SSE 658

Design Problems and Solutions

Project #1

by

Jason Payne

February 10, 2013

TABLE OF CONTENTS

Topics Covered	Topic Examples
Design Patterns	• Strategy • Observer • Factory • Decorator

1. Strategy Design Pattern

1.1 Applied Example – Football Simulator

For the purposes of demonstrating the Strategy design pattern, an application that simulates the actions of a football team will be presented in this section. The general notion is that this simulator could be used in video game applications or software applications for football coaches and players. For the scope of this project, this portion of the application will focus on simulating the behavior and play-calling of the offense. The other aspects of the simulated football team would be similar in nature (i.e. behavior/play-calling of defense) and thus warrant no need of elaboration in this section. It should be noted, however, that future sections (not necessarily related to the Strategy design pattern) will elaborate on certain aspects of this simulator.

It is assumed that the reader has a general understanding of the rules of football (American football, not soccer!) and thus warrants no need for explanation of the rules of the game. Any uncommon, yet pertinent details requiring explanation will be provided in context.

1.1.1 Original Design

The concept for simulating offensive behavior of a football team is centered on the idea of a FootballTeam object and its ability to adapt its offensive behavior based on the score of the game. The code and class diagram of the FootballTeam object is provided below.

Simulator Code:

```
private static void RunOriginal()
{
   Console.WriteLine("***Before Strategy Pattern Applied***");

   var offense = new Original.FootballTeam("Offense", 11);

   Console.WriteLine("Starting the game.  The score is tied at zero...");
   RunOffensiveSeries(offense, OffensivePlay.Balanced);

   Console.WriteLine("{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
                     offense.Name);
   RunOffensiveSeries(offense, OffensivePlay.PassHappy);

   Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...", offense.Name);
   RunOffensiveSeries(offense, OffensivePlay.BallControl);
}

private static void RunOffensiveSeries(Original.FootballTeam team, OffensivePlay play)
{
   for (int i = 0; i < 4; i++)
   {
      team.RunOffensivePlay(play);
   }
}
```

Original FootballTeam Class:

```
public class FootballTeam
{
   protected readonly int _rosterSize;

   private string _name;

   public string Name
   {
```

```
        get { return string.Format("[{0}]", _name); }
        private set { _name = value; }
    }

    private FootballTeam()
    {
    }

    public FootballTeam(string name, int size)
    {
        Name = name;
        _rosterSize = size;
    }

    #region Offensive Behavior

    /// <summary>
    /// Determines if the offense should execute a running play.
    /// </summary>
    private bool _timeForRunPlay = true;

    public void RunOffensivePlay(OffensivePlay play)
    {
        switch (play)
        {
            case OffensivePlay.Balanced:
                break;
            case OffensivePlay.PassHappy:
                _timeForRunPlay = false;
                break;
            case OffensivePlay.BallControl:
                _timeForRunPlay = true;
                break;
            default:
                throw new ArgumentOutOfRangeException("play");
        }
        RunBalancedPlay();
    }

    private void RunBalancedPlay()
    {
        var msg = (_timeForRunPlay) ? "Ran a running play..." : "Ran a passing play...";
        Console.WriteLine(msg);
        _timeForRunPlay = !_timeForRunPlay;
    }

    #endregion // Offensive Behavior
}
```

Output:

```
***Before Strategy Pattern Applied***
Starting the game.  The score is tied at zero...
Ran a running play...
Ran a passing play...
Ran a running play...
Ran a passing play...
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
Ran a passing play...
Ran a passing play...
[Offense] is now ahead 14-7!  Time to run some clock...
```

```
Ran a running play...
Ran a running play...
Ran a running play...
Ran a running play...
```

As seen from the simulation, the simulator code is responsible for changing the behavior of the offense via usage of the OffensivePlay enum and the RunOffensivePlay method on the FootballTeam object. Therefore, this shows how the FootballTeam class is dependent on any client code – the simulator code in this case – to explicitly provide to the FootballTeam object the type of play that is to be executed at given times, which defines the behavior of the offense. In a more realistic simulation, it would be ideal for the FootballTeam object to not have to rely on client code to determine what type of play to run.

Another thing to consider is what happens if a new play type is introduced to the OffensivePlay enum (such as TrickPlay)? Code in the FootballTeam class has to be altered or extended to handle the new play type and the client code must be altered so that the client code's algorithms know when to indicate to the FootballTeam object that it needs to exhibit its supported behavior for the new play type.

Another thing to consider is what happens if a play type in the OffensivePlay enum is removed? All client code and FootballTeam objects would have to be edited to remove support for the now non-existent enum. Considering that there could be many different FootballTeam types (sub-classes such as RunningFootballTeam, PassingFootballTeam, SmartFootballTeam – just to name a few – that inherit from FootballTeam and define different offensive behaviors for play-calling), this could become a maintenance and customer service nightmare.

