



Applying Design Patterns – For Everyone

Introducing patterns to you in a simple, human readable, and funny (?) way, by discussing the thought process behind applying design patterns

Anoop Madhusudanan – <http://amazedsaint.blogspot.com>

Contents

Part I	4
Introduction.....	4
Introduction To This Article	4
An Overview of Design Patterns	4
Architecting Your (Simple) Football Engine	5
Identifying Entities.....	5
Identifying Design Problems	6
Identifying Patterns To Use	7
1: Addressing the design problems related with the 'Ball'	7
2: Addressing the design problems related with 'Team' And 'TeamStrategy'	7
3: Addressing the design problems related with 'Player'	8
4: Addressing the design problems related with 'PlayGround'	8
Part II	9
Applying Observer Pattern	9
Understanding the Observer Pattern	9
Adapting the Observer Pattern.....	11
Ball (Subject).....	12
FootBall (ConcreteSubject)	12
IObserver (Observer)	13
Player (ConcreteObserver)	13
Referee (ConcreteObserver).....	14
Position Class	15
Putting It All Together.....	15
Running the project	16
Classification	17
Part III	18
Applying Strategy Pattern.....	18
Understanding the Strategy Pattern.....	18
Adapting the Strategy Pattern	19

Strategy Pattern Implementation.....	20
TeamStrategy (Strategy)	20
AttackStrategy (ConcreteStrategy)	20
DefendStrategy (ConcreteStrategy).....	20
Team (Context)	21
Putting It All Together.....	21
Running The Project.....	22
Part IV	23
Applying Decorator Pattern.....	23
Understanding Decorator Pattern	23
Adapting The Decorator Pattern.....	24
Decorator Pattern Implementation	26
Player (Component).....	26
FieldPlayer (ConcreteComponent)	26
GoalKeeper (ConcreteComponent)	26
PlayerRole (Decorator)	27
Forward (ConcreteDecorator)	27
MidFielder (ConcreteDecorator)	28
Defender (ConcreteDecorator).....	28
Putting It All Together.....	28
Running The Project.....	30

Part I

Solution Architect: "But you can use patterns"

Dumb Developer: "Yes, But can I get it as an ActiveX control?"

Introduction

Introduction To This Article

This article is expected to

- Introduce patterns to you in a simple, human readable (?) way
- Train you how to really 'Apply' patterns (you can learn patterns easily, but to apply them to solve a problem, you need real design skills)
- Provide you a fair idea regarding the contexts for applying the following patterns - Builder, Observer, Strategy and Decorator (well, they are few popular design patterns)
- Demonstrate you how to apply the Observer pattern, to solve a design problem

In this entire article, you will go through the following steps

1. You will model a very simple football game engine
2. You will identify the design problems in your football game engine
3. You will decide which patterns to use for solving your design problems
4. You will then actually use the observer pattern, to solve one of your design problem.

As a prerequisite

- You may need to get some grip on reading and understanding UML diagrams

Using The Code

- The related zip file includes the code, UML designs (in Visio format) etc. After reading this article, you may download and extract the zip file - using a program like Winzip - to play with the source code.

An Overview of Design Patterns

Even without much knowledge about design patterns, designers and developers tend to reuse class relationships and object collaborations to simplify the design process. In short, "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. For example, you can tell a friend that you used a 'Builder' pattern for addressing some design specifications in your project.

A consistent classification of patterns for common design problems are provided by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [also known as the Gang of Four (GOF)]. The Gang of Four (GOF) patterns are generally considered the foundation for all other patterns.

The basic principle of using patterns is reusability. Once a problem is address some way, you are not really expected to re-invent the wheel if you properly understand the concept of pattern centric software engineering. Here are some important points to remember about design patterns.

- 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 Design Pattern is normally represented with the help of a UML diagram.

Some real hands on experience with patterns may provide you a better idea!!

Architecting Your (Simple) Football Engine

You are working with a popular computer game developing company, and they made you the Solution Architect of one of their major projects - a Soccer (Football) Game Engine (Nice, huh?). Now, you are leading the process of designing the entire Football game engine, and suddenly you have a lot of design considerations, straight away. Let us see

- 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.

Identifying Entities

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

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

So, here is a very abstract view of the system. The boxes represent classes in your system, and the connectors depicts 'has' relationships and their multiplicity. The arrow head represents the direction of reading. I.e, a GameEngine has (can simulate) Games. A Game has (consists of) three referees, one ball, two teams, and one ground. A team can have multiple players, and one strategy at a time.

