DESIGN PATTERNS

Eng. Hany Safwat

Design Patterns

- General repeatable **solution** to a commonly occurring problem in software design. A design pattern is **not a finished design** that can be transformed directly into code.
- It is a **description** or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show **relationships and interactions** between classes or objects, without specifying the final application classes or objects that are involved
- GOF : ***Design Patterns: Elements of Reusable Object-Oriented Software***

Design Patterns

- A Design pattern consists of various co-operating objects (classes, relationships etc)".
- They provide solutions for common design problems. More than anything else, they offer a consistent idiom for designers and programmers to speak about their design.
- A Design Pattern is not code. It is in fact an approach or a model that can be used to solve a problem.
- Design Patterns are about design and interaction of objects and they provide reusable solutions for solving common design problems.

A Must Have Books

Creational Patterns (5)

Object Creation mechanisms

- **Abstract Factory** : Creates an instance of several families of classes
- **Factory Method :** Define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Builder** : Separates object construction from its representation
- **Prototype:** A fully initialized instance to be copied or cloned
- **Singleton:** A class of which only a single instance can exist

Structural Patterns (7)

Relationships between Entities

- **Adapter** : Match interfaces of different classes
- **Bridge:** Separates an object's interface from its implementation
- **Composite:** A tree structure of simple and composite objects
- **Decorator:** Add responsibilities to objects dynamically
- **Façade:** A single class that represents an entire subsystem
- **Flyweight:** A fine-grained instance used for efficient sharing
- **Proxy:** An object representing another object

Behavioral Patterns (11)

Communication Patterns

- **Chain of Responsibility:** A way of passing a request between a chain of objects
- **Command** : Encapsulate a command request as an object
- **Interpreter:** A way to include language elements in a program
- **Iterator:** Sequentially access the elements of a collection
- **Mediator** : Defines simplified communication between classes
- **Memento:** Capture and restore an object's internal state
- **Observer:** A way of notifying change to a number of classes
- **State:** Alter an object's behavior when its state changes
- **Strategy:** Encapsulates an algorithm inside a class
- **Template Method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

Memento
Proxy
Saving State of Iteration
Builder
Adapter
Iterator
Bridge
Creating Composites
Avoiding Hysteresis
Adding Responsibilities to Objects
Enumerating Children
Composite
Composed Using
Command
Decorator
Sharing Composites
Adding Operations
Defining Traversals
Flyweight
Visitor
Changing Skin Versus Guts
Defining the Chain
Defining Grammer
Sharing Strategies
Adding Operations
Sharing Strates
Interpreter
Chain of Responsibility
Strategy
Sharing Terminal Symbols
Mediatar
Complex Dependency Management
State
Observer
Defining Algorithm's Steps
Template Method
Often Uses
Prototype
Configure Factory Dynamically
Factory Method
Implement Using
Abstract Factory
Single Instance
Single Instance
Facade
Singleton

Let The Patterns Start

It all Started with Ducks

But We Need The Ducks To Fly!!!

"he's an OO programmer... *how hard can it be?*"

It Really Does Fly!!!!

What Does Really Happened

Maintenance Maintenance Maintenance …

How About An Interface

Not Again !!!!

That is, like, the dumbest idea you've come up with. Can you say, "duplicate code"? If you thought having to override a few methods was bad, how are you gonna feel when you need to make a little change to the flying behavior... in all 48 of the flying Duck subclasses?!

I Have a Dream

The One Constant in Software Development is

Change

The Solution

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

Here's another way to think about this principle: ***take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.***

As simple as this concept is, it forms the basis for almost every design pattern. All patterns provide a way to let *some part of a system vary independently of all other parts.*

Separating what changes from what stays the same

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.

Design The Behavior

- Program to an interface not an implementation
- We want to *assign* a behavior to the instance of Duck
- We Should include a Behavior setter methods in the duck class so we can *change* the Duck Behavior through runtime
- We Will implement a set of classes whose entire reason for living is to represent a Behavior
- The Point is to exploit Polymorphism by programming to as super type so that the actual runtime object isn't locked into the code

Design Principle

Program to an interface, not an implementation.

Before and After DP

Before

- The behavior Came from a concrete implementation in the Super Class Duck or a specialized implementation in the Sub Class Itself

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

That way, the Duck classes won't need to know any of the implementation details for their own behaviors.