Ideally, the simulation code would present an object to the FootballTeam object that would represent the current game conditions such as the weather, score, results from the last play, etc. From this set of conditions, the FootballTeam object would alter its behavior without any dependencies on the client code. Fortunately, the Strategy design pattern helps developers accomplish this.

1.1.2 Improved Design Using Strategy Pattern

Ideally, the simulation code would present an object to the FootballTeam object that would represent the current game conditions such as the weather, score, results from the last play, etc. From this set of conditions, the FootballTeam object would alter its behavior without any dependencies on the client code. Fortunately, the Strategy design pattern helps developers accomplish this by encapsulating algorithms (behaviors) so that they become interchangeable and independent from clients that use them.

Using Strategy design principles, the first step is to identify what the variable parts of the application are and encapsulating them into their own type for easy reuse. This results in fewer unintended consequences from code changes and more flexibility in the system. For this application, the behavior of the offense needs to be able to change as plays are added or removed. The FootballTeam also needs to be able to change this behavior dynamically so that the simulation can run appropriately. This is accomplished by writing code to an interface and not a concrete implementation. So, in order to meet these criteria and design principles, the IOffensiveBehavior interface is created to serve as the "strategic encapsulation" of offensive behavior for FootballTeam objects. The source code is provided below.

Simulator Code

```
private static void RunNew()
{
```

```
   Console.WriteLine("***Strategy Pattern***");

   // The constructor now accepts a new parameter that specifies the offensive behavior.
   var offense = new FootballTeam("Offense", new BalancedOffense(), 11);

   Console.WriteLine("Starting the game.  The score is tied at zero...");
   RunOffensiveSeries(offense);

   Console.WriteLine("{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
                     offense.Name);
   offense.OffensiveBehavior = new PassHappyOffense();
   RunOffensiveSeries(offense);

   Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...", offense.Name);
   offense.OffensiveBehavior = new BallControlOffense();
   RunOffensiveSeries(offense);
}

// This method takes one less parameter now because the offensive behavior logic is contained
// within the FootballTeam class now.
private static void RunOffensiveSeries(FootballTeam team)
{
   for (int i = 0; i < 4; i++)
   {
      team.RunOffensivePlay();
   }
}
```

"Strategic" FootballTeam Class

```
public class FootballTeam
{
   protected readonly int _rosterSize;

   private string _name;

   public string Name
   {
      get { return string.Format("[{0}]", _name); }
      private set { _name = value; }
   }

   public IOffensiveBehavior OffensiveBehavior { get; set; }

   private FootballTeam()
   {
   }

   public FootballTeam(string name, int size)
   {
      Name = name;
      _rosterSize = size;
   }

   public FootballTeam(string name, IOffensiveBehavior offense, int size)
      : this(name, size)
   {
      OffensiveBehavior = offense;
   }

   public void RunOffensivePlay()
   {
      OffensiveBehavior.RunPlay();
```

```
    }
}
public interface IOffensiveBehavior
{
    void RunPlay();
}
public class BalancedOffense : IOffensiveBehavior
{
    private bool _timeForRunPlay = true;

    public void RunPlay()
    {
        var msg = (_timeForRunPlay) ? "Ran a running play..." : "Ran a passing play...";
        Console.WriteLine(msg);
        _timeForRunPlay = !_timeForRunPlay;
    }
}
public class BallControlOffense : IOffensiveBehavior
{
    public void RunPlay()
    {
        Console.WriteLine("Ran a running play...");
    }
}
public class PassHappyOffense : IOffensiveBehavior
{
    public void RunPlay()
    {
        Console.WriteLine("Ran a passing play...");
    }
}
```

Output

```
***Strategy Pattern***
Starting the game.  The score is tied at zero...
Ran a running play...
Ran a passing play...
Ran a running play...
Ran a passing play...
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
Ran a passing play...
Ran a passing play...
[Offense] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
Ran a running play...
Ran a running play...
```

Advantages:

- Changes to the offensive behavior, no longer require changes to the simulator code and FootballTeam classes. By encapsulating the algorithms (behavior) of the offense, any changes to those algorithms are contained within the classes that implement IOffensiveBehavior and nowhere else.
- By implementing to an interface (versus a concrete class such as OffensiveBehavior), the offensive behavior can be changed dynamically at run-time yielding a better simulation experience for the system.
- OffensivePlay enum can be removed because it is OBE.

- Other types of objects can reuse the offensive behaviors because these are no longer hidden away in the FootballTeam class

Disadvantages

- Any changes to offensive behavior have to be made across ALL classes that implement IOffensiveBehavior.
- Client code still must be aware of concrete implementations of IOffensiveBehavior in order to change the offensive behavior in code. Any change to existing constructors will break the client code.

2. Observer Design Pattern

2.1 Applied Example – Football Game Simulator

Continuing with the theme of the football simulator application presented in section 1, this section will expand the features of the simulation by adding a defense and an officiating crew in order to simulate an actual game. This provides a good example for illustrating the **Observer** design pattern.

