CS 619 Introduction to OO Design and Development

GoF Patterns (Part 2)

Fall 2012

Open Closed Principle

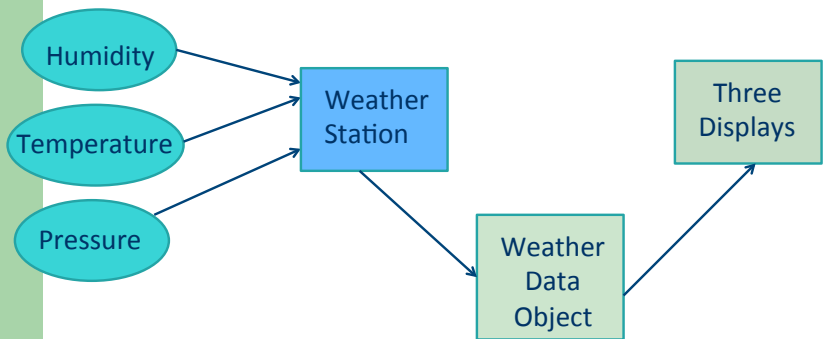
- Software entities (Classes, Modules, Methods, etc.) should be open for extension, but closed for modification.
- Also Known As Protected Variation
- Question: This principle is applied to which of the design patterns we have learned so far?

Single Responsibility Principle

- A Class should have one reason to change
  - A Responsibility is a reasons to change
- Can be tricky to get granularity right
- Single Responsibility = increased cohesion
- Not following results in needless dependencies
  - More reasons to change.
  - Rigidity, Immobility
- Question: This principle is applied to which of the design patterns we have learned so far?

Consider this…

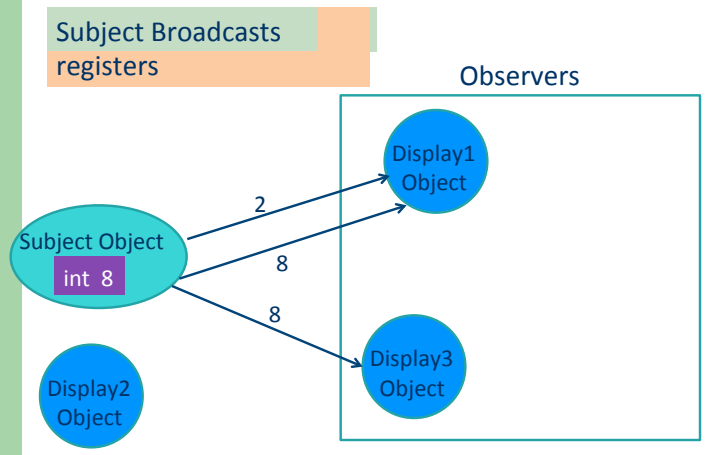
- The WeatherData class has getter methods that obtain measurement values from temperature, humidity and pressure. The class has a *measurementsChanged()* method that updates the three values.
- Three displays must be implemented: current conditions, statistics and forecast display. System must be expandable.



A First Attempt

```
public class WeatherData{
   //instance variables
   public void measurementChanged(){
      float temp = getTemperature();
      float humidity = getHumidity();
      float pressure = getPressure();
      currentConditionsDisplay.update(temp,humidity,
                                      pressure)
      statisticsDisplay.update(temp,humidity,pressure)
      forecastDisplay.update(temp,humidity,pressure)
   }
   //other WeatherData methods here
}
```


The Observer Pattern "Observed"



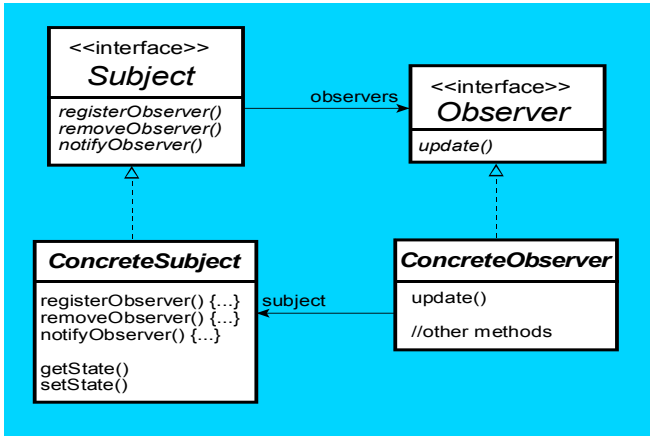
Observer Pattern

- The observer pattern implements a one-to-many relationship between a set of objects.
- A single object changes state and updates the objects (*dependants*) that are affected by the change.
- The object that changes state is called the *subject* and the other objects are the *observers*.

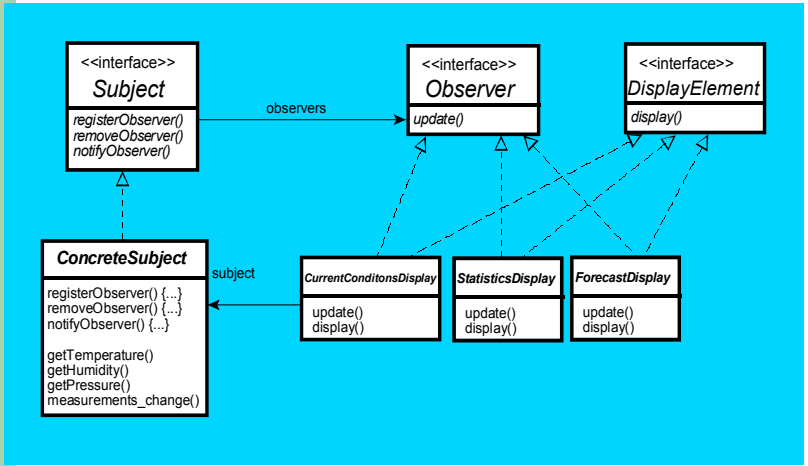
Loose Coupling

- Subjects and observers are loosely coupled.
- The subject only knows the observer interface and not its implementation.
- Observers can be added and removed at any time.
- In adding new observers the subject does not need to be modified.
- Subjects and observers can be reused independently.
- Changes to the subject or observer will not affect the other.