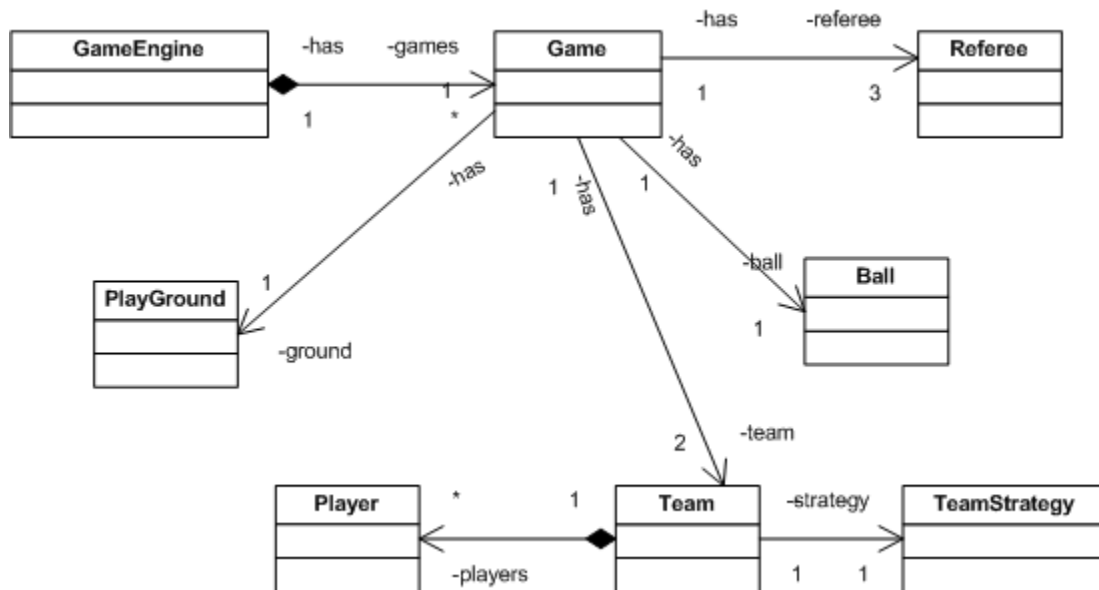


Fig 1 - High level view

Identifying Design Problems

Now, you should decide

- How these objects are structured
- How they are created
- Their behavior when they interact each other, to formulate the design specifications.

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.

So now, let us see how to identify the patterns, to address these design problems.

Identifying Patterns To Use

Have a look at the design problems you identified above (yes, do it once more). Now, let us see how to address these problems using design patterns.

1: Addressing the design problems related with the 'Ball'

First of all, take the specifications related to the ball. You need to design a framework such that when the state (position) of the ball is changed, all the players and the referee are notified regarding the new state (position) of the ball. Now, let us generalize the 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."*

Once you have such a design problem, you refer the GOF patterns - and suddenly you may find out that you can apply the 'Observer' pattern to solve the problem.



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.

In this case, we used this pattern because we need to notify all the players, when the position of the ball is changed.

2: Addressing the design problems related with 'Team' And 'TeamStrategy'

Next, we have to address the specifications related to the team and team strategy. As we discussed earlier, when the game is in progress, the end user can change the strategy of his team (E.g., From Attack to Defend). This clearly means that we need to separate the Team's Strategy from the Team that uses it.

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."*

Then, you can choose the 'Strategy' pattern to address the above design problem.



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.

3: Addressing the design problems related with 'Player'

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'.

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, without using sub classing"*

Then, you can choose the 'Decorator' pattern to address the above design problem.



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

4: Addressing the design problems related with 'PlayGround'

If you take a look at the specifications of Ground, we see that a ground's appearance is decided by various sub units like gallery, surface of the ground, audience etc. The appearance of the ground may vary, according to these sub units. 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.

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.

Now, you can chose the 'Builder' pattern to address the above design problem.



Part II

Solution Architect: "I asked you to learn about patterns"

Dumb Developer: "Yes, now I can develop a football engine using patterns"

Solution Architect: "Huh? What do you mean? !@@#!"

Applying Observer Pattern

In this section, we will have a closer look at the observer pattern, and then we will apply the pattern to solve our first design problem. If you can remember, our first design problem was,

- "When the position of a ball changes, all the players should be notified straight away."

Understanding the Observer Pattern

The UML class diagram of the observer pattern is shown below.

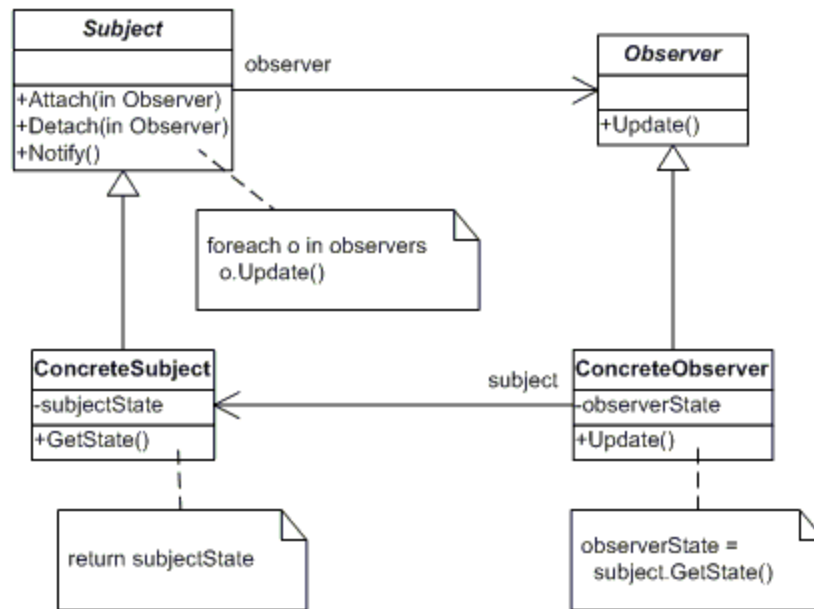


Fig 2 - Observer Pattern

The participants of the pattern are detailed below.

- **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**

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.

Adapting the Observer Pattern

Now, let us see how this pattern can be adapted to solve our specific problem. This will give you a better idea.