The two designs presented here will offer two different implementations of the **Observer** pattern. The first will be a custom implementation that utilizes basic C# coding constructs without the use of any built-in features from the .NET Framework. The second design will present a similar solution with the only difference being the use of .NET built-in features to implement the **Observer** pattern.

For both solutions, the main concept is that in an actual game scenario, two teams would play one another with an officiating crew presiding over the contest in order to enforce the rules of gameplay. From a simulation standpoint, a team needs to be made aware when certain actions are executed by the opposing team such as changing offenses, changing defenses, changing personnel groupings, etc. Similarly, the officiating crew needs to be made aware of many of those same events so that the crew can determine if the rules are being followed or not. In terms of the **Observer** pattern, the football team is the "**observable**" and the officiating crew is the "**observer**". Also, due to the fact that teams must be aware of the actions from the opposing team, the football team is also an observer. So in this case, the football team is an observable and an observer object.

2.1.1 Custom Observer Pattern

For this implementation, a simulation is executed that is intended to simulate the actions of an actual game involving two teams and an officiating crew (but obviously on a smaller scale in this example). This simulation focuses on only "listening" to changes by the offense as the same procedures would be in place for the defensive team. As seen below, the officiating crew and the defensive team are registered with the offensive team in order to be made aware of any notifications that may be sent out by the offensive team, such as changing offenses, etc.

Simulator Code

```
public static void Run()
{
   Console.WriteLine("***Custom Observer Pattern***");

   var offense = new FootballTeam("Offense", new BalancedOffense(),
                                  new BalancedDefense());
   var defense = new FootballTeam("Defense", new BalancedOffense(),
                                  new BalancedDefense());
   var officiatingCrew = new OfficiatingCrew();

   offense.Register(officiatingCrew);
   offense.Register(defense);

   Console.WriteLine("Starting the game.  The score is tied at zero...");

   RunPlay(offense, defense);

   offense.OffensiveBehavior = new BallControlOffense();
   RunPlay(offense, defense);
```

```
   offense.OffensiveBehavior = new IllegalOffense();
   RunPlay(offense, defense);

   offense.OffensiveBehavior = new PassHappyOffense();
   RunPlay(offense, defense);

}

private static void RunPlay(FootballTeam offense, FootballTeam defense)
{
   offense.RunOffensivePlay();
   defense.RunDefensivePlay();
   Console.WriteLine("---This play is over---");
}
```

Observable Interface

This is the interface that all observable objects must implement in the Observer pattern. The idea is that observers can register/unregister with the observable object for change/update notifications. It is then up to the observable object to ensure that it is sending out the proper notifications via its concrete implementation of the NotifyListeners method.

```
public interface ITeamObservable
{
   void Register(ITeamObserver observer);
   void Unregister(ITeamObserver observer);
   void NotifyListeners(TeamStateChange changeType);
}

public enum TeamStateChange
{
   Offense,
   Defense
}
```

Observer Interface

This is the interface that all observer objects must implement in the Observer pattern. By implementing this interface and registering with ITeamObservable object, ITeamObserver objects will receive notifications of changes/updates from ITeamObservable objects.

```
public interface ITeamObserver
{
   void Update(FootballTeam footballTeam, TeamStateChange changeType);
}
```

FootballTeam Class (Observer and Observable)

This class is the logical representation of a football team. Because a typical football game simulation requires that teams are aware of actions, or events, from their opposing team, this FootballTeam class implements both ITeamObserver (as an observer object) and ITeamObservable (as an observable object).

```
public class FootballTeam : ITeamObserver, ITeamObservable
{
   public string Name { get; private set; }

   private IOffensiveBehavior _offensiveBehavior;
```

```
   public IOffensiveBehavior OffensiveBehavior
   {
      get { return _offensiveBehavior; }
      set
      {
         if (_offensiveBehavior != null)
         {
            Console.WriteLine("Changing offensive behavior...");
         }
         _offensiveBehavior = value;
         NotifyListeners(TeamStateChange.Offense);
      }
   }

   public IDefensiveBehavior DefensiveBehavior { get; set; }

   private FootballTeam()
   {
      subscribers = new List<ITeamObserver>();
   }

   public FootballTeam(string name,
                       IOffensiveBehavior offense, IDefensiveBehavior defense)
      : this()
   {
      Name = name;
      OffensiveBehavior = offense;
      DefensiveBehavior = defense;
   }

   public void RunOffensivePlay()
   {
      OffensiveBehavior.RunPlay();
   }

   public void RunDefensivePlay()
   {
      DefensiveBehavior.DefendPlay();
   }

   #region ITeamObserver Methods

   public void Update(FootballTeam footballTeam, TeamStateChange changeType)
   {
      Console.WriteLine("The opponent's offense has changed to {0}.",
                        footballTeam.OffensiveBehavior.GetType());
   }

   #endregion

   #region ITeamObservable Methods

   private List<ITeamObserver> subscribers;

   public void Register(ITeamObserver observer)
   {
      if (!subscribers.Contains(observer))
         subscribers.Add(observer);
```

```
    }

    public void Unregister(ITeamObserver observer)
    {
        if (subscribers.Contains(observer))
            subscribers.Remove(observer);
    }

    public void NotifyListeners(TeamStateChange changeType)
    {
        foreach (var subscriber in subscribers)
        {
            subscriber.Update(this, changeType);
        }
    }

    #endregion
}
```