Interface Or Abstract Class

- Program to interface , Means Program to *Super Type*
- The Declared Type of the variables should be a *Super Type* usually an abstract class or an Interface so that the objects assigned to these variables can be of any concrete implementation of the super type
- The class Declaring the variable doesn't have to know about the actual object types

Programing to an Interface

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

Duck Behaviors

The Duck

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

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

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Class Diagram

Duck Class

```
public abstract class Duck
    {
        public IFlyBehavior FlyBehavior { get; set; }
        public IQuackBehavior QuackBehavior { get; set; }
        public abstract void Display();

        public void PerformFly()
        {             FlyBehavior.Fly();        }
        public void PerformQuack()
        {             QuackBehavior.Quack();        }
        public void Swim()
        {Console.WriteLine("All ducks float, even decoys!");}
    }
```

Mallard Duck

```
public class MallardDuck : Duck
    {
        public MallardDuck()
        {
            QuackBehavior = new LoudQuack();
            FlyBehavior = new FlyWithWings();
        }

        override public void Display()
        {
            Console.WriteLine("I'm a real Mallard duck");
        }
    }
```

HAS-A can be better than IS-A

Client makes use of an encapsulated family of algorithms for both flying and quacking.

Think of each set of behaviors as a family of algorithms.

These ~~behaviors~~ "algorithms" are interchangeable.

Design Principle

Favor composition over inheritance.

The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.

Strategy Pattern:

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

Open Closed Principle

- Software Entities (Classes , Modules , Functions , …) should be **Open** for **Extension** But **Closed** for **Modification .**
- **Open** for **Extension :** New Behavior can be added in the Future (In other words, we are able to change what the module does)
- **Closed** for **Modification :** Change to Source or Binary code are Not required Nor Permitted (No Compilation of Old Code is Required)

The Answer

- Rely on Abstraction
- C# Abstraction can be Achieved Via These Approaches
  - Abstract Classes
  - Interfaces
  - Procedural Programming Approach (Delegate & Lambda)

- If Client Class uses the Server Class ,and we want the Client to uses Different Server we need to change the Code .

Figure 9-1 Client is not open and closed

The Answer

- Strategy Pattern
- Depend on Abstraction ,
- Pass in Implementation

SOLID Principle

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

One last Word

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.

OO Principles

Encapsulate what varies.
Favor composition over inheritence.
Program to interfaces, not implementations.

We'll be taking a closer look at these down the road and also adding a few more to the list

OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Throughout the book think about how patterns rely on OO basics and principles.

One down, many to go!

Final word

- Patterns aren't *invented*, they are *discovered*.
- Most patterns and principles address issues of *change* in software.
- Most patterns allow some part of a system to vary independently of all other parts.
- We often try to take what varies in a system and encapsulate it.

the Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

What We Know

WeatherData

- getTemperature()
- getHumidity()
- getPressure()
- measurementsChanged()

// other methods

These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

The developers of the WeatherData object left us a clue about what we need to add...

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

First Approach

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

What is Wrong?

```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Area of change, we need to encapsulate this.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

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

Publisher Subscriber Model

The Observer Pattern defined: the class diagram

Loose Coupling

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

The Observer Pattern provides an object design where subjects and observers are loosely coupled.

The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Pros & Cons

- ***Abstract coupling between Subject and Observer***. 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 doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
- ***Support for broadcast communication***. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.
- ***Unexpected updates***. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocent operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

Class Diagram

Dot Net Observer Built in Pattern

- Delegates
- Events
- Event Arguments
- UI Controls

Last Word

OO Basics
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.
OO Patterns
Observer – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
A new pattern for communicating state to a set of objects in a loosely coupled manner. We

the Decorator Pattern

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality

Starbuzz Coffee

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The cost() method is abstract; subclassses need to define their own implementation.

The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.

Each subclass implements cost() to return the cost of the beverage.

Each cost method computes the cost of the coffee along with the other condiments in the order.

Another Solution

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

Things to Remember

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

Decorator VS. Extension Methods ?

4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

(You'll see how in a few pages.)

2 **Whip calls cost() on Mocha.**

1 **First, we call cost() on the outmost decorator, Whip.**

3 **Mocha calls cost() on DarkRoast.**

4 **DarkRoast returns its cost, 99 cents.**

5 **Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.**

5 **Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.**

Decorator

Okay, here's what we know so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Key Point!

Decorator

<<interface>>
IComponent
+Operation()
Client
Component
+Operation()
Decorator
–addedState
–component : IComponent
+Operation()
+AddedBehavior()
Calls the stored component's operation

Decorator

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorating our Beverage

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

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

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

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

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Decorated with Mocha

```
public class StarbuzzCoffee {

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

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

Last Word

We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

New means Concrete

```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible.

But we have to create an instance of a concrete class!