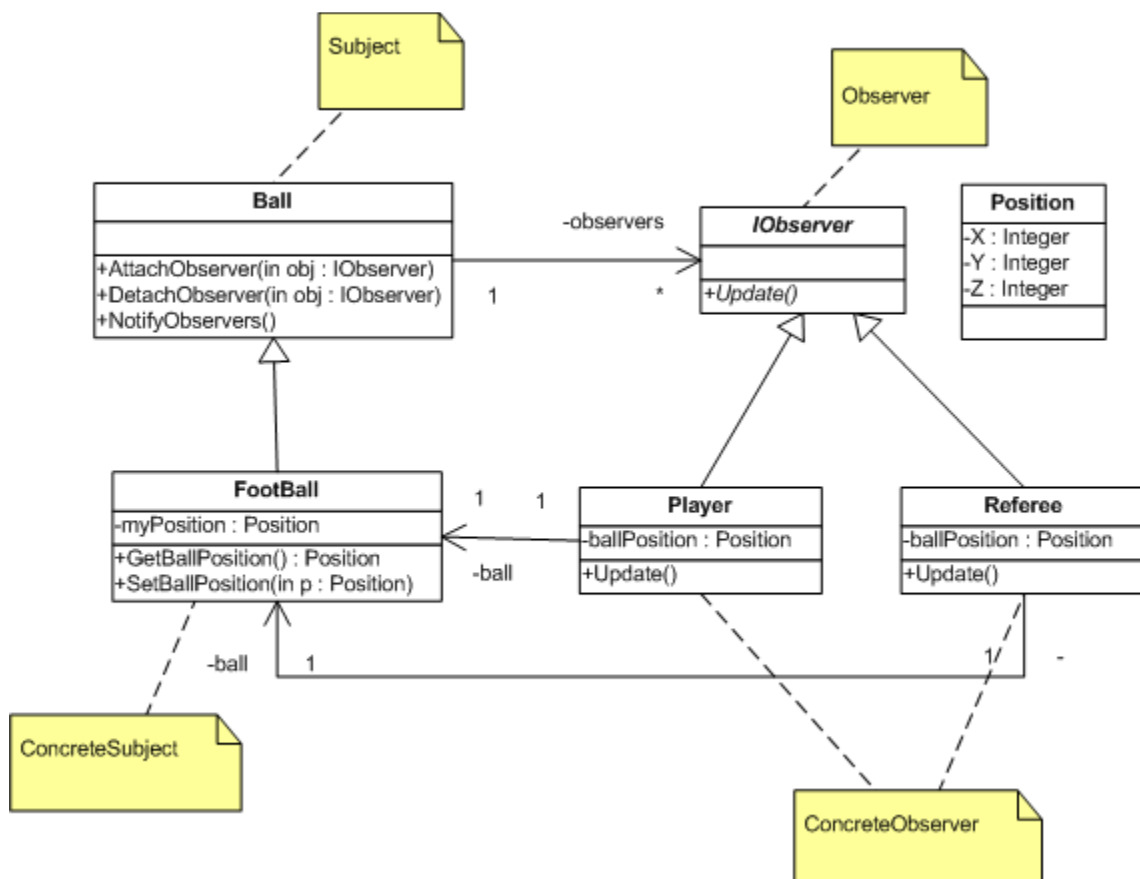


Fig 3 - Solving Our First Design Problem

When we call the **SetBallPosition** function of the ball to set the new position, it inturn calls the **Notify** function defined in the Ball class. The Notify function iterates all observers in the list, and invokes the **Update** function in each of them. When the Update function is invoked, the observers will obtain the new state position of the ball, by calling the **GetBallPosition** function in the Foot ball class.

The participants are detailed below.

Ball (Subject)

The implementation of Ball class is shown below.

```
' Subject : The Ball Class

Public Class Ball

'A private list of observers

Private observers As new System.Collections.ArrayList

'Routine to attach an observer

Public Sub AttachObserver(ByVal obj As IObserver)
observers.Add(obj)
End Sub

'Routine to remove an observer

Public Sub DetachObserver(ByVal obj As IObserver)
observers.Remove(obj)
End Sub

'Routine to notify all observers

Public Sub NotifyObservers()
Dim o As IObserver
For Each o In observers
o.Update()
Next
End Sub

End Class ' END CLASS DEFINITION Ball
```

Football (ConcreteSubject)

The implementation of FootBall class is shown below.

```
' ConcreteSubject : The FootBall Class

Public Class FootBall
Inherits Ball
```

```

'State: The position of the ball

Private myPosition As Position

'This function will be called by observers to get current position

Public Function GetBallPosition() As Position
Return myPosition
End Function

'Some external client will call this to set the ball's position

Public Function SetBallPosition(ByVal p As Position)
myPosition = p
'Once the position is updated, we have to notify observers

NotifyObservers()
End Function

'Remarks: This can also be implemented as a get/set property

End Class ' END CLASS DEFINITION FootBall

```

IObserver (Observer)

The implementation of `IObserver` class is shown below. This class provides interface specifications for creating Concrete Observers.

```

' Observer: The IObserver Class

'This class is an abstract (MustInherit) class

Public MustInherit Class IObserver

'This method is a mustoverride method

Public MustOverride Sub Update()

End Class ' END CLASS DEFINITION IObserver

```

Player (ConcreteObserver)

The implementation of Player class is shown below. Player is inherited from `IObserver` class

```

' ConcreteObserver: The Player Class

'Player inherits from IObserver, and overrides Update method

Public Class Player
Inherits IObserver

```

```

'This variable holds the current state(position) of the ball

Private ballPosition As Position

'A variable to store the name of the player

Private myName As String

'This is a pointer to the ball in the system

Private ball As FootBall

'Update() is called from Notify function, in Ball class

Public Overrides Sub Update ()
ballPosition = ball.GetBallPosition()
System.Console.WriteLine("Player {0} say that the ball is at {1},{2},{3} ", _
    myName, ballPosition.X, ballPosition.Y, ballPosition.Z)
End Sub

'A constructor which allows creating a reference to a ball

Public Sub New(ByRef b As FootBall, ByVal playerName As String)
ball = b
myName = playerName
End Sub

End Class ' END CLASS DEFINITION Player

```

Referee (ConcreteObserver)

