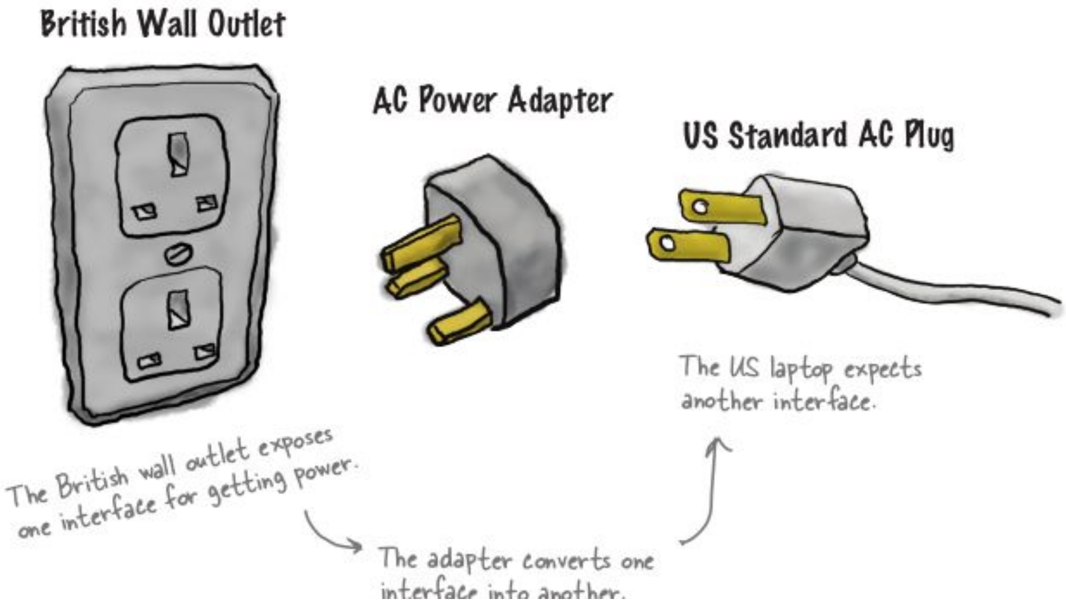
Decorator: Class Diagram



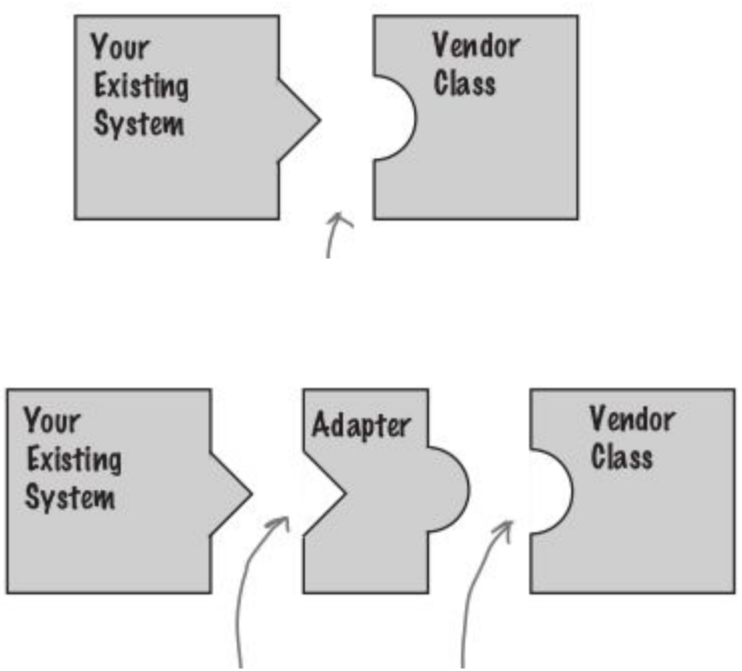
source: https://refactoring.guru/design-patterns/decorator

Structural Pattern: Adapter Pattern

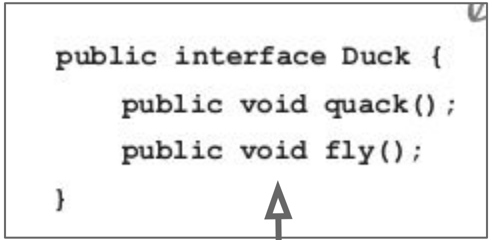
acts as a connector between two incompatible interfaces


British Wall Outlet
AC Power Adapter
US Standard AC Plug
The British wall outlet exposes one interface for getting power.
The adapter converts one interface into another.
The US laptop expects another interface.