```
Duck duck;

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

We have a bunch of different duck classes, and we don't know until runtime which one we need to instantiate.

Pizza Factory

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

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

When you see "new", thi

```
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;
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings & cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

What Varies and What Stay

```
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("clam") {
            pizza = new ClamPizza();
        } else if (type.equals("veggie") {
            pizza = new eggiePizza();
        }

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

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Encapsulating Object Creation

```
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();
}
```

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

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

First we pull the object creation code out of the orderPizza Method

What's going to go here?

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to.

Creating The Factory

Now 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 create **method** on the factory object. No more concrete instantiations here!

The Simple Factory

- It is Not a REAL Design Pattern (YET)

Example

```
interface IProduct {
  string ShipFrom( );
}

class ProductA : IProduct {
  public String ShipFrom ( ) {
    return " from South Africa";
  }
}

class ProductB : IProduct {
  public String ShipFrom ( )
    return "from Spain";
  }
}
```

```
class Creator {
  public IProduct FactoryMethod(int month) {
    if (month >= 4 && month <=11)
        return new ProductA( );
    else
    if (month == 1 || month == 2 || month == 12)
        return new ProductB( );
    else return new DefaultProduct( );
  }
static void Main( ) {
  Creator c = new Creator( );
  IProduct product;

  for (int i=1; i<=12; i++) {
    product = c.FactoryMethod(i);
    Console.WriteLine("Avocados "+product.ShipFrom( ));
  }
}
```

Still Not A Pattern

Introducing the Factory Method Pattern

Factory Method Pattern

Product P = Creator.FactoryMethod( )

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Franchising the pizza store

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

Allowing the subclasses to decide

Each subclass overrides the createPizza() method, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.

If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.

Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
```

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

Creator

The Creator classes

Product

The Product classes
Factories produce products, and in the PizzaStore, our product is a Pizza.
Pizza
These are the concrete products – all the pizzas that are produced by our stores.
NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza
ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza

The Client

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("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-styled pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("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 flavored ones

Another perspective: parallel class hierarchies

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.

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.

The factory method is the key to encapsulating this knowledge.

UML Class Diagram

LAB Assignments D01

Game Project

- How you identify the entities in your game system,
- How you identify the design problems, and
- How you apply patterns to address your design specifications

First Phase

- First of all, you need to identify the objects you use in your game engine. For this, you should visualize how the end user is going to use the system. Let us assume that the end user is going to operate the game in the following sequence (let us keep things simple).
- Start the game
- Select two teams
- Add or remove players to/from a team
- Pick a play ground
- Start the game

Going On….

- Your system may have a number of PlayGrounds in it, a number of Teams etc. To list a few real world objects in the system, you have
- **Player** who play the soccer
- **Team** with various players in it
- **Ball** which is handled by various players.
- **PlayGround** where the match takes place.
- **Referee** in the ground to control the game.

- Also, you may need some logical objects in your game engine, like

- **Game** which defines a football game, which constitutes teams, ball, referee, playground etc
- **GameEngine** to simulate a number of games at a time.
- **TeamStrategy** to decide a team's strategy while playing

Class Diagram

Entities

- First of all, you have to write down a minimum description of your soccer engine, to identify the design problems. For example, here are few design problems related to some of the objects we identified earlier.
- **Ball**
  - When the position of a ball changes, all the players and the referee should be notified straight away.
- **Team and TeamStrategy**
  - When the game is in progress, the end user can change the strategy of his team (E.g., From Attack to Defend)
- **Player**
  - A player in a team should have additional responsibilities, like Forward, Defender etc, that can be assigned during the runtime.
- **PlayGround**
  - Each ground constitutes of gallery, ground surface, audience, etc - and each ground has a different appearance.

Problem

- ***Specific Design Problem:*** *"When the position of a ball changes, all the players and the referee should be notified straight away."*

- ***Problem Generalized:*** *"When a subject (in this case, the ball) changes, all its dependents (in this case, the players) are notified and updated automatically."*

Observer Pattern:

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

UML

Figure 9-9. Observer pattern UML diagram

Another UML

Subject
+Attach(in Observer)
+Detach(in Observer)
+Notify()
observer
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Observer Explained

- **Subject** This class provides an interface for attaching and detaching observers. Subject class also holds a private list of observers. Functions in Subject class are
  - ***Attach*** *- To add a new observer to the list of observers observing the subject*
  - ***Detach*** *- To remove an observer from the list of observers observing the subject*
  - ***Notify****- To notify each observer by calling the Update function in the observer, when a change occurs.*
-
- **ConcreteSubject** This class provides the state of interest to observers. It also sends a notification to all observers, by calling the Notify function in its super class (i.e, in the Subject class). Functions in ConcreteSubject class are
  - ***GetState*** *- Returns the state of the subject*

Observer Explained

- **Observer** This class defines an updating interface for all observers, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers
  - ***Update*** *- This function is an abstract function, and concrete observers will over ride this function*
-
- **ConcreteObserver** This class maintains a reference with the subject, to receive the state of the subject when a notification is received.
  - ***Update*** *- This is the overridden function in the concrete class. When this function is called by the subject, the ConcreteObserver calls the GetState function of the subject to update the information it have about the subject's state.*

Game UML

Subject
Observer
Ball
+AttachObserver(in obj : IObserver)
+DetachObserver(in obj : IObserver)
+NotifyObservers()
-observers
1
*
IObserver
+Update()
Position
-X : Integer
-Y : Integer
-Z : Integer
FootBall
-myPosition : Position
+GetBallPosition() : Position
+SetBallPosition(in p : Position)
1
1
-ball
Player
-ballPosition : Position
+Update()
Referee
-ballPosition : Position
+Update()
-ball
1
1
-
ConcreteSubject
ConcreteObserver

Problem

- ***Specific Design Problem:*** *"When the game is in progress, the end user can change the strategy of his team (E.g., From Attack to Defend)"*
- ***Problem Generalized: "****We need to let the algorithm (TeamStrategy) vary independently from clients (in this case, the Team) that use it."*

Strategy Pattern:

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

UML

Context
+ContextInterface()
strategy
Strategy
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Strategy Explained

Strategy

- This class is an abstract class for the algorithm (or strategy), from which all concrete algorithms are derived. In short, it provides an interface common to all the concrete algorithms (or concrete strategies). I.e, if there an abstract (must override) function called *foo()* in the Strategy class, all concrete strategy classes should override the *foo()* function.

ConcreteStrategy

- This class is where we actually implement our algorithm. In other words, it is the concrete implementation of the Strategy class. Just for an example, if *Sort* is the strategy class which implements the algorithm, then the concrete strategies can be *MergeSort*, *QuickSort* etc

Context

- This Context can be configured with one or more concrete strategy. It will access the concrete strategy object through the strategy interface.

UML

Context
Strategy
-has -strategy
Team
+SetStrategy(in s : TeamStrategy)
+PlayGame()
1
1
TeamStrategy
+Play()
AttackStrategy
+Play()
DefendStrategy
+Play()
ConcreteStrategy

Design Specification

- Now, let us address the design specifications related to the player. From our problem definition, it is clear that we need to assign responsibilities (like forward, defender etc) to each player during run time. At this point, you can think about sub classing (i.e, inheritance) - by creating a player class, and then inheriting classes like Forward, Defender etc from the base class. But the disadvantage is that, when you do sub classing, you cannot separate the responsibility of an object from its implementation.

- I.e, In our case, sub classing is not the suitable method, because we need to separate the responsibilities like 'Forward', 'Midfielder', 'Defender' etc from the Player implementation. Because, a player can be a 'Forward' one time, and some other time, the same player can be a 'Midfielder'.

Problem

- ***Specific Design Problem:*** *"A player in a team should have additional responsibilities, like Forward, Defender etc, that can be assigned during the runtime."*

- ***Problem Generalized: "****We need to attach additional responsibilities (like Forward, Midfielder etc) to the object (In this case, the Player) dynamically, with out using sub classing"*

Decorator Pattern:

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality

Decorator

- Used when you want to add functionality, you will be working with existing classes that cannot be changed.
- The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically. The object does not know it is being "decorated" which makes this a useful pattern for evolving systems.
- A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.

Decorator

<<interface>>
IComponent
+Operation()
Client
Component
+Operation()
Decorator
–addedState
–component : IComponent
+Operation()
+AddedBehavior()
Calls the stored component's operation

UML

Component
+Operation()
ConcreteComponent
+Operation()
Decorator
+Operation()
component
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();

Decorator