The implementation of Referee class is shown below. Referee is also inherited from `IObserver` class

```

' ConcreteObserver : The Referee Clas

Public Class Referee
Inherits IObserver

'This variable holds the current state(position) of the ball

Private ballPosition As Position

'This is a pointer to the ball in the system

Private ball As FootBall

'A variable to store the name of the referee

Private myName As String

'Update() is called from Notify function in Ball class

Public Overrides Sub Update()
ballPosition = ball.GetBallPosition()
System.Console.WriteLine("Referee {0} say that the ball is at {1},{2},{3} ", _
    myName, ballPosition.X, ballPosition.Y, ballPosition.Z)
End Sub

'A constructor which allows creating a reference to a ball

```

```
Public Sub New(ByRef b As Football, ByVal refereeName As String)
myName = refereeName
ball = b
End Sub

End Class ' END CLASS DEFINITION Referee
```

Position Class

Also, we have a position class, to hold the position of the ball.

```
'Position: This is a data structure to hold the position of the ball

Public Class Position

Public X As Integer
Public Y As Integer
Public Z As Integer

'This is the constructor

Public Sub New(Optional ByVal x As Integer = 0, _
Optional ByVal y As Integer = 0, _
Optional ByVal z As Integer = 0)

Me.X = x
Me.Y = y
Me.Z = z
End Sub

End Class ' END CLASS DEFINITION Position
```

Putting It All Together

Now, let us create a ball and few observers. We will also attach these observers to the ball, so that they are notified automatically when the position of the ball changes. The code is pretty self explanatory.

```
'Let us create a ball and few observers

Public Class GameEngine

Public Shared Sub Main()

'Create our ball (i.e, the ConcreteSubject)

Dim ball As New Football()

'Create few players (i.e, ConcreteObservers)

Dim Owen As New Player(ball, "Owen")
Dim Ronaldo As New Player(ball, "Ronaldo")
Dim Rivaldo As New Player(ball, "Rivaldo")
```

```

'Create few referees (i.e, ConcreteObservers)

Dim Mike As New Referee(ball, "Mike")
Dim John As New Referee(ball, "John")

'Attach the observers with the ball

ball.AttachObserver(Owen)
ball.AttachObserver(Ronaldo)
ball.AttachObserver(Rivaldo)
ball.AttachObserver(Mike)
ball.AttachObserver(John)

System.Console.WriteLine("After attaching the observers...")
'Update the position of the ball.

'At this point, all the observers should be notified automatically
ball.SetBallPosition(New Position())

'Just write a blank line
System.Console.WriteLine()

'Remove some observers

ball.DetachObserver(Owen)
ball.DetachObserver(John)

System.Console.WriteLine("After detaching Owen and John...")

'Updating the position of ball again

'At this point, all the observers should be notified automatically
ball.SetBallPosition(New Position(10, 10, 30))

'Press any key to continue..
System.Console.Read()

End Sub

End Class

```

Running the project

After running the project, you'll get the output as


```

E:\My Publications\Published Articles\Codeproject Submissions\Popular Pat...
After attaching the observers...
Player Owen say that the ball is at 0,0,0
Player Ronaldo say that the ball is at 0,0,0
Player Rivaldo say that the ball is at 0,0,0
Referee Mike say that the ball is at 0,0,0
Referee John say that the ball is at 0,0,0

After detaching Owen and John...
Player Ronaldo say that the ball is at 10,10,30
Player Rivaldo say that the ball is at 10,10,30
Referee Mike say that the ball is at 10,10,30

```

Side Note - Classification

Patterns can be classified

- With respect to purpose.
- With respect to scope.

With respect to purpose, patterns are classified to Creational, Structural and Behavioral. For example,

- The Observer pattern we just learned is a behavioral pattern (because it help us model the behavior and interactions of objects)
- The Builder pattern is a creational pattern (because it details how an object can be created in a particular way) and so on.

Here is the complete classification diagram.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	<ul style="list-style-type: none"> • Factory Method 	<ul style="list-style-type: none"> • Adapter (class) 	<ul style="list-style-type: none"> • Interpreter • Template Method
	Object	<ul style="list-style-type: none"> • Abstract Factory • Builder • Prototype • Singleton 	<ul style="list-style-type: none"> • Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy 	<ul style="list-style-type: none"> • Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor



Part III

Applying Strategy Pattern

In this section, we will have a closer look at the strategy pattern, and then we will apply the pattern to solve our second design problem. Refer the previous article at this point - just to remind yourself regarding our second design problem..

If you can remember, our second design problem was,

- **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."

As we discussed earlier, when the game is in progress, we need to change the strategy of the team (E.g., From Attack to Defend). This clearly means that we need to separate the Team's Strategy from the Team that uses it.

As we know, we can apply strategy pattern to solve the above design problem, because it lets the algorithm (i.e, the Team's strategy) vary independently from clients (i.e, the Team) that use it. Let us see how we can apply Strategy pattern to solve this design problem.

Understanding the Strategy Pattern

Strategy pattern is pretty simple. The UML diagram of Strategy Pattern is shown below.

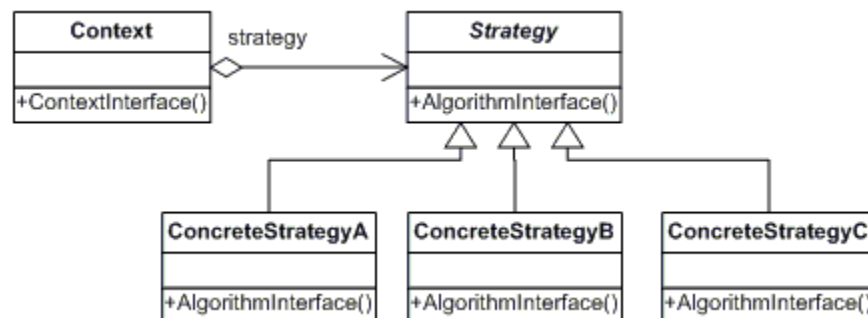


Fig - Strategy Pattern

The participants of the pattern are detailed below.