OfficiatingCrew Class (Observer only)

This class is the logical representation of an officiating crew for a football game. For the sake of the simulation, OfficiatingCrew objects are only concerned with observing the FootballTeam objects and ensuring that any changes made by the teams are within the rules of the game. That is the reason these objects only implement ITeamObserver. In a more advanced simulation, this class could become an observable object in order to notify the simulation engine or involved teams of certain events or actions.

```
class OfficiatingCrew : ITeamObserver
{
    public void Update(FootballTeam footballTeam, TeamStateChange changeType)
    {
        switch (changeType)
        {
            case TeamStateChange.Offense:
                if(footballTeam.OffensiveBehavior is IllegalOffense)
                    Console.WriteLine("\tPENALTY!! --> The offense is illegal!");
                else
                    Console.WriteLine("The officiating crew has OK'd this offense.");
                break;
            case TeamStateChange.Defense:
                // Respond to defensive changes here
                break;
            default:
                throw new ArgumentOutOfRangeException("changeType");
        }
    }
}
```

Output

```
*** Custom Observer Pattern***
Starting the game.  The score is tied at zero...
Ran a running play...
Defending play with balanced defense.
---This play is over---
Changing offensive behavior...
The officiating crew has OK'd this offense.
The opponent's offense has changed to ObserverPatternExample.BallControlOffense.
Ran a running play...
```

```
Defending play with balanced defense.
---This play is over---
Changing offensive behavior...
       PENALTY!! --> The offense is illegal!
The opponent's offense has changed to ObserverPatternExample.IllegalOffense.
ILLEGAL OFFENSE!!!
Defending play with balanced defense.
---This play is over---
Changing offensive behavior...
The officiating crew has OK'd this offense.
The opponent's offense has changed to ObserverPatternExample.PassHappyOffense.
Ran a passing play...
Defending play with balanced defense.
---This play is over---
```

2.1.2 .NET Event-Based Observer Pattern

This implementation solution utilizes the .NET Framework's built-in delegate and event types to implement the **Observer** pattern. By utilizing these features, the ITeamObserver and ITeamObservable are no longer necessary, but are left in to illustrate how delegates and events can accomplish the same thing as the custom implementation in the preceding section. The most apparent difference is how this implementation no longer requires that the observable objects maintain their own collection of subscribers. Using delegates and events, this is handled automatically.

Updated FootballTeam Class (Observer and Observable)

```
public class FootballTeam : ITeamObserver, ITeamObservable
{
   private delegate void FootballTeamChangedEventHandler(
       FootballTeam footballTeam, TeamStateChange changeType);

   private event FootballTeamChangedEventHandler FootballTeamChanged;

   public string Name { get; private set; }

   private IOffensiveBehavior _offensiveBehavior;

   public IOffensiveBehavior OffensiveBehavior
   {
      get { return _offensiveBehavior; }
      set
      {
         if (_offensiveBehavior != null)
         {
            Console.WriteLine("Changing offensive behavior...");
         }
         _offensiveBehavior = value;
         NotifyListeners(TeamStateChange.Offense);
      }
   }

   public IDefensiveBehavior DefensiveBehavior { get; set; }

   private FootballTeam()
   {
   }

   public FootballTeam(string name, IOffensiveBehavior offense, IDefensiveBehavior
defense)
```

```
        : this()
    {
        Name = name;
        OffensiveBehavior = offense;
        DefensiveBehavior = defense;
    }

    public void RunOffensivePlay()
    {
        OffensiveBehavior.RunPlay();
    }

    public void RunDefensivePlay()
    {
        DefensiveBehavior.DefendPlay();
    }

    #region ITeamObserver Methods

    public void Update(FootballTeam footballTeam, TeamStateChange changeType)
    {
        Console.WriteLine("The opponent's offense has changed to {0}.",
                          footballTeam.OffensiveBehavior.GetType());
    }

    #endregion

    #region ITeamObservable Methods

    public void Register(ITeamObserver observer)
    {
        FootballTeamChanged += observer.Update;
    }

    public void Unregister(ITeamObserver observer)
    {
        FootballTeamChanged -= observer.Update;
    }

    public void NotifyListeners(TeamStateChange changeType)
    {
        if (FootballTeamChanged != null) FootballTeamChanged(this, changeType);
    }

    #endregion
}
```