The Class Diagram



Weather Station Class Diagram



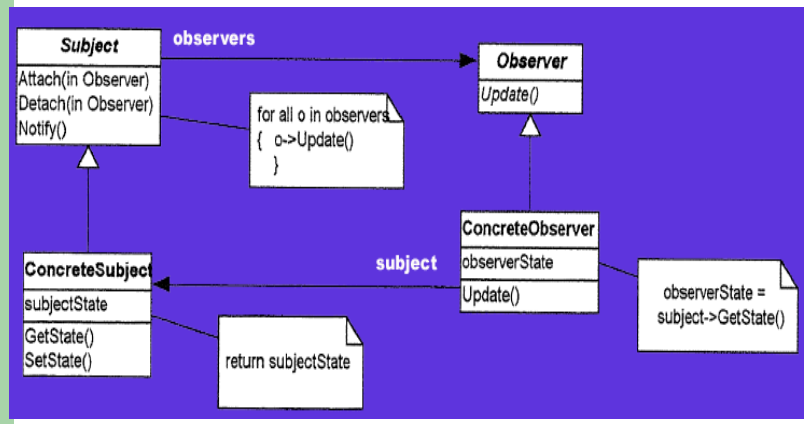
The Solution

- All observers must have the same interface and register themselves. This makes them responsible for knowing what they are watching for.
- We must add two methods to the subject.

  `attach(Observer)` adds the given observer to its list of observers.

  `detatch(Observer)` removes the given observer from its list of observers.
- The observer must implement a method called **`update`**.
- The subject implements the **`notify`** method that goes through its list of Observers and calls this update method for each of them.

Observer Pattern: Class Diagram



Observer Applicability

- Use the Observer pattern in any of the following situations
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independtly.
  - When a change to one object requires changing others, and you do not know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are.

The Observer Pattern

- **Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Problem:** You need to notify a varying list of objects that an event has occurred.
- **Solution:** Observers delegate the responsibility for monitoring for an event to a central object: The subject.

Participants:

Subject

Knows its observers. Any numberof Observer objects may observe a subject.

Provides an interface for attaching and detaching Observer Objects.

Observer

Defines an updating interface for objects that should be notified of changes in a subject

ConcreteSubject

Stores a state of interest to ConcreteObserver objects.

Sends a notification to its observers when its state changes.

ConcreteObserver

Maintains a reference to a ConcreteSubject object

Stores state that should stay consistent with the subject state.

Implements the Observer updating interface to keep its state consistent with the subject state.

Implementation:

Have objects (Observers) that want to know when an event happen attach themselves to another object (Subject) that is watching for the event to occur or that triggers the event itself.

When the event occurs, the Subject tells the Observers that it has occurred.

The Adapter pattern is sometimes needed to be able to implement the Observer interface for all the Observer-type objects.

Consequences

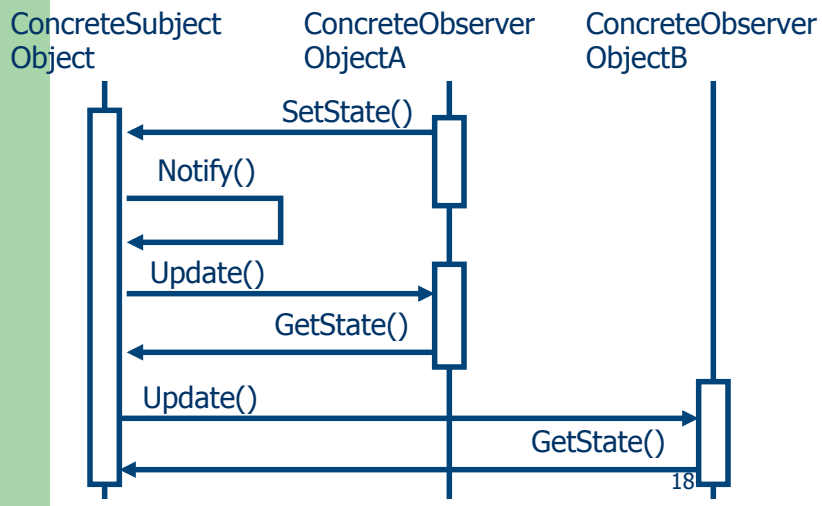
Abstract coupling between Subject and Object

All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject does not know the concrete class of any observer.

Support for broadcast communication

Unlike and ordinary request, the notification that a subject sends need not specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it.

Observer Sequence Diagram



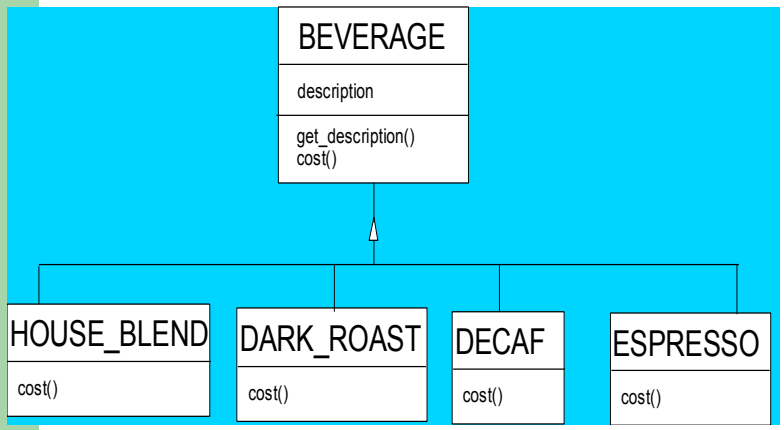
Decorator Pattern

Example: StarBuzz Coffee

- Starbuzz coffee sells different beverages.
- The cost of each beverage is calculated differently.
- The beverages sold by Starbuzz are HouseBlend, DarkRoast, Decaf and Espresso.
- Starbuzz needs a system to calculate the cost for each beverage when purchases are made.

Draw a class diagram for the system.

Class Diagram



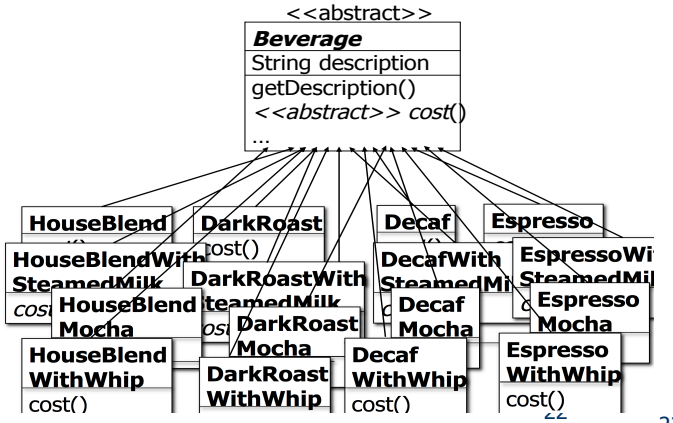
Revisions to System

In addition to standard coffee a customer can ask for several condiments such as steamed milk, soy, and mocha and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they require them to be built into their order system.