- **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.

Adapting the Strategy Pattern

Now, let us see how we actually adapt the Strategy pattern, to solve our problem. This will give you a very clear picture.

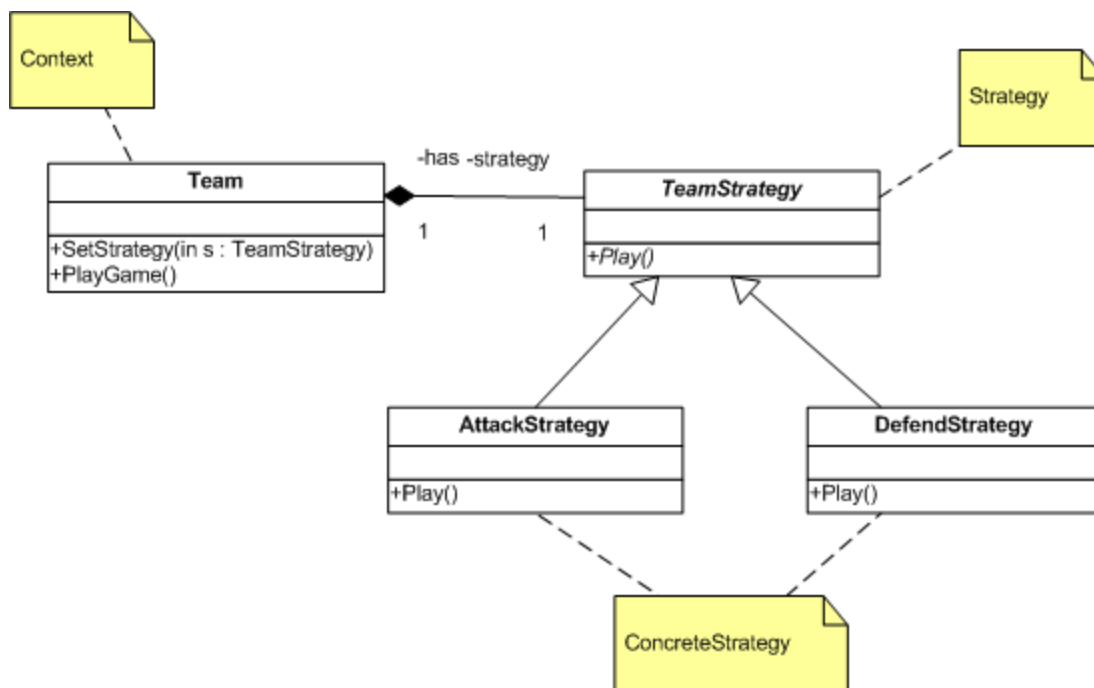


Fig - Solving Our Second Design Problem

Here, the **TeamStrategy** class holds the *Play* function. **AttackStrategy** and **DefendStrategy** are the concrete implementations of the **TeamStrategy** class. The **Team** holds a strategy, and this strategy can be changed according to the situation of the match (for example, we change the active strategy from **AttackStrategy** to **DefendStrategy**, if we lead by a number of goals - huh, well, I'm not a good football coach anyway). When we call *PlayGame* function in the **Team**, it calls the *Play* function of the current strategy. Kindly have a look at the code. It is straight forward, and everything is commented neatly.

By using strategy pattern, we separated the algorithm (i.e, the strategy of the team) from the Team class.

Strategy Pattern Implementation

TeamStrategy (Strategy)

The code for TeamStrategy class is shown below.

```
'Strategy: The TeamStrategy class
'
'This class provides an abstract interface
'to implement concrete strategy algorithms

Public MustInherit Class TeamStrategy

'AlgorithmInterface : This is the interface provided
Public MustOverride Sub Play ()

End Class ' END CLASS DEFINITION TeamStrategy
```

AttackStrategy (ConcreteStrategy)

The code for AttackStrategy class is shown below. It is derived from TeamStrategy

```
'ConcreteStrategy: The AttackStrategy class
'
'This class is a concrete implementation of the
'strategy class.

Public Class AttackStrategy
Inherits TeamStrategy

'Overrides the Play function.
'Let us play some attacking game

Public Overrides Sub Play()
'Algorithm to attack
System.Console.WriteLine(" Playing in attacking mode")
End Sub

End Class ' END CLASS DEFINITION AttackStrategy
```

DefendStrategy (ConcreteStrategy)

The code for DefendStrategy class is shown below. It is derived from TeamStrategy

```
'ConcreteStrategy: The DefendStrategy class
'
'This class is a concrete implementation of the
'strategy class.

Public Class DefendStrategy
Inherits TeamStrategy
```

```

'Overrides the Play function.
'Let us go defensive
Public Overrides Sub Play()
'Algorithm to defend
System.Console.WriteLine(" Playing in defensive mode")
End Sub

End Class ' END CLASS DEFINITION DefendStrategy

```

Team (Context)

The code for Team class is shown below. A team can have one strategy at a time, according to our design.

```

'Context: The Team class
'This class encapsulates the algorithm

Public Class Team

'Just a variable to keep the name of team
Private teamName As String

'A reference to the strategy algorithm to use
Private strategy As TeamStrategy

'ContextInterface to set the strategy
Public Sub SetStrategy(ByVal s As TeamStrategy)
'Set the strategy
strategy = s
End Sub

'Function to play
Public Sub PlayGame()
'Print the team's name
System.Console.WriteLine(teamName)
'Play according to the strategy
strategy.Play()
End Sub

'Constructor to create this class, by passing the team's
'name

Public Sub New(ByVal teamName As String)
'Set the team name to use later
Me.teamName = teamName
End Sub

End Class ' END CLASS DEFINITION Team