Output

```
***Event-Based Observer Pattern***
Starting the game.  The score is tied at zero...
Ran a running play...
Defending play with balanced defense.
---This play is over---
Changing offensive behavior...
The officiating crew has OK'd this offense.
The opponent's offense has changed to EventPatternExample.BallControlOffense.
Ran a running play...
Defending play with balanced defense.
---This play is over---
```

```
Changing offensive behavior...
       PENALTY!! --> The offense is illegal!
The opponent's offense has changed to EventPatternExample.IllegalOffense.
ILLEGAL OFFENSE!!!
Defending play with balanced defense.
---This play is over---
Changing offensive behavior...
The officiating crew has OK'd this offense.
The opponent's offense has changed to EventPatternExample.PassHappyOffense.
Ran a passing play...
Defending play with balanced defense.
---This play is over---
```

Advantages

- Allows for one object to update/notify many objects.
- Observers and Observables can interact without being tightly coupled.
- (Custom) The registering/unregistering and notification mechanisms can be treated as any other class allowing for greater customization and flexibility.
- (Event-based) Can accomplish the same thing as Observer without the need to maintain a collection of subscribers.

Disadvantages

- (Custom) Have to maintain a collection of subscribers

3. Factory Design Pattern

3.1 Applied Example – Enhanced Football Simulator

For this section, the football simulator application presented in section 1 will be enhanced to allow for different simulations depending on the type of football team to simulate against. The thought is that an application like this could be used for debugging and testing different scenarios without requiring a separate simulation scenario for every type of FootballTeam object. This could be valuable because often times developers will end up using a lot of the same logic for simulations (or artificial intelligence) while only changing the type of team. Due to their popularity, game developers typically create college and NFL versions of games that have the same simulation engines with the only difference being the slight differences in the algorithms that exist due to different rules for the different levels of football (high school versus college versus professional).

3.1.1 Original Design

The initial design implemented a pattern based on the **Simple Factory** pattern. The CollegeFootballTeamFactory is the simple factory that produces the "college-type" FootballTeam objects while NflFootballTeamFactory does the same for "NFL-type" FootballTeam objects. This gets passed into the constructor for the simulator class (FootballGameSim) and when the simulation executes, it will run a simulation based upon the type of teams that its factory creates. The simulation then executes for each type of offense that is defined by the OffensiveType enum.

Simulator Code

The CollegeFootballTeamFactory is the simple factory that produces the "college-type" FootballTeam objects while NflFootballTeamFactory does the same for "NFL-type" FootballTeam objects. This gets passed into the constructor for the simulator class (FootballGameSim) and when the simulation executes, it will run a simulation based upon the type of teams that its factory creates. The simulation then executes for each type of offense that is defined by the OffensiveType enum.

```
public static void Run()
{
   Console.WriteLine("***Simple Factory Pattern***");

   var collegeTeamFactory = new CollegeFootballTeamFactory();
   var collegeSim = new FootballGameSim(collegeTeamFactory);

   var nflTeamFactory = new NflFootballTeamFactory();
   var nflSim = new FootballGameSim(nflTeamFactory);

   for (int i = 0; i < 3; i++)
   {
      collegeSim.Run((OffensiveType) i);
      nflSim.Run((OffensiveType) i);
   }
}
```

OffensiveType Enum

This is used for the purposes of this example as this defines the criteria requested of the simulation. There could be many more types that could be put into this type (i.e. TripleOption, NoHuddle, 5Wide, SpreadOption, Spread, etc.), but this satisfies the contexts for this example.

```
public enum OffensiveType
{
   Balanced,
   PassHappy,
   BallControl
};
```

Simulator Engine

This is the main class that contains the simulation logic, or engine, that drives the simulation scenarios. It is the same logic used in section 1 as this provides good detail to the concepts explained in this section. In a realistic scenario, this class could provide different and/or extendable measures for providing different scenarios for simulation.

The highlighted line is the factory's creator method that is implementing the **Simple Factory** design pattern. By providing an interface to this class's constructor, different types of factories can be used by this simulation (as long as the factory implements the IFootballTeamFactory interface) which satisfies several design principles. However, a potential pitfall to this design is the dependence of the FootballGameSim on an IFootballTeamFactory object. Ideally, a **Factory Method** pattern would be implemented in place of the simple factory, thus removing any dependencies. In the following section, the improved design will provide a solution for this problem.