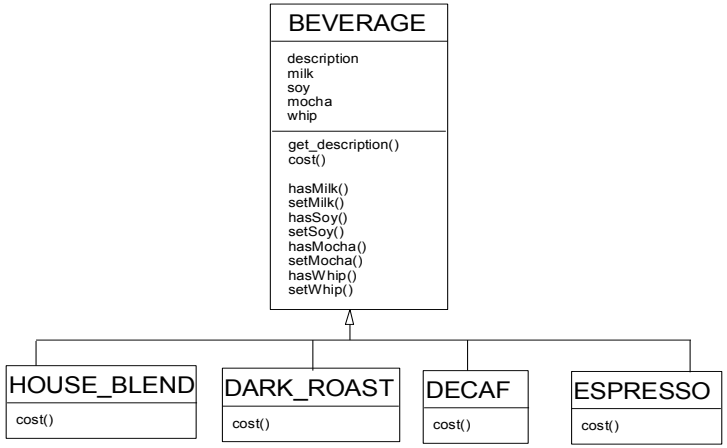
What do we do to the class diagram?

First Design

- Make a beverage class and a subclass for each legal combination



Another Option



Problems with Inheritance

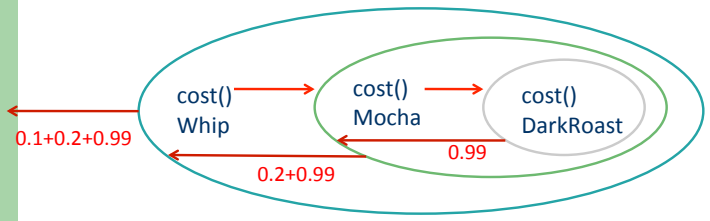
- Too many subclasses may be needed.
- Need to define the implementation of each subclass.
- All classes inherit a static behaviour which may have to be overridden.
- The inherited behaviour cannot be changed at runtime, i.e. behaviour is static at runtime.
- Thus, the design is not flexible or maintainable.

Design Principle

- In order to maintain code we should aim at adding new functionality by adding new code rather than changing existing code.
- Design principle: classes should be open for extension but closed for modification.
- In this way the chances of introducing bugs or causing unintended side effects are reduced.
- Only apply this to areas that are most likely to change.

Applying Decorators to the Starbuzz Example

Order: Dark Roast with Mocha and Whip



Calculating the cost t Object

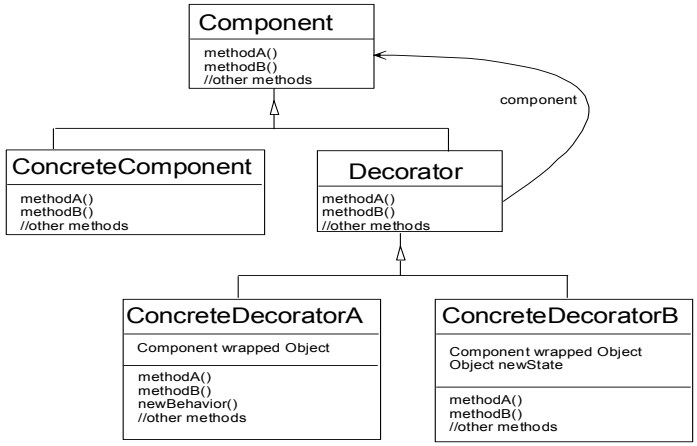
Design Principle for the Decorator Pattern

- Classes should be open for extension but closed for modification.
- Systems can be extended without changing existing code.
- Tradeoffs:
  - Takes time and effort.
  - Introduces new levels of abstraction which makes designs more complicated and code hard to understand.
- Due to the tradeoffs the decorator pattern should not be used throughout the design but in areas that are most likely to change.
- Experience and looking at other examples helps one determine which areas are likely to change.

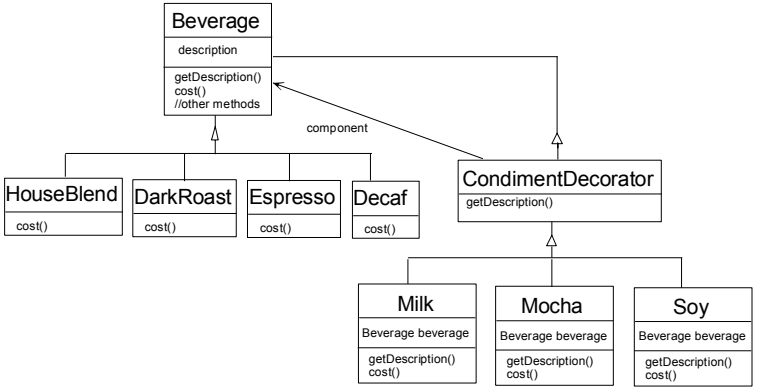
Decorator Pattern

- Provides a flexible alternative to using inheritance to extend functionality.
- This achieved by introducing decorators that "decorate" objects.
- Decorators have the same type as the objects they decorate.
- An object can have one or more decorators.