```
interface IComponent
   {    string Operation();     }

class Component :IComponent
   {
            public string Operation()  {……..}
   }

class Decorator01 : IComponent
   {
      IComponent component;
     public string AddedState ;
      public string Operation()                    {……….}
      public string AddedBehavior() {……….}
   }
```

Decorator

- The participants of the pattern are detailed below.

Component

- The Component class indicates an abstract interface for components. Later, we attach additional responsibilities to these components.

ConcreteComponent

- The ConcreteComponent class is the concrete implementation of the Component class. It actually defines an object to which additional responsibilities can be attached.

Decorator

- Decorator class is derived from Component class. That means, it inherits all the interfaces (functions, properties etc) of the component. It also keeps a reference to an object which is inherited from the component class. Hence, one concrete decorator can keep references to other concrete decorators as well (because Decorator class is inherited from the Component class).

Concrete Decorator

- This class is the actual place where we attach responsibilities to the component

Use the Decorator pattern when…

You have:

- An existing component class that may be unavailable for subclassing.

You want to:

- Attach additional state or behavior to an object dynamically.
- Make changes to some objects in a class without affecting others.
- Avoid subclassing because too many classes could result.

But consider using instead:

- The Adapter pattern, which sets up an interface between different classes.
- The Composite pattern, which aggregates an object without also inheriting its interface.
- The Proxy pattern, which specifically controls access to objects.
- The Strategy pattern, which changes the original object rather than wrapping it.

Game

Component
Player
+PassBall()
1
#player
Decorator
GoalKeeper
+PassBall()
FieldPlayer
+PassBall()
PlayerRole
+PassBall()
+AssignPlayer(in p : Player)
-refer
1
ConcreteComponent
Forward
+ShootGoal()
MidFielder
+Dribble()
Defender
+Defend()
ConcreteDecorator

Decorator Explained

- You can see that we have two concrete components, **GoalKeeper** and **FieldPlayer**, inherited from the Player class. We have three concrete decorators, **Forward**, **MidFielder**, and **Defender**. For a team, we may need 11 Field players and one goal keeper. Our design intend is, we need to assign responsibilities like Forward, Defender etc to the players during run time. We have only 11 field players - but it is possible that we can have 11 forwards and 11 midfielders at the same time, because a single player can be a forward and a midfielder at the same time. This will enable us to formulate good playing strategies - by assigning multiple roles to players, by swapping their roles etc.
- For example, you can ask a player to go forward and shoot a goal at some point of the match, by temporarily assigning him to a **Forward** decorator.
- To give additional responsibilities to a concrete component, first you create an object of the concrete component, and then you will assign it as the reference of a decorator. For example, you can create a field player and a Mid fielder decorator, and then you can assign the field player to the mid fielder decorator to add the responsibility of mid fielder to your player. Later, if you want, you can assign the same player to an object of a Forward decorator..

Design Requirement

- Hence, we need to construct the ground in such a way that, the construction of the ground can create different representations of the ground. I.e, a ground in Italy may have different gallery structure and surface when compared to a ground in England. But, the game engine may create both these grounds by calling the same set of functions

Problem:

- ***Specific Design Problem:*** *"Each ground constitutes of gallery, ground surface, audience, etc - and each ground has a different appearance."*
- ***Problem Generalized:*** *"We need to separate the construction of an object (ground) from its representation (the appearance of the ground) and we need to use the same construction process to create different representations."*

Builder Pattern:

- Separate the construction of a complex object from its representation so that the same construction process can create different representations

UML

Client
Director
+Construct() : Product
IBuilder
+BuildPart()
+GetResult() : Product
Invoke BuildPart
as required
Builder
+BuildPart()
+GetResult() : Product
Product
–parts : List

Builder

- The Builder pattern is based on Directors and Builders.
- Any number of Builder classes can conform to an IBuilder interface,
- and they can be called by a director to produce a product according to specification.
- The builders supply parts that the Product objects accumulate until the director is finished with the job.
- Suppose a Director wants two of one kind of part and one of another.
- It would request them from a Builder and pass them on to be added to the product's list.
- The point about the Builder pattern is that different builders can supply different (though conforming) parts.

Builder Explained

IBuilder

- An interface defining what can be built

Director

- A sequence of operations that can be followed to create a Product

Builder

- A class that is called by a Director to make Product parts

Product

- The object under construction, by parts

Builder Class Diagram