The RunSim method does not care what type of FootballTeam it is passed, just as long as that parameter implements the abstract FootballTeam class. This simulation has added some information that is to be presented when the simulation begins. This will prove useful for debugging and logging purposes.

```
public class FootballGameSim
{
   private readonly IFootballTeamFactory _teamFactory;

   public FootballGameSim(IFootballTeamFactory footballTeamFactory)
   {
      _teamFactory = footballTeamFactory;
   }

   public void Run(OffensiveType type)
   {
      var team = _teamFactory.Create(type);

      RunSim(team);
   }

   private static void RunSim(FootballTeam team)
   {
      Console.WriteLine("{0} is coached by {1} and has {2} players on the roster.",
                        team.Name, team.HeadCoach, team.RosterSize);
      Console.WriteLine("Starting the game.  The score is tied 0-0...");
      team.OffensiveBehavior = new BalancedOffense();
      RunOffensiveSeries(team);

      Console.WriteLine(
         "{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
         team.Name);
      team.OffensiveBehavior = new PassHappyOffense();
      RunOffensiveSeries(team);
```

```
      Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...",
                        team.Name);
      team.OffensiveBehavior = new BallControlOffense();
      RunOffensiveSeries(team);

      Console.WriteLine("The game is over!" + Environment.NewLine);
   }

   private static void RunOffensiveSeries(FootballTeam team)
   {
      for (int i = 0; i < 2; i++)
      {
         team.RunOffensivePlay();
      }
   }
}
```

Simple Factory Classes

As seen below, these factory classes implement the IFootballTeamFactory interface which allows it to be used and changed dynamically in the FootballGameSim constructor. The main problem with this design pattern stems from the highlighted code below. Different and specific classes must be created for the multiple enumerations of OffensiveType combined with the product class that the factory is expected to create. This results in double the amount of classes required by the application which also compounds potential maintenance problems. In the improved design, the **Abstract Factory** pattern will be utilized to help alleviate these issues.

```
public interface IFootballTeamFactory
{
   FootballTeam Create(OffensiveType type);
}

public class CollegeFootballTeamFactory : IFootballTeamFactory
{
   public FootballTeam Create(OffensiveType type)
   {
      FootballTeam result;

      switch (type)
      {
         case OffensiveType.Balanced:
            result = new CollegeBalancedFootballTeam();
            break;
         case OffensiveType.PassHappy:
            result = new CollegePassHappyFootballTeam();
            break;
         case OffensiveType.BallControl:
            result = new CollegeBallControlFootballTeam();
            break;
         default:
            throw new ArgumentOutOfRangeException("type");
      }

      return result;
   }
}
```

```
public class NflFootballTeamFactory : IFootballTeamFactory
{
   public FootballTeam Create(OffensiveType type)
   {
      FootballTeam result;

      switch (type)
      {
         case OffensiveType.Balanced:
            result = new NflBalancedFootballTeam();
            break;
         case OffensiveType.PassHappy:
            result = new NflPassHappyFootballTeam();
            break;
         case OffensiveType.BallControl:
            result = new NflBallControlFootballTeam();
            break;
         default:
            throw new ArgumentOutOfRangeException("type");
      }

      return result;
   }
}
```

Concrete Product Classes

This is a variation of the FootballTeam class presented in section 1. Here, it is an abstract class as other "product" classes will inherit from this class in order to create the more specific classes that will be returned by the above simple factory classes.

One important aspect of this class is the inclusion of a variable that represents the roster size of a team. Football teams have different restrictions on the number of "active" members a team can have at one time. This differs based on the developmental level of the team. High school teams have no roster limits, whereas, college and NFL teams have roster limits of 85 and 53 members, respectively. To further complicate things, the Canadian Football League has standard roster limits and further limits based on whether a player is a Canadian citizen or not! Details like these illustrate how complex it could be for different simulations for different types of football teams. For the purposes of this example, the roster limits based on college and NFL rules will be imposed in the implementation.

```
public abstract class FootballTeam
{
   protected readonly int _rosterSize;

   public int RosterSize
   {
      get { return _rosterSize; }
   }

   public string Name { get; protected set; }
   public string HeadCoach { get; protected set; }

   public IOffensiveBehavior OffensiveBehavior { get; set; }

   protected FootballTeam()
   {
      _rosterSize = -1;
```

```
    }

    protected FootballTeam(int rosterSize)
    {
        _rosterSize = rosterSize;
    }

    public void RunOffensivePlay()
    {
        OffensiveBehavior.RunPlay();
    }
}
```