Class Diagram for the Decorator Pattern



Starbuzz Coffee Example



The Decorator Pattern

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

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();
  }
}
```

The Decorator Pattern

```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
  + " $" + beverage.cost());

Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);

System.out.println(beverage2.getDescription()
  + " $" + beverage2.cost());
```

The Decorator Pattern

- The decorated object and the original object have the same type. Thus, the decorated object can be passed instead of the original object.
- The decorator delegates part of its behaviour to the object it decorates.
- It adds its own behaviour before or after the delegation.
- Objects can be delegated dynamically at runtime.
- The objects are "wrapped" with decorators.

Decorator Pattern Overview

- Adds flexibility to designs.
- Results in a number of small classes being added to the design which makes the project more complicated.
- The decorators are a set of wrappers around an object of an abstract class or interface.
- Code maybe dependant on specific types and introducing decorators can lead to unexpected side effects.
- Code needed to instantiate the component can be complex as a number of decorators have to be wrapped around the object.

Decorator Pattern Summary

- Main design principle: Classes should be open for extension but closed for modification.
- Achieves flexibility- behaviour can be extended without changing existing code.
- Composition and delegation is used to add new behaviours at runtime.
- A set of decorator classes are used to wrap concrete components.
- Overuse of decorators can be complex as they can introduce a number of small objects into a design.

Exercise

- Suppose that StarBuzz has now introduced sizes to their menu.
- Three sizes of coffee can now be ordered: tall, grande and venti.
- Condiments have to be charged according to size. For example, Milk costs 10c, 15c and 20c respectively for tall, grande and venti.
- What changes need to be made to the classes?

The Decorator Pattern

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

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

  public double cost() {
     double cost = beverage.cost();
     if (getSize() == Beverage.TALL)
         cost += .10;
     else if (getSize() == Beverage.GRANDE)
         cost += .15;
      else if (getSize() == Beverage.VENTI)
          cost += .20;
     return cost
}
```

Factory Pattern:

Consider the Following Code Fragment

```
Duck duck;

if (picnic) {
  duck = new MallardDuck();
} else if (hunting) {
  duck = new DecoyDuck();
} else if (inBathTub) {
  duck = new RubberDuck();
```

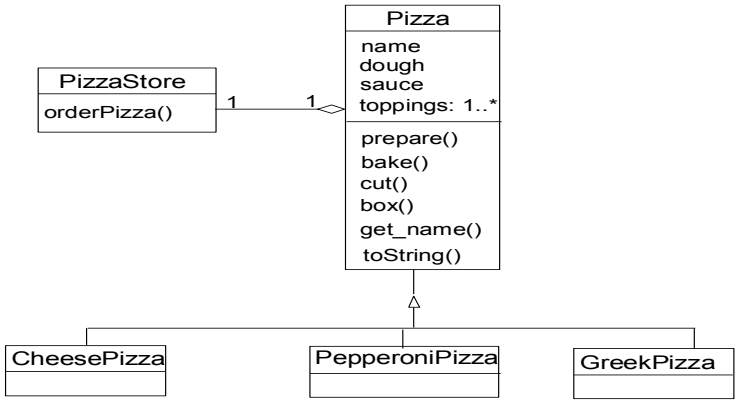
When you have several related classes, that's probably a good sign that they might change in the future

Example

Suppose that you are required to develop a system that accepts orders for pizzas. There are three types of pizzas, namely, cheese, Greek, and pepperoni. The pizzas differ according to the dough used, the sauce used and the toppings.

Draw a class diagram for the system.

Class Diagram



Problems with the design?

Design Principle

- What should you try to do with code that changes?

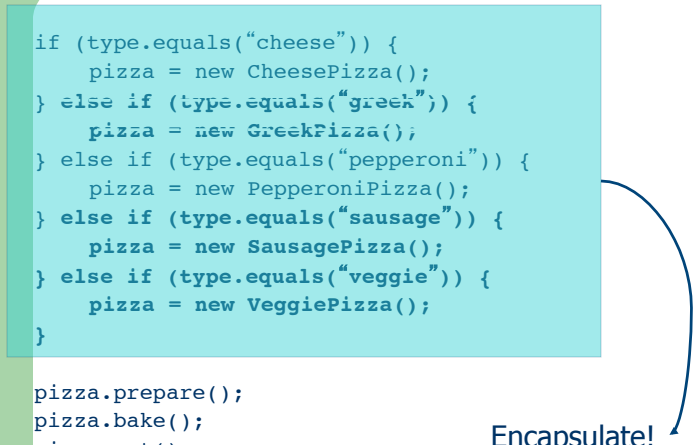
```
Pizza orderPizza() {
  Pizza pizza = new Pizza();

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

  return pizza;
}
```

Revised System

- Suppose that the Greek pizza is not popular and must be removed and two new pizzas, Clam and Veggie, must be added to the menu.
- Programming to implementation like makes such changes difficult.
- Creating a SimpleFactory to encapsulate the code that changes will make the design more flexible.
- Remove the code that creates a pizza – forms a factory.

```
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();
  } else if (type.equals("sausage")) {
      pizza = new SausagePizza();
  } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();
  }

  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();
  } else if (type.equals("sausage")) {
      pizza = new SausagePizza();
  } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();
  }

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

     return pizza;
   }
}
```

No new() operator

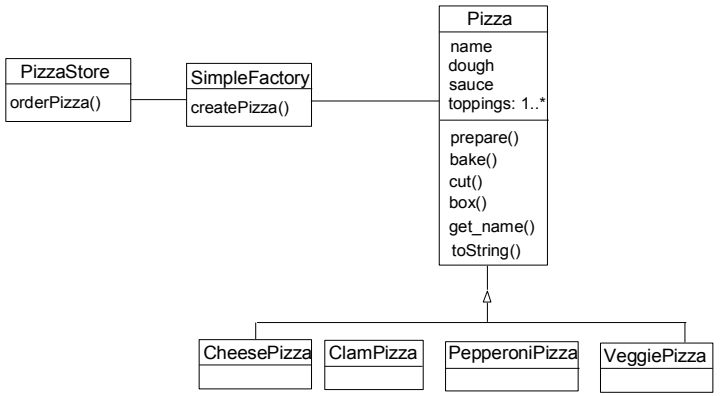
The Simple Factory Pattern

```
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;
  }
}
```

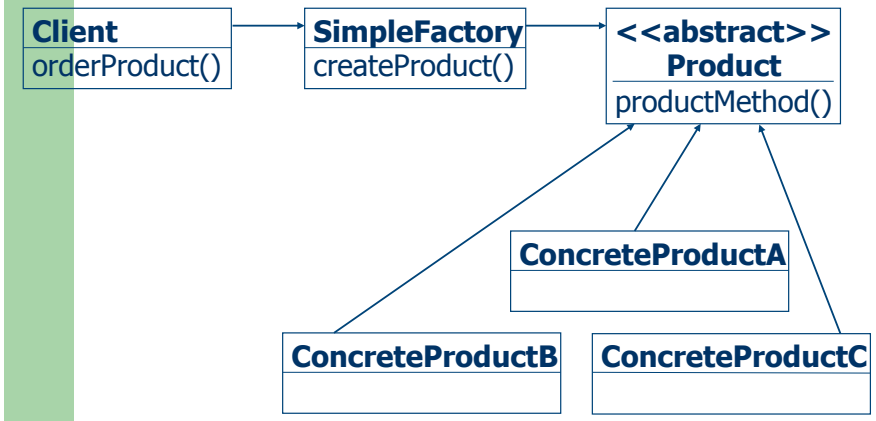
Instantiating Concrete Classes

- Using *new* instantiates a concrete class.
- This is programming to **implementation** instead of **interface**.
- Concrete classes are often instantiated in more than one place.
- Thus, when changes or extensions are made all the instantiations will have to be changed.
- Such extensions can result in updates being more difficult and error-prone.