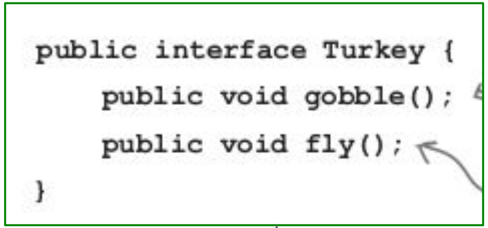
Adapter Pattern



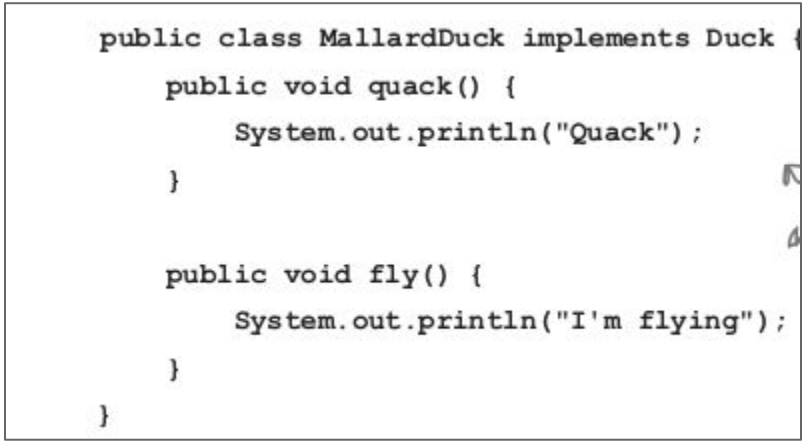
Duck Simulator for Turkey



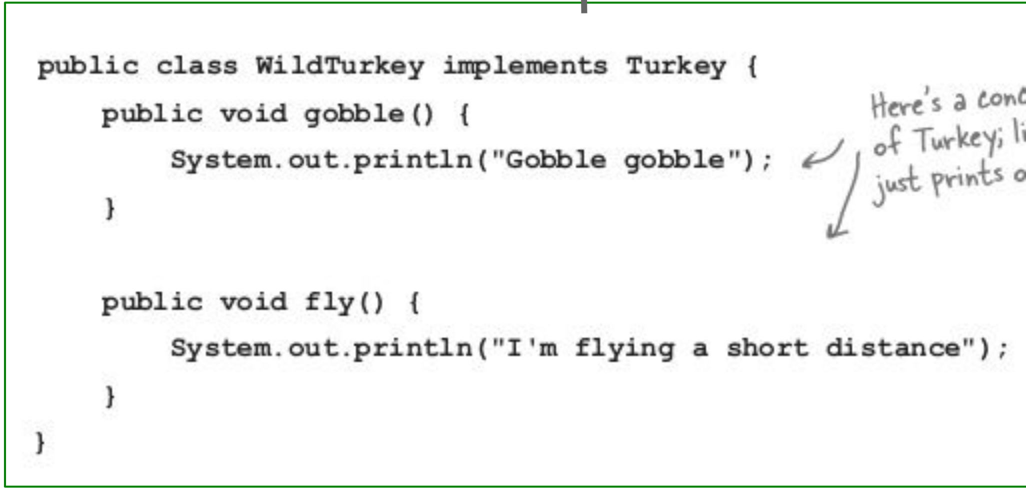
```
public interface Duck {
    public void quack();
    public void fly();
}
```



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



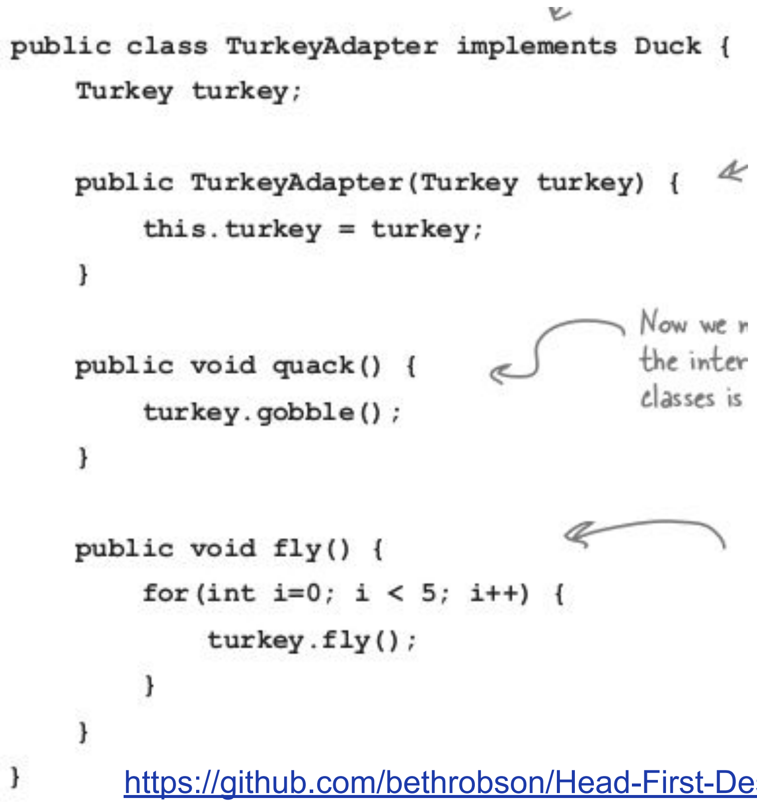
```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

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



```
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 conc
of Turkey; li
just prints o