Below are the concrete implementations of FootballTeam for a “college-type” and an “NFL-type” class. Note the differences in the roster sizes and how the Name and HeadCoach are initialized. Ideally, implementation details regarding the roster size and format of the strings of Name and HeadCoach would not need be left to these product classes to enforce because there is no guarantee that these product classes would implement them the same way. As seen in the improved design, using an **Abstract Factory** object to provide the criteria, or “ingredients”, for object creation will help alleviate these concerns while eliminating the need for these specifically typed classes.

```
public class CollegeBalancedFootballTeam : FootballTeam
{
    private const int ROSTER_SIZE = 85;

    public CollegeBalancedFootballTeam()
        : base(ROSTER_SIZE)
    {
        Name = "[Balanced College Team]";
        HeadCoach = "Balanced College Coach";
    }
}

public class NflBalancedFootballTeam : FootballTeam
{
    private const int ROSTER_SIZE = 53;

    public NflBalancedFootballTeam()
        : base(ROSTER_SIZE)
    {
        Name = "[Balanced NFL Team]";
        HeadCoach = "Balanced NFL Coach";
    }
}

public class CollegePassHappyFootballTeam : FootballTeam
{
    // Another FootballTeam concrete class...
}

public class NflPassHappyFootballTeam : FootballTeam
{
    // Another FootballTeam concrete class...
}

public class CollegeBallControlFootballTeam : FootballTeam
{
```

```
    // Another FootballTeam concrete class...
}

public class NflBallControlFootballTeam : FootballTeam
{
    // Another FootballTeam concrete class...
}
```

Output

```
***Simple Factory Pattern***
[Balanced College Team] is coached by Balanced College Coach and has 85 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[Balanced College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Balanced College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[Balanced NFL Team] is coached by Balanced NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[Balanced NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Balanced NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[PassHappy College Team] is coached by PassHappy College Coach and has 85 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[PassHappy College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[PassHappy College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[PassHappy NFL Team] is coached by PassHappy NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[PassHappy NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[PassHappy NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[BallControl College Team] is coached by BallControl College Coach and has 85 players on the
roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
```

```
Ran a passing play...
[BallControl College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[BallControl College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[BallControl NFL Team] is coached by BallControl NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[BallControl NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[BallControl NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!
```

3.1.2 Improved Design Using Factory Pattern

One of the immediate observations from the original design is the sheer number of concrete product classes that are required to be implemented in order to support the variations of the simulations. Because there are two simulations – one for college teams and one for NFL teams – and 3 types of OffensiveTypes, then that equates to 6 total classes needed for implementation. In each concrete product class there are implementation details (roster size, team name, etc.) that must be duplicated across each concrete implementation. In a real world scenario, under this current design, this could equate to hundreds of concrete classes requiring similar or duplicated code for each class increasing the possibility of errors and maintenance difficulties.

To help minimize the duplication and remove dependencies, the **Factory Method** and **Abstract Factory** design patterns were utilized to address these concerns. For conciseness, only the impacted files are listed below.

Updated Simulator Code

```
public static void Run()
{
   Console.WriteLine("***Abstract Factory Method Pattern***");

   var collegeSim = new CollegeFootballGameSim();
   var nflSim = new NflFootballGameSim();

   for (int i = 0; i < 3; i++)
   {
      collegeSim.Run((OffensiveType) i);
      nflSim.Run((OffensiveType)i);
   }
}
```

Updated Simulator Engine

By applying the **Factory Method** pattern, the FootballGameSim class was converted to an abstract class so that concrete implementations of FootballGameSim could override the CreateTeam method. Also note that there is no longer a dependency (or a constructor) to a FootballTeam factory object because the concrete implementations of this class will handle the creation of the FootballTeam concrete classes (via the CreateTeam method).

```
public abstract class FootballGameSim
{
   public void Run(OffensiveType type)
   {
      FootballTeam team;

      team = CreateTeam(type);

      RunSim(team);
   }

   private static void RunSim(FootballTeam team)
   {
      Console.WriteLine("{0} is coached by {1} and has {2} players on the roster.",
                        team.Name, team.HeadCoach, team.RosterSize);
      Console.WriteLine("Starting the game.  The score is tied 0-0...");
      team.OffensiveBehavior = new BalancedOffense();
      RunOffensiveSeries(team);

      Console.WriteLine(
         "{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
         team.Name);
      team.OffensiveBehavior = new PassHappyOffense();
      RunOffensiveSeries(team);

      Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...",
                        team.Name);
      team.OffensiveBehavior = new BallControlOffense();
      RunOffensiveSeries(team);

      Console.WriteLine("The game is over!" + Environment.NewLine);
   }

   private static void RunOffensiveSeries(FootballTeam team)
   {
      for (int i = 0; i < 2; i++)
      {
         team.RunOffensivePlay();
      }
   }

   protected abstract FootballTeam CreateTeam(OffensiveType type);
}
```

Rather than requiring a direct dependency to a FootballTeam factory object, the FootballGameSim class was abstracted so that it could be extended with concrete implementations providing their own custom creation methods based on the type of simulation that is being run. This allows for easy extensibility for any future FootballTeam types and the simulations that would need to be run with them.