Example: Revised Class Diagram



Simple Factory Pattern

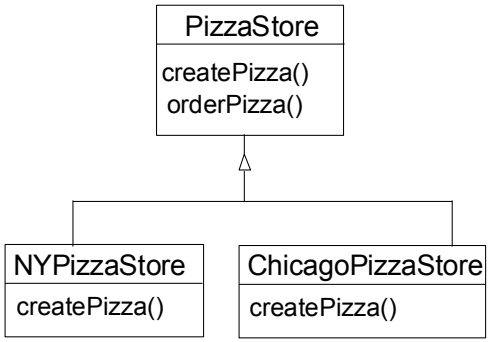


Example: 2nd System Revision

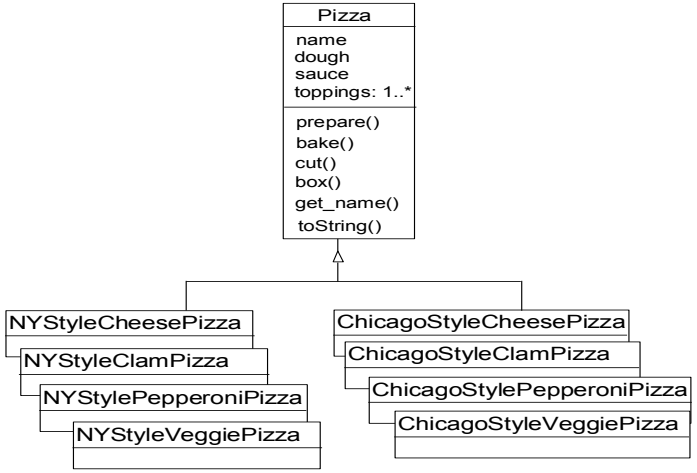
Franchises in different parts of the country are now adding their own special touches to the pizza. For example, customers at the franchise in New York like a thin base, with tasty sauce and little cheese. However, customers in Chicago prefer a thick base, rich sauce and a lot of cheese.

You need to extend the system to cater for this.

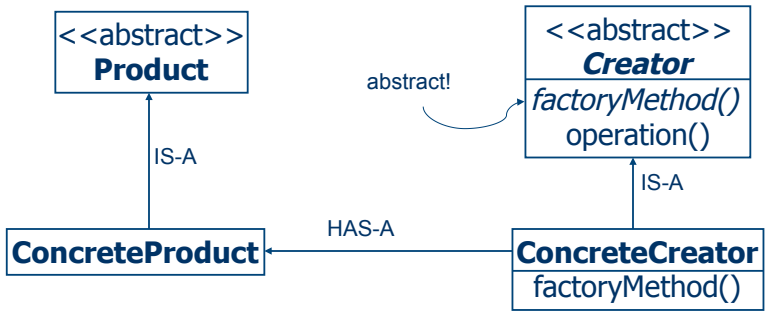
Creator Class



Product Class



Factory Method Pattern



No more SimpleFactory **class**
Object creation is back in our class, but …
delegated to concrete classes

The Factory Method Pattern

```
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;
  }
}
```

The Factory Method Pattern

```
PizzaStore nyStore = new NYPizzaStore();
PizzaStore chicagoStore = new ChicagoPizzaStore();

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

pizza = chicagoStore.orderPizza("cheese");
System.out.println("Joel ordered a " + pizza.getName()
+ "\n");

pizza = nyStore.orderPizza("clam");
System.out.println("Ethan ordered a " +
pizza.getName() + "\n");
```

Applicability

- Use Factory Method when:
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
  - Classes delegate to one of several helper subclasses, and you want to localize knowledge about which is the delegate

Participants:

- Product
  - Defines interface of objects to be created
- ConcreteProduct
  - Implements Product interface
- Creator
  - Declares factory method
    - Returns object of type Product
    - May define default implementation
  - May call factory method to create Products

Consequence:

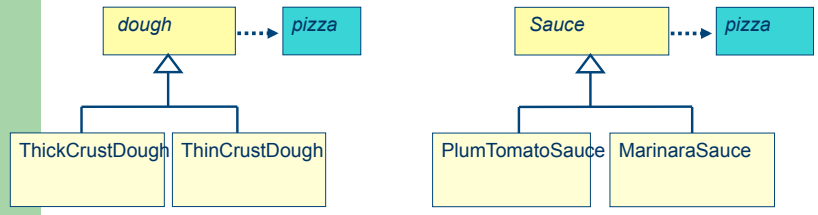
- Eliminates need to bind creation code to specific subclasses
- May need to subclass Creator for each ConcreteProduct
- Provides hooks for subclasses
- Connects parallel class hierarchies

The Abstract Factory

- Our code can now work with different concrete factories, through a *Factory* interface (Pizzastore)
- What if we need to create several types of ”products”, not just a single type?
  - Pizza dough (Thin in NY, Thick in Chicago)
  - Sauce (…)

The Abstract Factory

- Answer seems simple: just use Factory Method pattern twice

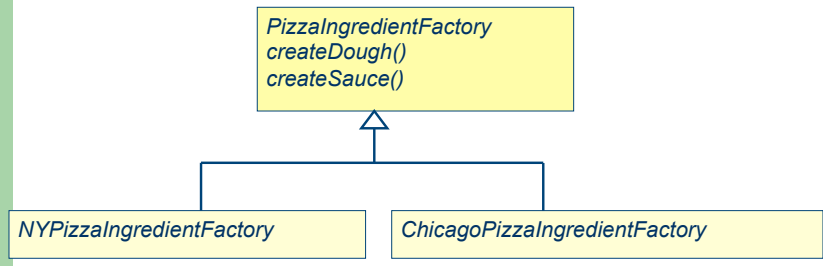


This looks fine…but does it reflect our intention?
Would it really make sense to have a pizza, with Chicago Thick Crust + NY Marinara Sauce?
Model should not include any ”binding” between related products.