Turkey Adapter



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

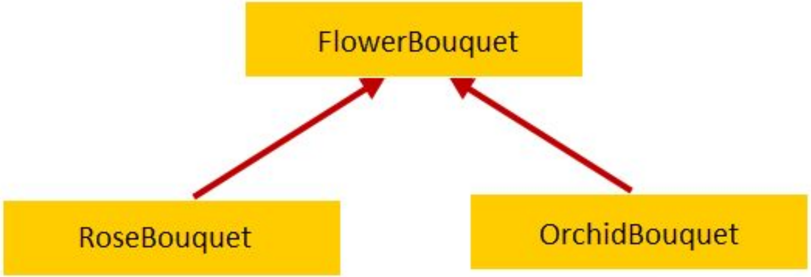
Now we r
the inter
classes is

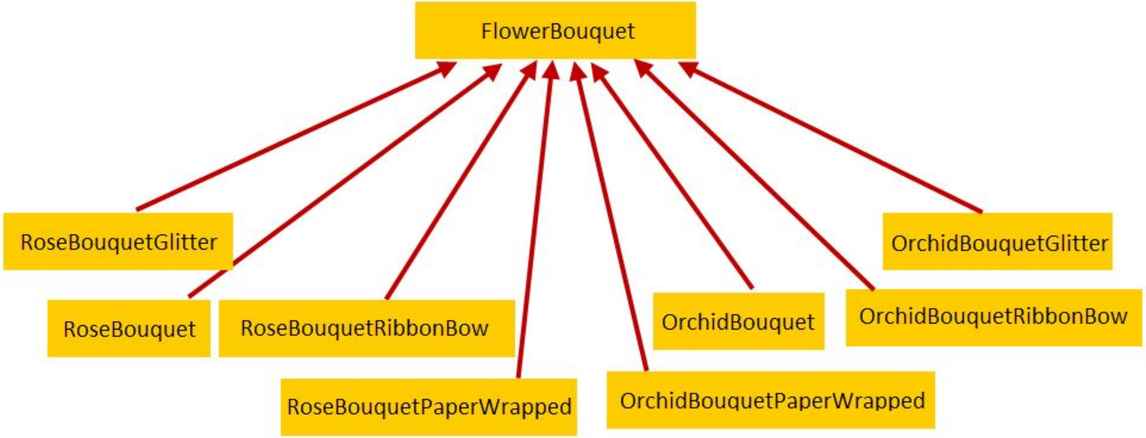
Duck Simulator Demo

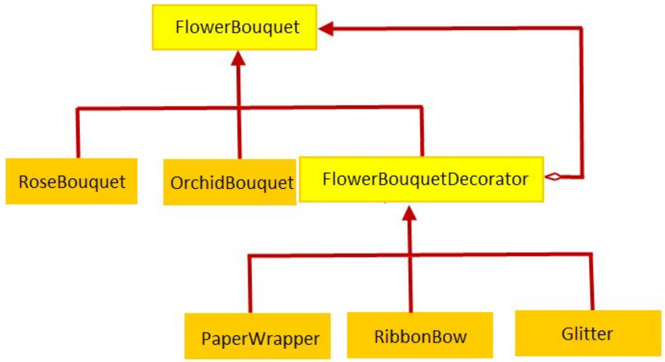
```
Turkey turkey = new WildTurkey();
Duck duck = new TurkeyAdapter(turkey);

duck.quack();
duck.fly();
```

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/adapter/ducks

Decorator: Flower Bouquet example






FlowerBouquet
RoseBouquet
OrchidBouquet
FlowerBouquetDecorator
PaperWrapper
RibbonBow
Glitter

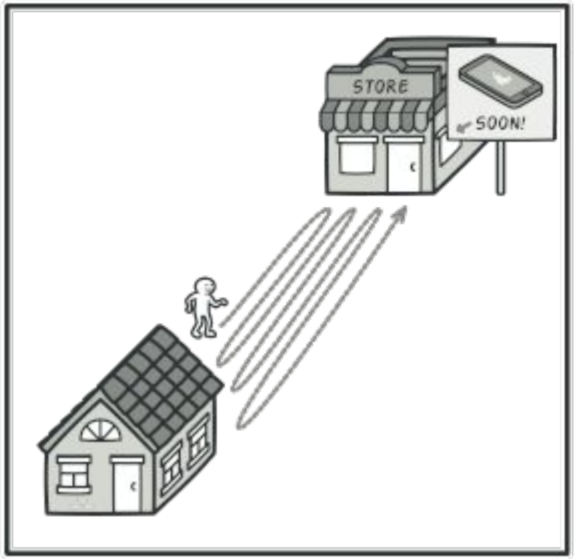
Behavioural Pattern

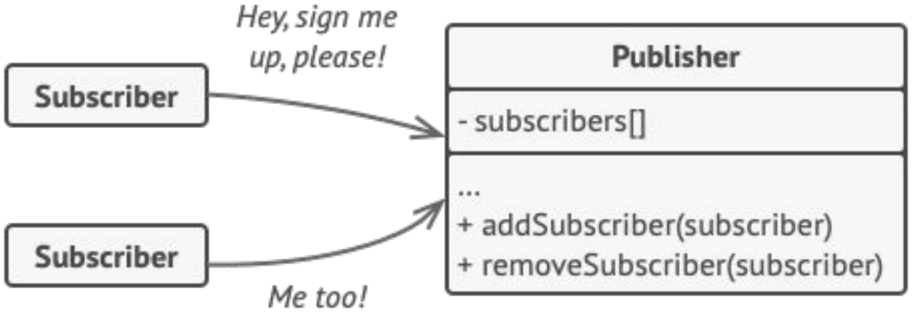
Observer Pattern

Source: https://refactoring.guru/design-patterns/iterator

Example Scenario

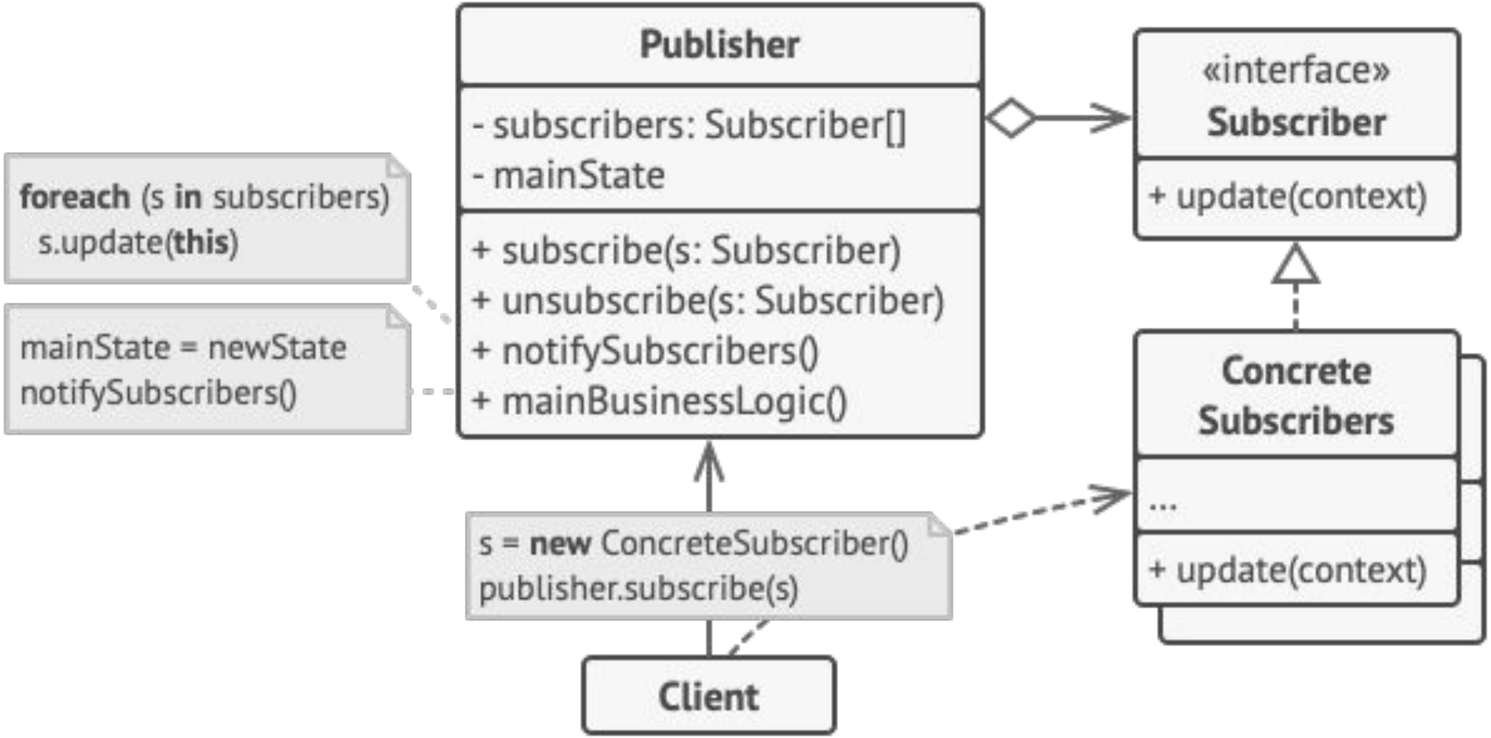
Get updates from a store when a specific phone is available.