```

Putting It All Together

This is the GameEngine class to create teams, to set their strategies, and to make them play the game. The code is pretty simple and commented heavily.

```

'GameEngine class for demonstration

```

```

Public Class GameEngine

Public Shared Sub Main()

'Let us create a team and set its strategy,
'and make the teams play the game

'Create few strategies
Dim attack As New AttackStrategy()
Dim defend As New DefendStrategy()

'Create our teams
Dim france As New Team("France")
Dim italy As New Team("Italy")

System.Console.WriteLine("Setting the strategies..")

'Now let us set the strategies
france.SetStrategy(attack)
italy.SetStrategy(defend)

'Make the teams start the play
france.PlayGame()
italy.PlayGame()

System.Console.WriteLine()
System.Console.WriteLine("Changing the strategies..")

'Let us change the strategies
france.SetStrategy(defend)
italy.SetStrategy(attack)

'Make them play again
france.PlayGame()
italy.PlayGame()

'Wait for a key press
System.Console.Read()

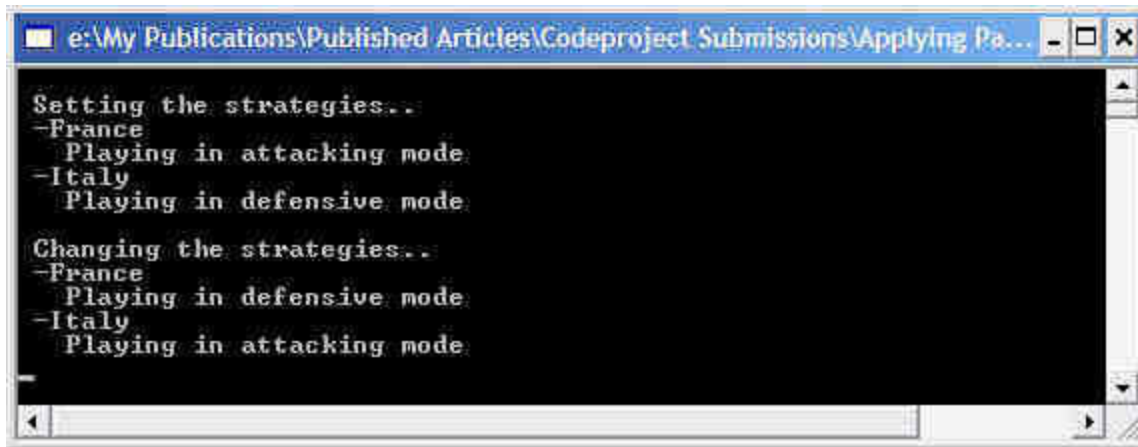
End Sub

End Class

```

Running The Project

Execute the project and you'll get the following output.



```
e:\My Publications\Published Articles\Codeproject Submissions\Applying Pa...
Setting the strategies..
-France
  Playing in attacking mode
-Italy
  Playing in defensive mode
Changing the strategies..
-France
  Playing in defensive mode
-Italy
  Playing in attacking mode
```



Part IV

Applying Decorator Pattern

In this section, we will see how to apply the Decorator pattern to solve our third design problem (Just refer the previous article if required). Our third design problem was related to assigning responsibilities (like Forward, Midfielder etc) to a player at runtime.

You can think about creating a player class, and then deriving sub classes like Forward, Midfielder, Defender etc. But it is not the best solution, because as we discussed earlier - a player can be a forward at one time, and at some other time, the same player can be a mid fielder. At least, it will be so in our soccer engine. (any football experts around? ;)) . So, these were our design problems.

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"

Understanding Decorator Pattern

Decorator pattern can be used to add responsibilities to objects dynamically. They also provide an excellent alternative to sub classing. The UML diagram of Decorator pattern is shown below.

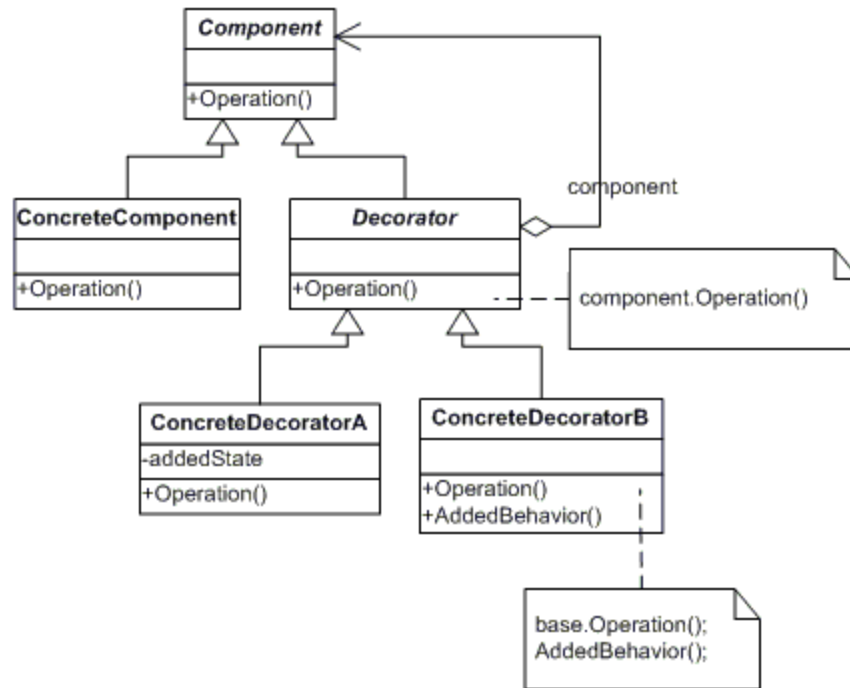


Fig - Decorator Pattern

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.