The Abstract Factory

- A ***Dough*** and a ***Sauce*** are not – as seen from a type point-of-view – related
- Would be somewhat artificial – or perhaps even impossible – to introduce a common base class
- However, we can enforce the binding through a shared factory class!

The Abstract Factory



More Ingredience

- We want to list the exact list of ingredients for each concrete pizza. For example :
  - Chicago Cheese Pizza : Plum tomato Sauce, Mozzarella, Parmesan, Oregano;
  - New York Cheese Pizza : Marinara Sauce, Reggiano, Garlic.
- Each "style" uses a different set of ingredients,
- We could change implement Pizza as in:

```
public class Pizza {
   Dough dough;
   Sauce sauce;
   Veggies veggies[];
   Cheese cheese;
   Pepperoni pepperoni;
   Clams clam;
   . . .
}
```


- and the constructor naturally becomes something like :

```
public Pizza(Dough d, Sauce s, Cheese c, Veggies v,
             Pepperoni p, Clams c)
{
  dough = d;
  sauce = s;
  veggies = v;
  cheese = c;
  pepperoni  = p;
  clam = c;
}
```


- but then:

```
public class NYPizzaStore extends PizzaStore {
  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new NYStyleCheesePizza(new ThinCrustDough(),
             new Marinara(), new Reggiano(), null, null );
    } else if (item.equals("veggie")) {
      return new NYStyleVeggiaPizza(new ThinCrustDough(),
            new Marinara(), new Reggiano(), mew Garlic() ,
           null);
    }
  }
```

This will cause a lot of maintenance headaches!
Imagine what happens when we create a new pizza!

- We know that we have a certain set of ingredients that are used for New York..yet we have to keep repeating that set with each constructor. Can we define this unique set just once?
- After all we are creating concrete instances of dough, ingredients etc. :
  - let's use the factory of ingredients!

```
public interface PizzaIngredientFactory {

   public Dough createDough();
   public Sauce createSauce();
   public Cheese createCheese();
   public Veggies[] createVeggies();
   public Pepperoni createPepperoni();
   public Clams createClam();
}
```


- We then "program to the interface" by implementing different concrete ingredients factories. For example here are the ingredients used in New York pizza style:

```
public class NYPizzaIngredientFactory : PizzaIngredientFactory {
   public Dough createDough() {
        return new ThinCrustDough();
   }
   public Sauce createSauce() {
        return new MarinaraSauce();
   }
   public Cheese createCheese() {
        return new ReggianoCheese();
   }
   //...
}
```


- Our Pizza class will remain an abstract class:

```
public abstract class Pizza {
  protected string name;
  protected Dough dough;
  protected Sauce sauce;
  protected ArrayList toppings = new ArrayList();

  abstract void Prepare(); //now abstract
  public virtual string Bake() {
    Console.WriteLine("Bake for 25 minutes at 350 \n");
  }
  public virtual string Cut() {
    Console.WriteLine("Cutting the pizza into diagonal slices \n");
  }
// ...
}
```


- and our concrete Pizza simply become:

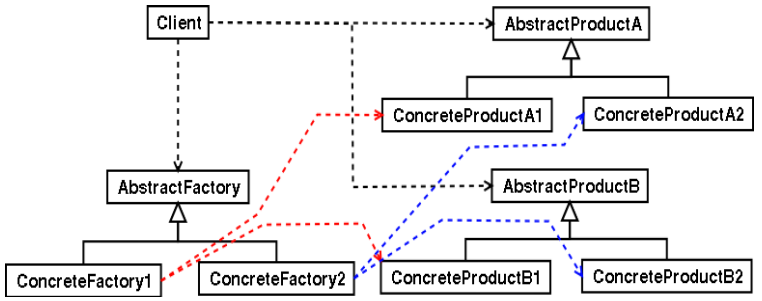
```
public class CheesePizza : Pizza {
   PizzaIngredientFactory ingredientFactory;
   public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
   }
}
```

the creation of the ingredients is delegated to a factory.

- finally we must have our concrete Pizza store e.g.

```
public class NYPizzaStore : PizzaStore {
   protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
                new NYPizzaIngredientFactory();
        if (item.equals("cheese")) {
                pizza = new CheesePizza(ingredientFactory);
        } else if (item.equals("veggie")) {
                pizza = new VeggiePizza(ingredientFactory);
        } else if (item.equals("clam")) {
                pizza = new ClamPizza(ingredientFactory);
        } else if (item.equals("pepperoni")) {
                pizza = new PepperoniPizza(ingredientFactory);
        }
        return pizza;
   }
}
```


Abstract Factory: Class Diagram




```
public class Client{
    public static void main(String args[]){
       AbstractFactory pf=getFactory("a");  // →
       AbstractProductA product=pf.createProductA();
        //more function calls on product }
}
```

```
 if(kit.equals("a")){
     pf=new ConcreteFactory1();
  }
else if(kit.equals("b")){
     pf=new ConcreteFactory2();
}
return pf;
```

Abstract Factory: Participants

- *AbstractFactory*
  - declares interface for operations that create abstract products
- *ConcreteFactory*
  - implements operations to create concrete products
- *AbstractProduct*
  - declares an interface for a type of product object
- *ConcreteProduct*
  - defines the product object created by concrete factory
  - implements the AbstractProduct interface
- *Client*
  - uses only interfaces of ***AbstractFactory*** / ***AbstractProduct***

Abstract Factory – intent and context

- provides an interface for creating families of related or dependent objects without specifying their concrete classes
- use *AbstractFactory* when
  - a system should be independent of how its products are created, composed, represented
  - a system should be configured with one or multiple families of products
  - a family of related product objects is designed to be used together and you need to enforce this constraint
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory Summary

- The creator gives you an interface for writing objects.
- The creator specifies the "factory method", e.g. the createPizza method.
- Other methods generally included in the creator are methods that operate on the product produced by the creator, e.g. orderPizza.
- Only subclasses implement the factory method.

Programming to interface not implementation.

GoF Singleton Pattern

- In some cases there should be at most one instance of a class.
- This one instance must be accessible by all "clients", e.g. a printer spooler.
- This usually occurs when a global resource has to be shared.
- The singleton pattern ensures that a class has only one instance and provides only one point of entry.