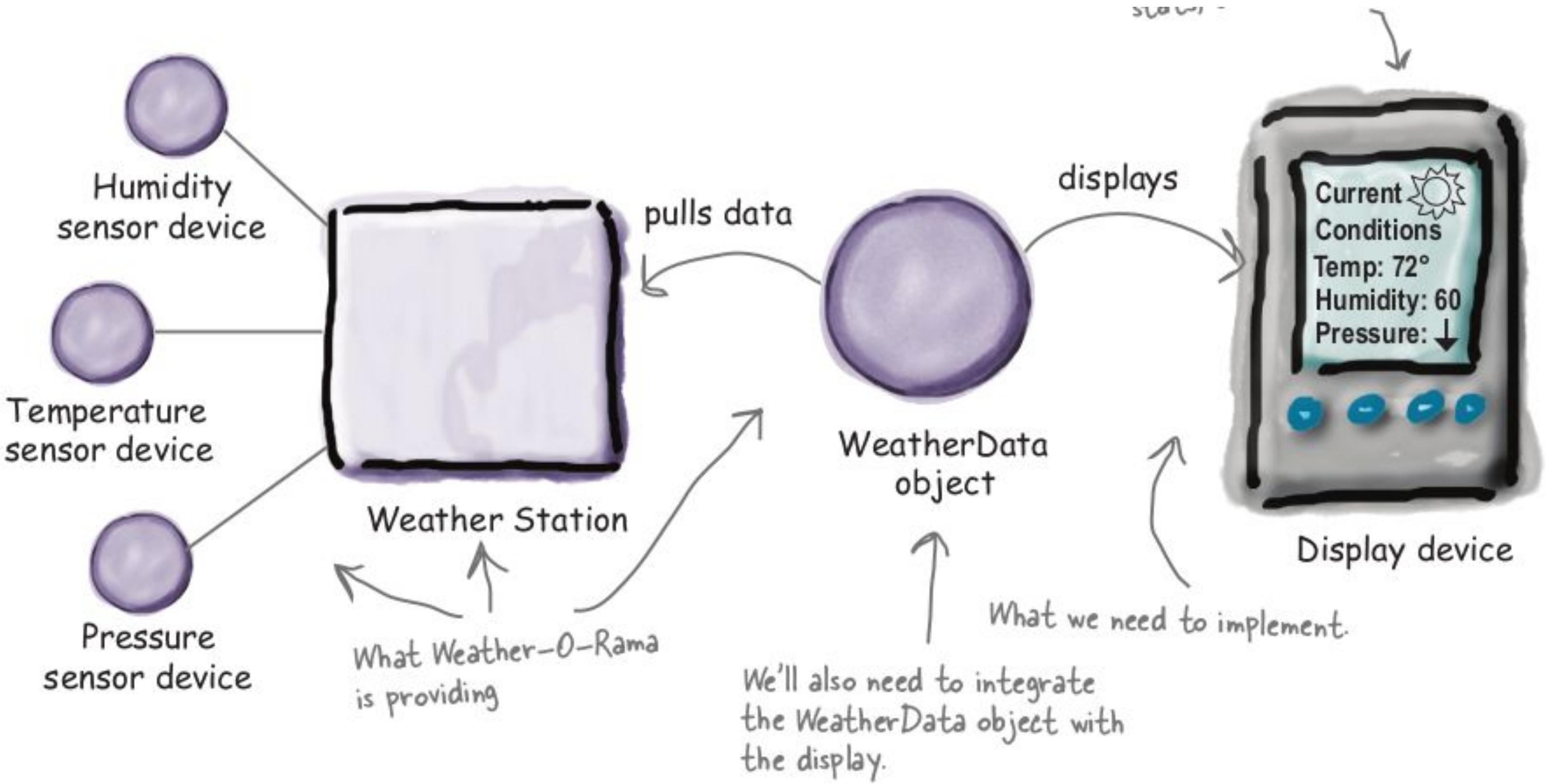


source: https://refactoring.guru/design-patterns/observer

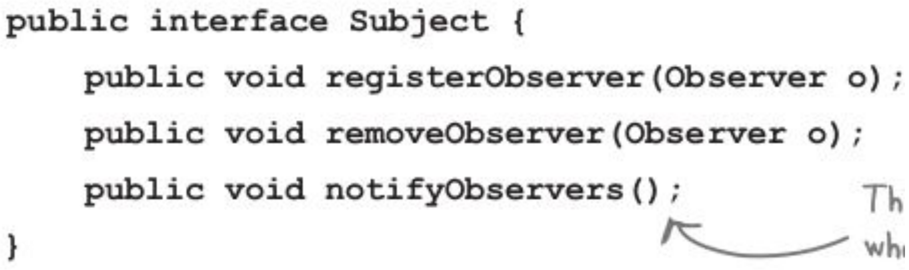
Solution with Observer Pattern



Observer Pattern: Weather Monitoring Application

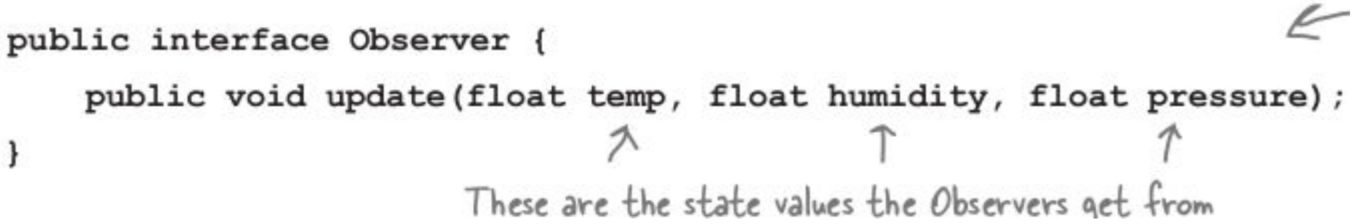


Weather Application: Base Classes



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

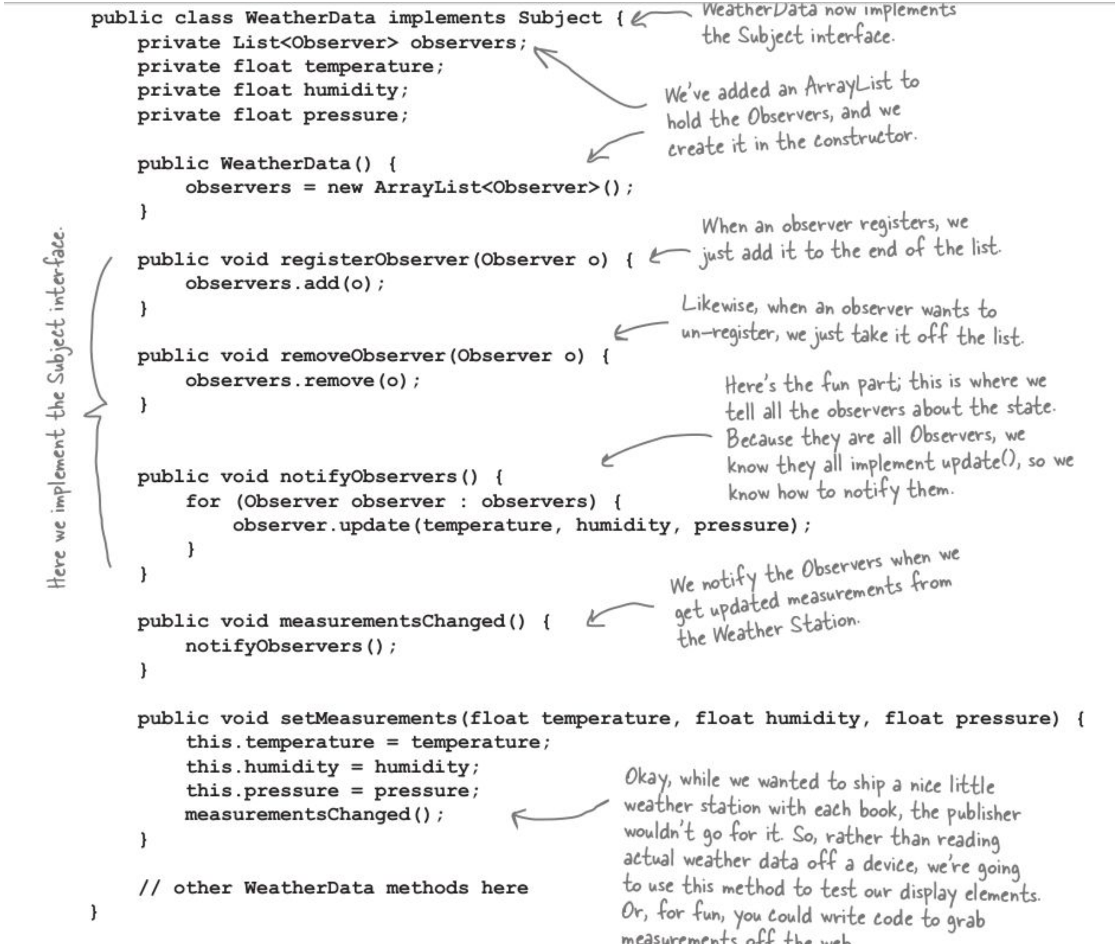
Th
wh



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