Adapting The Decorator Pattern

Now, it is time to adapt the Decorator pattern to solve our design problem related to the player.

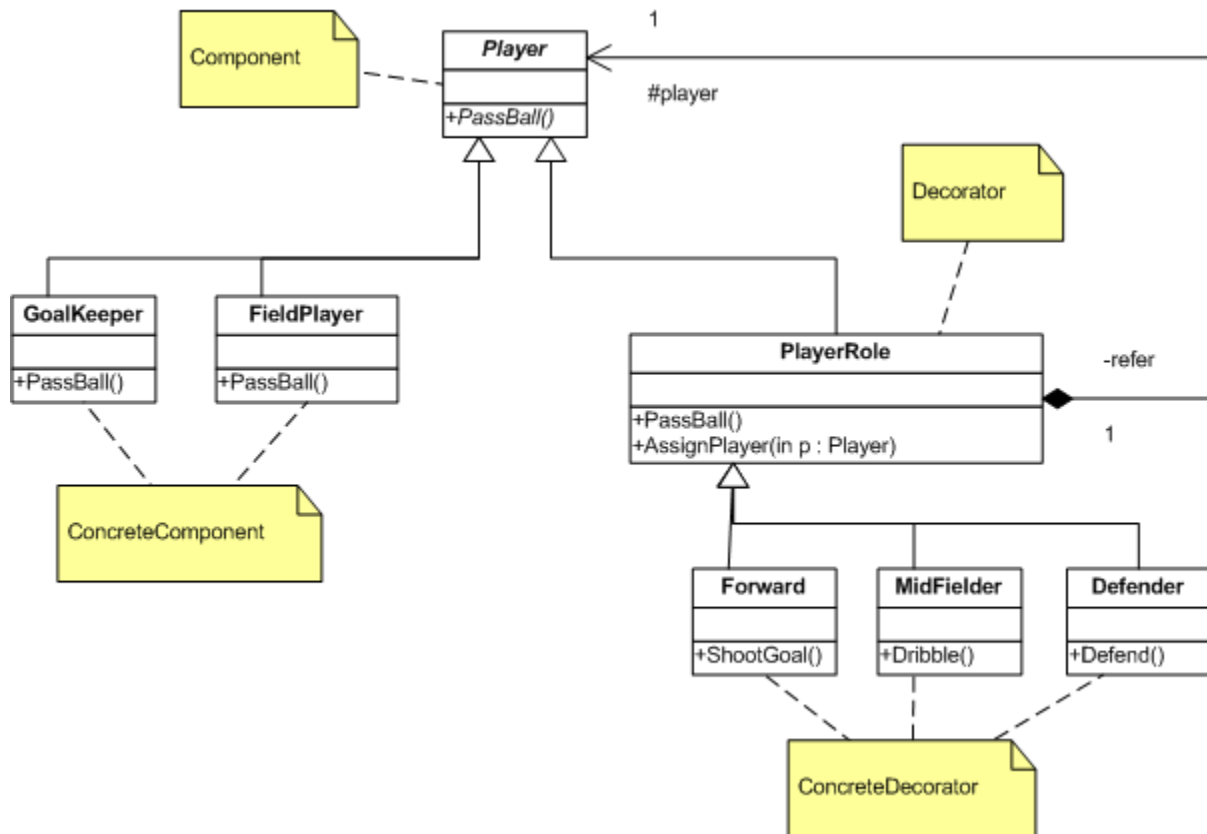


Fig - Solving Our Third Design Problem

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. This is very well explained in the **GameEngine** module of the Decorator pattern sample code.

See the implementation below. It is heavily commented.

Decorator Pattern Implementation

Player (Component)

The implementation of Player class is shown below

```
' Component: The Player class

Public MustInherit Class Player

'Just give a name for this player
Private myName As String

'The property to get/set the name
Public Property Name() As String
Get
Return myName
End Get
Set(ByVal Value As String)
myName = Value
End Set
End Property

'This is the Operation in the component
'and this will be overridden by concrete components
Public MustOverride Sub PassBall()

End Class ' END CLASS DEFINITION Player
```

FieldPlayer (ConcreteComponent)

The implementation of FieldPlayer class is shown below

```
' ConcreteComponent : Field Player class

'This is a concrete component. Later, we will add additional responsibilities
'like Forward, Defender etc to a field player.

Public Class FieldPlayer
Inherits Player

'Operation: Overrides PassBall operation
Public Overrides Sub PassBall ()
System.Console.WriteLine(" Fieldplayer ({0}) - passed the ball", _
MyBase.Name)
End Sub

'A constructor to accept the name of the player
Public Sub New(ByVal playerName As String)
MyBase.Name = playerName
End Sub

End Class ' END CLASS DEFINITION FieldPlayer
```

GoalKeeper (ConcreteComponent)

The implementation of GoalKeeper class is shown below

```
' ConcreteComponent : GaolKeeper class

'This is a concrete component. Later, we can add additional responsibilities
'to this class if required.

Public Class GoalKeeper
Inherits Player

'Operation: Overriding the base class operation
Public Overrides Sub PassBall ()
System.Console.WriteLine(" GoalKeeper ({0}) - passed the ball", MyBase.Name)
End Sub

'A constructor to accept the name of the player
Public Sub New(ByVal playerName As String)
MyBase.Name = playerName
End Sub

End Class ' END CLASS DEFINITION GoalKeeper
```

PlayerRole (Decorator)

The implementation of PlayerRole class is shown below

```
'Decorator: PlayerRole is the decorator

Public Class PlayerRole
Inherits player

'The reference to the player
Protected player As player

'Call the base component's function
Public Overrides Sub PassBall()
player.PassBall()
End Sub

'This function is used to assign a player to this role
Public Sub AssignPlayer(ByVal p As player)
'Keep a reference to the player, to whom this
'role is given
player = p
End Sub

End Class ' END CLASS DEFINITION PlayerRole
```