Implementing the Singleton Pattern

- Implement a private constructor to prevent other classes from declaring more than one instance.
- Implement a method to create a single instance. Make this method static.
- Create a lazy instance of the class in the class.
- Make the data element static.

The Singleton Pattern

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

// other useful methods here
}
```

Thread Example

A dual processor machine, with two threads calling the *getInstance()* method for the chocolate boiler

Thread 1

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
        new ChocolateBoiler()

return uniqueInstance;
```

Thread 2

```
public stat ChocolateBoiler
        getInstance()

if (uniqueInstance == null)

uniqueInstance =
        new ChocolateBoiler()

return uniqueInstance;
```


Problems with Multithreading

- In the case of multithreading with more than one processor the getInstance() method could be called at more or less the same time resulting in to more than one instance being created.
- Possible solutions:
  - Synchronize the getInstance() method
  - Do nothing if the getInstance() method is not critical to the application.
  - Move to an eagerly created instance rather than a lazily created one.

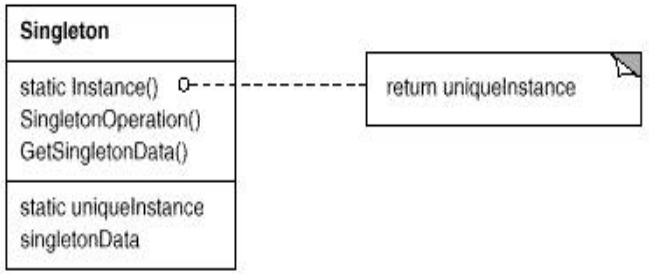
Use an Eagerly Created Instance Rather than a Lazy One

- Code:

```
//Data elements
public static Singleton uniqueInstance = new
   Singleton()

private Singleton() {}

public static Singleton getInstance() {
   return uniqueInstance
}
```

- Disadvantage – Memory may be allocated and not used.

Summary



- The singleton pattern ensures that there is just one instance of a class.
- The singleton pattern provides a global access point.
- The pattern is implemented by using a private constructor and a static method combined with a static variable.
- Possible problems with multithreading.
- Versions of Java earlier that 1.2 automatically clear singletons that are not being accessed as part of garbage collection.

Example: Remote Control

- Given remote control with seven programmable slots.
- A different device can be put into each slot.
- There is an On and Off switch for each device slot.
- Global Undo button undoes the last button pressed.
- Also given a CD with different vendor classes that have already been written (for the different devices, TV, Light, Sprinkler, etc)

What Varies? What stays the same?

- What Varies
  - The actual device assigned to a slot
  - The instruction for "On" on a specific device
  - The instruction for "Off" on a specific device
- What Stays the Same
  - Device with seven slots
  - Capability to assign a slot to a device
  - Capability to request that a device turn On or Off
  - Capability to undo the last action requested against the device

The Vendor Classes

- Vendor classes have been provided to us via a CD.
  - Ceiling Light
  - TV
  - Hottub
- We know that each device needs an "On" or "Off" state.
  - Since the capability to turn a device "On" or "Off" is something that "stays the same".
  - However, each vendor class has their own unique way of doing "On" or "Off".

One possible solution…

```
if (slot1 == Light)
    light.on();

Else if (slot1 == Hottub) {
    hottub.prepareJets();
    hottub.jetsOn();

} Else if (slot1 == TV)
   tv.on();

//…
```

Also…what about undo????

Problems:

The Remote needs to be aware of all the details about turning a device on (or off).

If device On/Off mechanism changes, the Remote code will need to be changed.

If a new device is added, this code would need to be changed.

Is this Open for Extension??

Separation of Concerns

- The Vendor Class
  - One (or more) methods that define "On"
  - One (or more) methods that define "Off"
- The Command
  - Sends a message to a device (On or Off)
  - Handle the undo of a message
  - Possible future enhancements
    - Logging request
    - Queue request
- The Remote – handles one or more Commands. The Remote doesn't know anything about the actual vendor class specifics.

The Command Pattern

- "Allows you to decouple the requestor of the action from the object that performs the action."
- "A Command object encapsulates a request to do something."
  - Note: A Command object handles a single request.
- The Command interface (in this example) just does one thing… executes a command.

LightOnCommand – A command

```
public class LightOnCommand implements Command {

   Light light;

   public LightOnCommand(Light light) {
        this.light = light;
   }

   public void execute() {
        light.on();
   }
}
```

The command is composed of a vendor class.

Constructor takes the vendor class as parameter.

Here the command delegates execution to the vendor class. Could be more steps.

SimpleRemoteControl – An invoker

```
public class SimpleRemoteControl {
   Command slot;

   public SimpleRemoteControl() {}

   public void setCommand(Command command) {
        slot = command;
   }

   public void buttonWasPressed() {
        slot.execute();
   }
}
```

This version of the remote just has one slot.

setCommand assigns a Command to a slot.

Tells the command to execute. Note that any command would work here. The remote doesn't know anything about the specific vendor class.

RemoteControlTest – A client

```
SimpleRemoteControl remote = new SimpleRemoteControl();

Light light = new Light();
GarageDoor garageDoor = new GarageDoor();

LightOnCommand lightOn =
            new LightOnCommand(light);
GarageDoorOpenCommand garageOpen =
          new GarageDoorOpenCommand(garageDoor);

remote.setCommand(lightOn);
remote.buttonWasPressed();

remote.setCommand(garageOpen);
remote.buttonWasPressed();
```

Create two vendor classes.

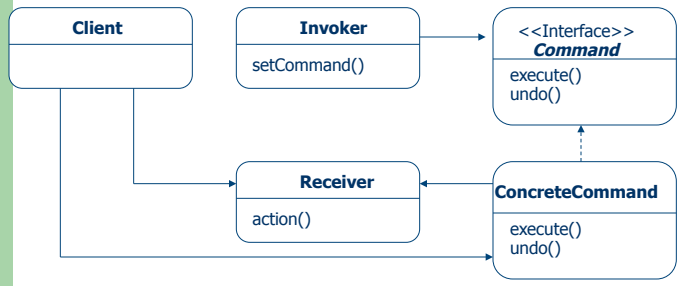
Create two commands based on these vendor classes.