These are the state values the Observers get from

Weather Application: Publisher Concrete Class



```
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

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

    public void removeObserver(Observer o) {
        observers.remove(o);
    }


    public void notifyObservers() {
        for (Observer observer : observers) {
            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();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

Here we implement the Subject interface.

When an observer registers, we just add it to the end of the list.

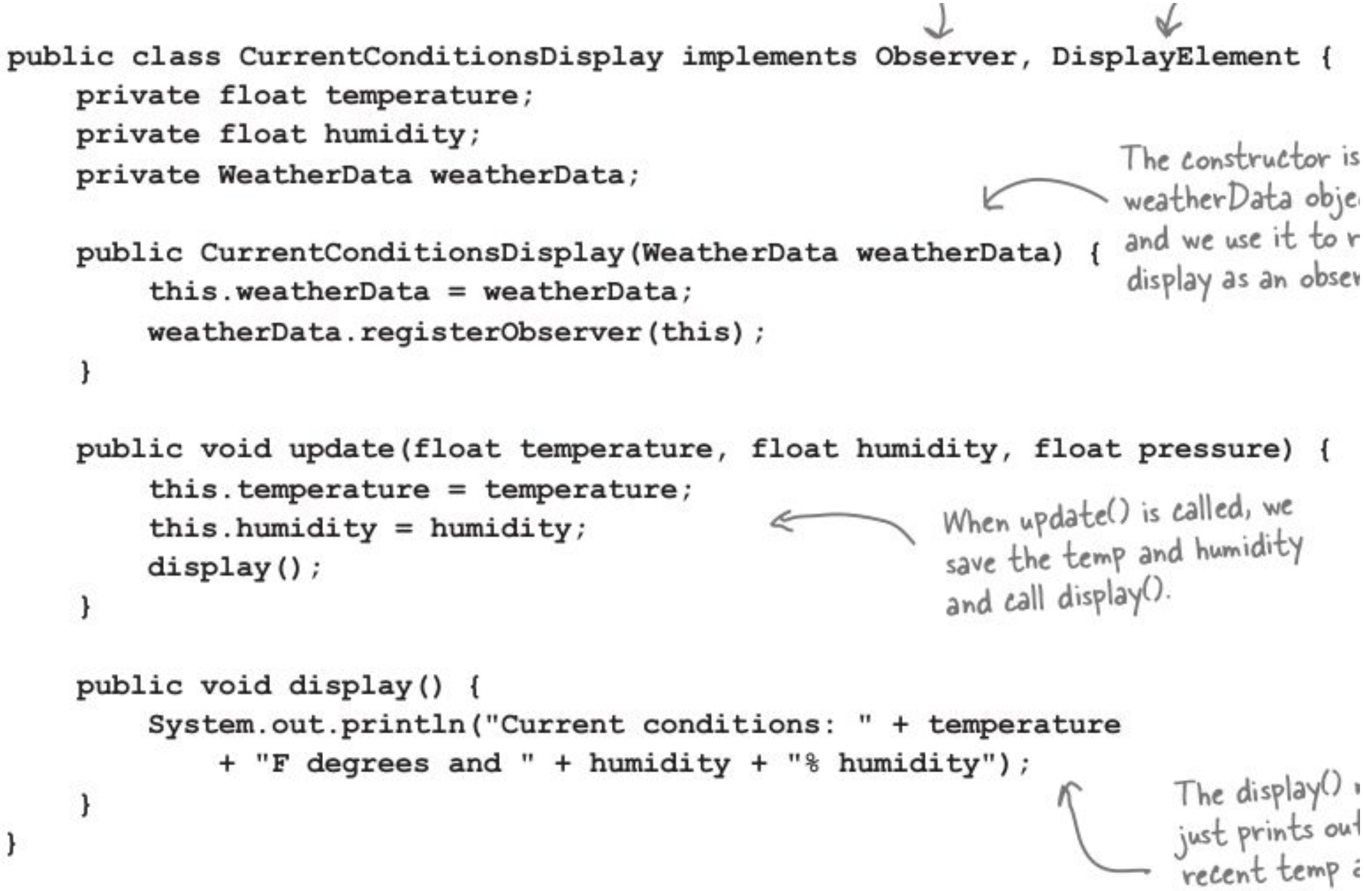
Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Weather Application: Observer Concrete Class



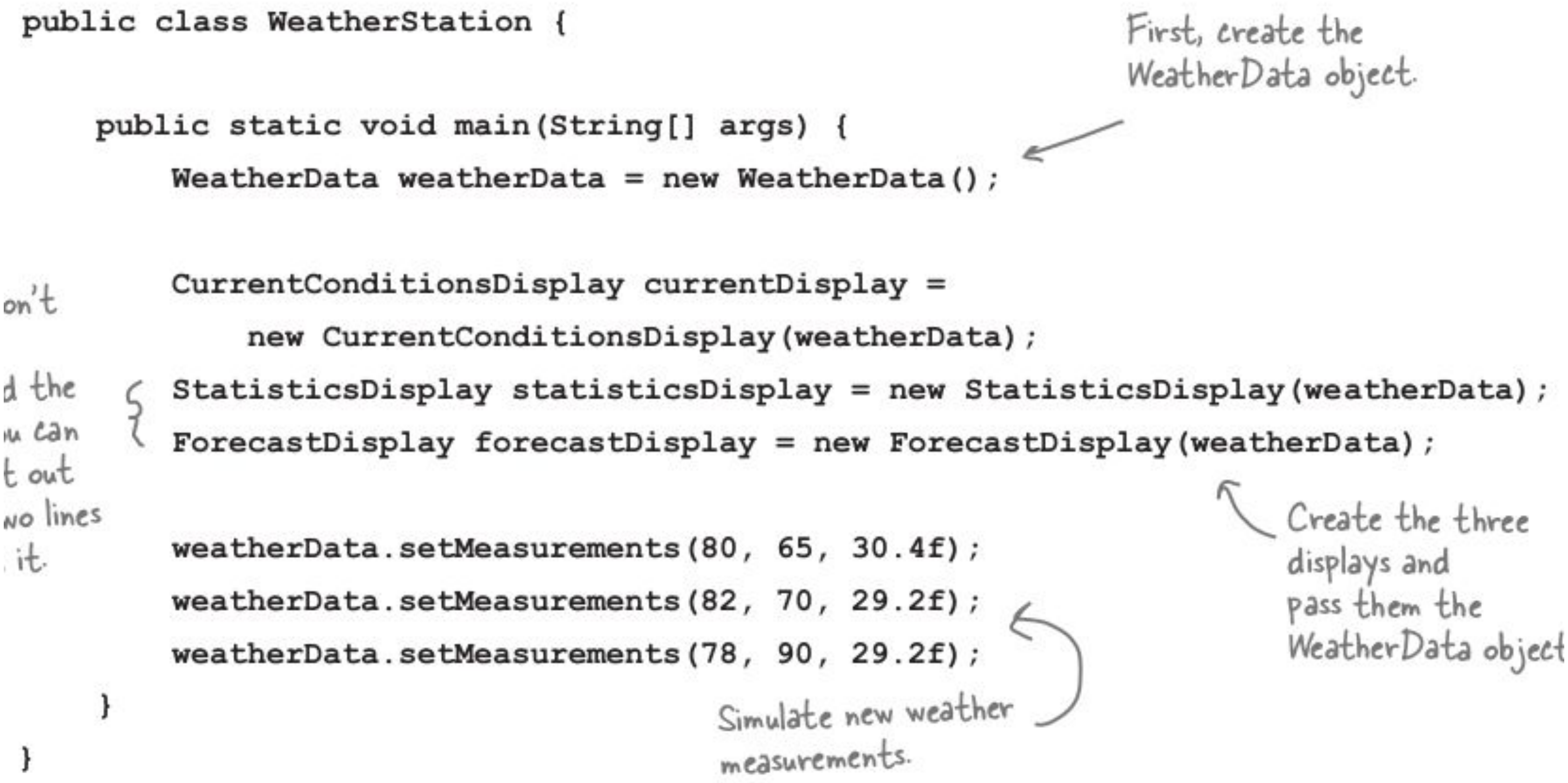
```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private WeatherData weatherData;