Forward (ConcreteDecorator)

The implementation of Forward class is shown below

```
'ConcreteDecorator: Forward class is a Concrete implementation
'of the PlayerRole (Decorator) class

Public Class Forward
Inherits PlayerRole
```

```

'Added Behavior: This is a responsibility exclusively for the Forward
Public Sub ShootGoal()
System.Console.WriteLine(" Forward ({0}) - Shot the ball to goalpost", _
MyBase.player.Name)

End Sub

End Class ' END CLASS DEFINITION Forward

```

MidFielder (ConcreteDecorator)

The implementation of MidFielder class is shown below

```

'ConcreteDecorator: MidFielder class is a Concrete implementation
'of the PlayerRole (Decorator) class

Public Class MidFielder
Inherits PlayerRole

'AddedBehavior: This is a responsibility exclusively for the Midfielder
'(Don't ask me whether only mid fielders can dribble the ball - atleast
'it is so in our engine)

Public Sub Dribble()
System.Console.WriteLine(" Midfielder ({0}) - dribbled the ball", _
MyBase.player.Name)
End Sub

End Class ' END CLASS DEFINITION Midfielder

```

Defender (ConcreteDecorator)

The implementation of Defender class is shown below

```

'ConcreteDecorator: Defender class is a Concrete implementation
'of the PlayerRole (Decorator) class

Public Class Defender
Inherits PlayerRole

'Added Behavior: This is a responsibility exclusively for the Defender
Public Sub Defend()
System.Console.WriteLine(" Defender ({0}) - defended the ball", _
MyBase.player.Name)
End Sub

End Class ' END CLASS DEFINITION Defender

```

Putting It All Together

```

'Let us put it together
Public Class GameEngine

Public Shared Sub Main()

'-- Step 1:
'Create few players (concrete components)

```

```

'Create few field Players
Dim owen As New FieldPlayer("Owen")
Dim beck As New FieldPlayer("Beckham")

'Create a goal keeper
Dim khan As New GoalKeeper("Khan")

'-- Step 2:
'Just make them pass the ball
'(during a warm up session ;))

System.Console.WriteLine()
System.Console.WriteLine(" > Warm up Session... ")

owen.PassBall()
beck.PassBall()
khan.PassBall()

'-- Step 3: Create and assign the responsibilities
'(when the match starts)

System.Console.WriteLine()
System.Console.WriteLine(" > Match is starting.. ")

'Set owen as our first forward
Dim forward1 As New Forward()
forward1.AssignPlayer(owen)

'Set Beckham as our midfielder
Dim midfielder1 As New MidFielder()
midfielder1.AssignPlayer(beck)

'Now, use these players to do actions
'specific to their roles

'Owen can pass the ball
forward1.PassBall()
'And owen can shoot as well
forward1.ShootGoal()

'Beckham can pass ball
midfielder1.PassBall()
'Beckham can dribble too
midfielder1.Dribble()

' [ Arrange the above operations to some meaningful sequence, like
' "Beckham dribbled and passed the ball to owen and owen shooted the
' goal ;) - just for some fun ]"

'-- Step 4: Now, changing responsibilities
'(during a substitution)

'Assume that owen got injured, and we need a new player
'to play as our forward1

System.Console.WriteLine()
System.Console.WriteLine(" > OOps, Owen got injured. " & _
"Jerrard replaced Owen.. ")

'Create a new player
Dim jerrard As New FieldPlayer("Jerrard")

```

```

'Ask Jerrard to play in position of owen
forward1.AssignPlayer(jerrard)
forward1.ShootGoal()

'-- Step 5: Adding multiple responsibilities
'(When a player need to handle multiple roles)

'We already have Beckham as our midfielder.
'Let us ask him to play as an additional forward

Dim onemoreForward As New Forward()
onemoreForward.AssignPlayer(beck)

System.Console.WriteLine()
System.Console.WriteLine(" > Beckham has multiple responsibilities.. ")

'Now Beckham can shoot
onemoreForward.ShootGoal()
'And use his earlier responsibility to dribble too
midfielder1.Dribble()

'According to our design, you can attach the responsibility of
'a forward to a goal keeper too, but when you actually
'play football, remember that it is dangerous ;)

'Wait for key press
System.Console.Read()

End Sub

End Class

```

Running The Project

After executing the project, you'll get the following output.

```

> Warm up Session...
Fieldplayer (Owen) - passed the ball
Fieldplayer (Beckham) - passed the ball
GoalKeeper (Khan) - passed the ball

> Match is starting..
Fieldplayer (Owen) - passed the ball
Forward (Owen) - shot the ball to goalpost
Fieldplayer (Beckham) - passed the ball
Midfielder (Beckham) - dribbled the ball

> OOps, Owen got injured, Jerrard replaced Owen..
Forward (Jerrard) - shot the ball to goalpost

> Beckham has multiple responsibilities..
Forward (Beckham) - shot the ball to goalpost
Midfielder (Beckham) - dribbled the ball

```

Conclusion

I hope this article

- May help you to **understand** how to use design patterns.
- May help you some way to **apply** patterns in your projects
- May help you to **give** a brief talk about patterns to your friends :)

And bookmark/subscribe <http://amazedsaint.blogspot.com> for any update

Appendix – A: Source Code

For Section I & 2 – Goto <http://amazedsaint.blogspot.com/2008/01/design-patterns-part-i-and-ii.html> and click Download Source Files link

For Section 3 & 4- Goto <http://amazedsaint.blogspot.com/2008/01/practically-applying-design-patterns.html> and click Download Source files link