Set the command and press button

The Command Pattern

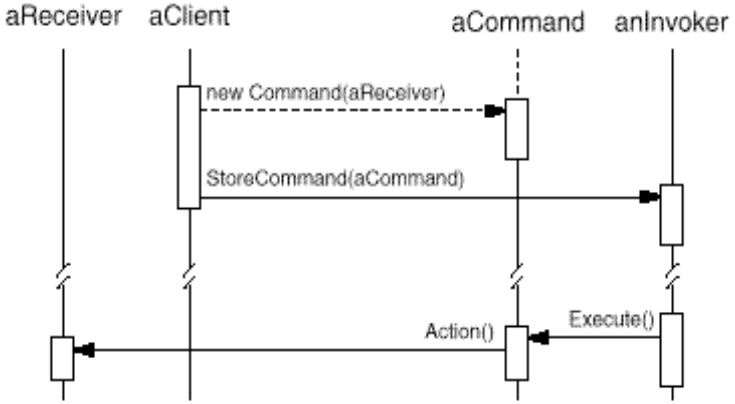
- GoF **Intent**: "Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations."
- **Participants:**
  - **Client** (RemoteControlTest) – creates command and associates command with receiver.
  - **Receiver** (TV, HotTub, ec)– knows how to perform the work.
  - **Concrete Command** (LightOnCommand) - implementation of Command interface
  - **Command Interface** – defines interface for all commands.
  - **Invoker** (Remote Control) – holds reference to a command and calls execute() method

Command Pattern Class Diagram

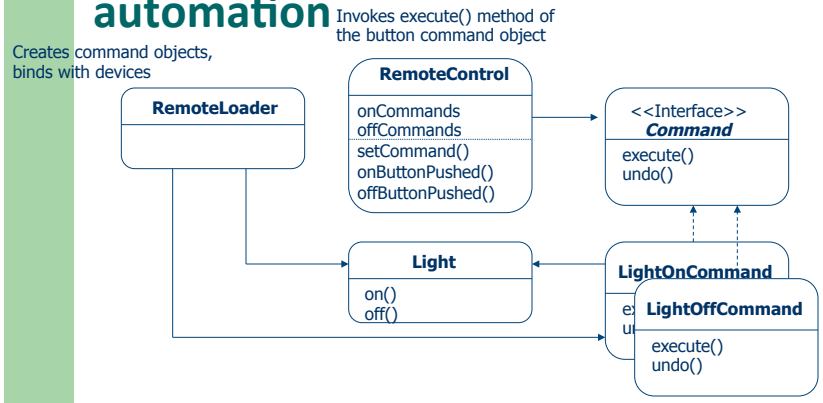


- **Client** creates a **ConcreteCommand** and binds it with a **Receiver.**
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.

The Sequence Diagram



Class Diagram for Home automation



Adding Undo

- Easy for some objects

```
public class LightOnCommand implements Command {
   Light light;

   public LightOnCOmmand(Light light) {
      this.light = light;
   }

   public void execute() {
      light.on();
   }

   public void undo() {
      light.off();
   }
}
```


Adding Undo

- Implementing the remote control with undo.

```
public class RemoteControl {
   Command[] onCommands;
   Command[] offCommands;
   Command[] undoCommands;

   public RemoteControl() {
      onCommands = new Command[7];
      offCommands = new Command[7];

      Command noCommand = new NoCommand();
      for (int i = 0; i < 7; i++) {
         onCommands[i] = noCommand;
         offCommands[i] = noCommand;
      }
      undoCommand = noCommand;
}
public void setCommand(int slot, Command onCommand, Command offCommand)
{
   onCommand[slot] = onCommand;
   offCommand[slot] = offCommand;
}
```


Adding Undo

```
public void onButtonWasPushed(int slot) {
   onCommands[slot].execute();
   undoCommand = onCommands[slot];

}

public void offButtonWasPushed(int slot) {
   offCommands[slot].execute();
   undoCommand = offCommands[slot];

}

public  void undoButtonwWasPushed() {
   undoCommand.undo();
}

// …
}
```

Adding Undo For Ceiling Fan

```
public class CeilingFan {
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW= 1;
    public static final int OFF= 0;
    public CeilingFan(String location) {
        this.location = location;
        speed = OFF;
    }
    public void high() {
        speed = HIGH;
        //code to set fan to high
    }
    public void medium() {
        speed = MEDIUM;
        //code to set fan to medium
    }
    // …
    public void off() {
        speed = OFF;
        //code to turn fan off
    }
```


Adding Undo For Ceiling Fan

```
public class CeilingFanHighCommand implements Command{
    CeilingFan ceilingFan;
    int prevSpeed;
    public CeilingFanHighCommand(CeilingFan ceilingFan{
        this.ceilingFan = ceilingFan;
    }
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }
    public void undo() {
    if (prevSpeed == CeilingFan.HIGH) {
        celingFan.high();
    } else if (prevSpeed = CeilingFan.MEDIUM) {
        ceilingFan.medium();
    } else if (prevSpeed = CeilingFan.LOW) {
        ceilingFan.low();
    } else if (prevSpeed = CeilingFan.OFF) {
        ceilingFan.off();}
}
```


History of Undo Operations

- If the Undo button is pressed multiple times, we want to undo each command that had been previously applied.
- Which object should we enhance to store a history of each command applied? Why?
  - Client (RemoteControlTest)
  - Receiver (TV, DVD, etc)
  - ConcreteCommand (LightOnCommand, LightOffCommand, etc)
  - Invoker (RemoteControl)
- We need to be able to add commands to a list and then later get the most recent one. What kind of object can we use?

History of Undo Operations

```
public class RemoteControl {
    Stack<Command> undoStack;  //this gets initialized in
                                //constructor.

    public void onButtonWasPushed(int slot) {
            onCommands[slot].execute();
            undoStack.push(onCommands[slot]);
    }

    public void offButtonWasPushed(int slot) {
            offCommands[slot].execute();
            undoStack.push(offCommands[slot]);
    }

    public void undoButtonWasPushed() {
            Command c = undoStack.pop();
            c.undo();
}
```

Summary so far

- **OO Basics**
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- **OO Principles**
  - **Encapsulate what varies**
  - **Favor composition over inheritance**
  - **Program to interfaces not to implementations** (Depend on abstracts. Do not depend on concrete classes)
  - **Classes should be open for extension but closed for modification** (Strive for loosely coupled designs between objects that interact.)