    public CurrentConditionsDisplay(WeatherData 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");
    }
}
```

The constructor is
weatherData obje
and we use it to r
display as an obser

When update() is called, we
save the temp and humidity
and call display().

The display()
just prints out
recent temp a

Weather Application: Client Code



```
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);
    }
}
```

First, create the WeatherData object.

on't

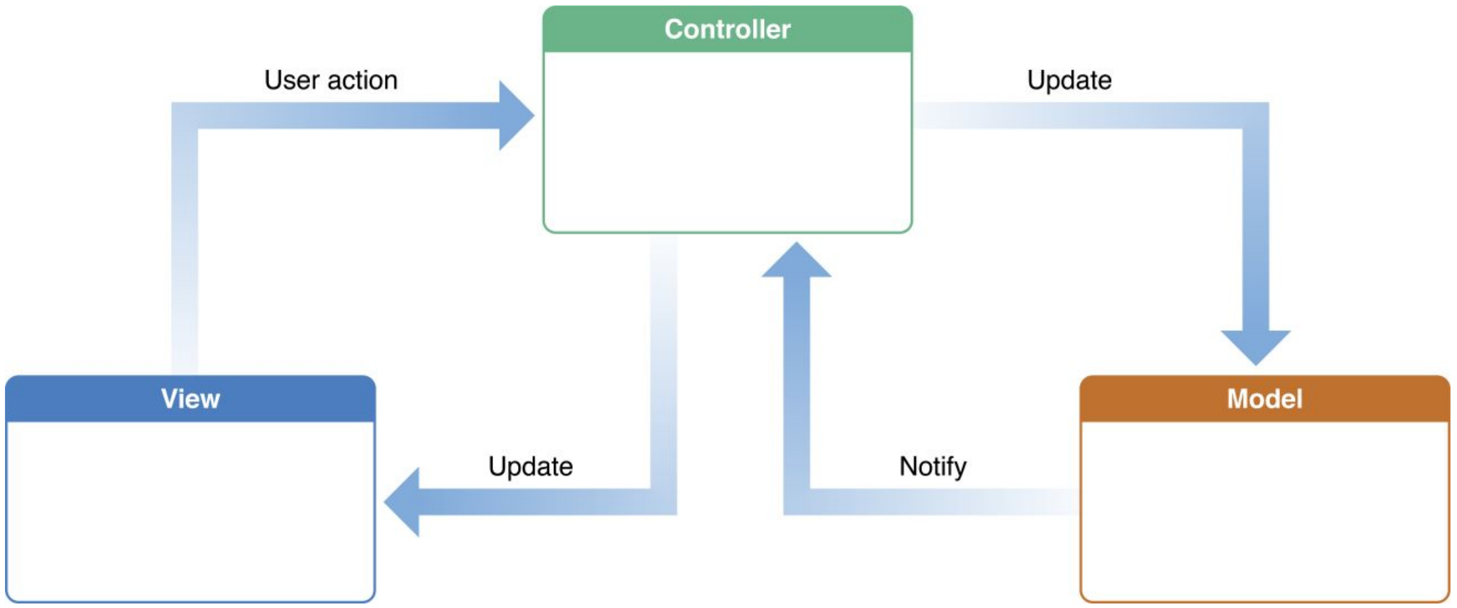
d the
u can
t out
wo lines
it.

Create the three displays and pass them the WeatherData object

Simulate new weather measurements.

Model View Controller (MVC)

MVC is an architectural pattern.
Architectural patterns have a broader scope compared to Design Patterns.



source: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

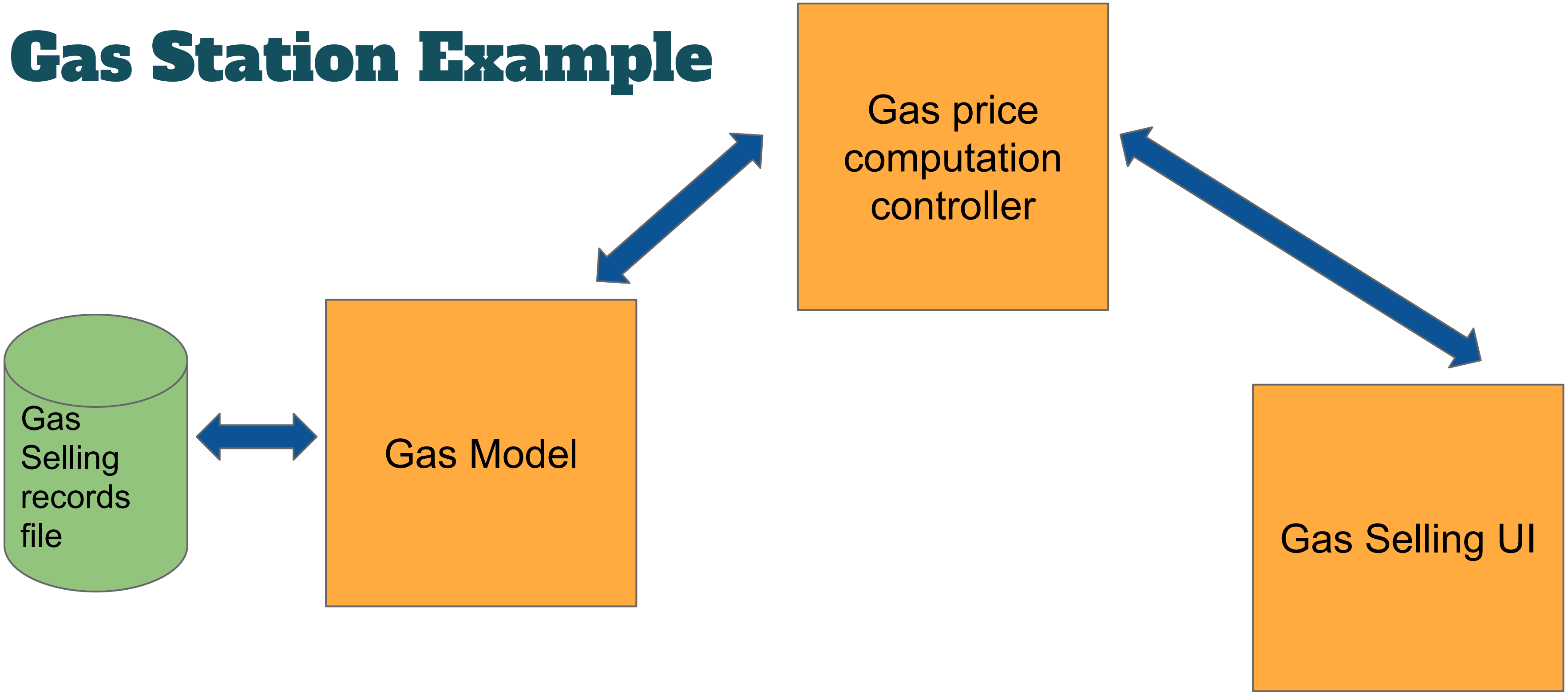
MVC: Components

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data.

View object is an object in an application that users can see.

Controller object acts as an intermediary between Model and View.

Gas Station Example



Source https://www.makeuseof.com/introduction-to-mvc-architecture-examples/

Model

```
Public class GasPriceModel implements Serializable{
       //attributes
     private static final long private String
driverName;
     private float gasAmount;
     private String gasType;
     private float cost;

     //primary constructors
     public GasPriceModel(String driverName, float
gasAmount, String gasType, float cost) {
           this.driverName = driverName;
           this.gasAmount = gasAmount;
           this.gasType = gasType;
           this.cost = cost;
     }
```

```
     public String getDriverName() {
           return driverName;
     }
     public void setDriverName(String
driverName) {
           this.driverName = driverName;
     }
     public float getGasAmount() {
           return gasAmount;
     }
     public void setGasAmount(float
gasAmount) {
           this.gasAmount = gasAmount;
     }
     public String getGasType() {
           return gasType;
     }
     public void setGasType(String gasType) {
           this.gasType = gasType;
     }

     public float getCost() {
           return cost;
     }
     public void setCost(float cost) {
           this.cost = cost;
     }
}
```

Controller

```
public class GasPriceController {
     //calculates the cost of a customer's gas and
returns it
     public float calculateCost(float amount,
String gasType){
           float cost = 0.00f;
           final float dieselPrice = 4.925f;
           final float premiumPrice = 5.002f;
           final float regularPrice = 4.680f;

           if (gasType == "Diesel")
                 cost = amount * dieselPrice;
           if (gasType == "Premium")
                 cost = amount * premiumPrice;
           if (gasType == "Regular")
                 cost = amount * regularPrice;

           return cost;
     }
```

```
 //saves the data from each sale to a file
using the model
     public boolean saveEntry(GasPriceModel
data){
           try {
                 FileOutputStream fs = new
FileOutputStream(new File("data.dat"), true);
                 ObjectOutputStream os = new
ObjectOutputStream(fs);
                 os.writeObject(data);
                 os.flush();
                 os.close();
                 return true;
           } catch (FileNotFoundException e) {
                 e.printStackTrace();
           } catch (IOException e) {
                 e.printStackTrace();
           }
           return false;
     }
}
```

View

```
public class GasPriceView extends JFrame implements
ActionListener {
     private static final long serialVersionUID = 1L;
     private GasPriceController controller;
     private JLabel driverName;
     private JTextField nameField;
     private JLabel gasAmount;
     private JTextField amountField;
     private JLabel gasType;
     private JComboBox<String> typeCombo;
     private JButton btnClear;
     private JButton btnSave;
     private static final String[] type =
           {"Diesel", "Premium", "Regular"};
     public GasPriceView() {
           this(new GasPriceController());
     }
     public GasPriceView(GasPriceController controller) {
           super("Gas Sale Application");
           setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           setSize(400,500);
           setVisible(true);
           this.controller = controller;
           configureView();

     }
```

```
private void configureView() {
    setLayout(new BorderLayout());
    JPanel pnl = new JPanel(new GridLayout(4,2,2,2));
    driverName = new JLabel("Driver's Name:");
    pnl.add(driverName);
    nameField = new JTextField();
    pnl.add(nameField);
    gasAmount = new JLabel("Gas Amount (Gallon):");
    pnl.add(gasAmount);
    amountField = new JTextField();
    pnl.add(amountField);
    gasType = new JLabel("Gas Type:");
    pnl.add(gasType);
    typeCombo = new JComboBox<String>(type);
    pnl.add(typeCombo);
    btnClear = new JButton("Clear");
    pnl.add(btnClear);
    btnSave = new JButton("Save");
    pnl.add(btnSave );
    add(pnl, BorderLayout.CENTER);
    ActionListener();
}
```