Also note how these concrete implementations create abstract factory objects that are passed to the concrete FootballTeam objects. This too allows for easier maintenance and reusability. One very important thing to note, however, is that the concrete implementations are no longer type specific as they were in the original design (CollegeXXXFootballTeam, NflXXXFootballTeam). This is because the abstract factories provide the details for creating the type specific objects. It has been abstracted away into the ITeamCreationCriteriaAbstractFactory objects, which is the desired effect of the **Abstract Method**.

```
public class CollegeFootballGameSim : FootballGameSim
{
   protected override FootballTeam CreateTeam(OffensiveType type)
   {
      FootballTeam result;

      var collegeCriteriaAbstractFactory =
         new CollegeTeamCreationCriteriaAbstractFactory(type);

      switch (type)
      {
         case OffensiveType.Balanced:
            result = new BalancedFootballTeam(collegeCriteriaAbstractFactory);
            break;
         case OffensiveType.PassHappy:
            result = new PassHappyFootballTeam(collegeCriteriaAbstractFactory);
            break;
         case OffensiveType.BallControl:
            result = new BallControlFootballTeam(collegeCriteriaAbstractFactory);
            break;
         default:
            throw new ArgumentOutOfRangeException("type");
      }

      return result;
   }
}

public class NflFootballGameSim : FootballGameSim
{
   protected override FootballTeam CreateTeam(OffensiveType type)
   {
      FootballTeam result;

      var nflCriteriaAbstractFactory =
         new NflTeamCreationCriteriaAbstractFactory(type);

      switch (type)
      {
         case OffensiveType.Balanced:
            result = new BalancedFootballTeam(nflCriteriaAbstractFactory);
            break;
         case OffensiveType.PassHappy:
            result = new PassHappyFootballTeam(nflCriteriaAbstractFactory);
            break;
         case OffensiveType.BallControl:
            result = new BallControlFootballTeam(nflCriteriaAbstractFactory);
            break;
         default:
            throw new ArgumentOutOfRangeException("type");
      }

      return result;
   }
}
```

New Factory Classes

These are the abstract factories that provide the criteria, or “ingredients”, for the concrete FootballTeam objects that are to be created. Note the different sizes of the rosters for the two concrete factories. If the criteria ever changes for any of the FootballTeam objects, those details are all contained within these abstract factories making maintenance efforts much more efficient and isolated.

```
public interface ITeamCreationCriteriaAbstractFactory
{
   int RosterSize { get; }
   string Name { get; }
   string HeadCoach { get; }
}

public class CollegeTeamCreationCriteriaAbstractFactory :
   ITeamCreationCriteriaAbstractFactory
{
   public int RosterSize
   {
      get { return 85; }
   }

   public string Name { get; private set; }

   public string HeadCoach { get; private set; }

   public CollegeTeamCreationCriteriaAbstractFactory(OffensiveType type)
   {
      HeadCoach = string.Format("{0} {1}", type, "College Coach");
      Name = string.Format("[{0} College Team]", type);
   }
}

public class NflTeamCreationCriteriaAbstractFactory :
ITeamCreationCriteriaAbstractFactory
{
   public int RosterSize
   {
      get { return 53; }
   }

   public string Name { get; private set; }

   public string HeadCoach { get; private set; }

   public NflTeamCreationCriteriaAbstractFactory(OffensiveType type)
   {
      HeadCoach = string.Format("{0} {1}", type, "NFL Coach");
      Name = string.Format("[{0} NFL Team]", type);
   }
}
```

Updated Concrete Product Classes

These are the updated product classes that are essentially produced by the factory classes. For the most part it is the same, but the constructor has been updated to accept a team name and a head coach name along with the roster size. The BuildRoster abstract method was added to illustrate how different concrete implementations would utilize the ITeamCreationCriteriaAbstractFactory object that is now being passed to them. This is important

because this shows how the factory object can be used to specify type-specific details depending on the implementation. This is what allowed the six concrete implementations of the concrete FootballTeam classes (from the original design) to be reduced to three!

```
public abstract class FootballTeam
{
   public int RosterSize { get; private set; }
   public string Name { get; private set; }
   public string HeadCoach { get; private set; }
   public IOffensiveBehavior OffensiveBehavior { get; set; }

   protected FootballTeam(string name, string coach, int rosterSize)
   {
      Name = name;
      HeadCoach = coach;
      RosterSize = rosterSize;
      BuildRoster();
   }

   public void RunOffensivePlay()
   {
      OffensiveBehavior.RunPlay();
   }

   protected abstract void BuildRoster();
}

public class BalancedFootballTeam : FootballTeam
{
   private ITeamCreationCriteriaAbstractFactory _teamCreationCriteria;

   public BalancedFootballTeam(
      ITeamCreationCriteriaAbstractFactory teamCreationCriteriaAbstractFactory)
      : base(
         teamCreationCriteriaAbstractFactory.Name,
         teamCreationCriteriaAbstractFactory.HeadCoach,
         teamCreationCriteriaAbstractFactory.RosterSize)
   {
      _teamCreationCriteria = teamCreationCriteriaAbstractFactory;
   }

   protected override void BuildRoster()
   {
      // Build team specific roster here...

      // AddQuarterbacks();
      // AddRunningBacks();
      // etc.
   }
}

public class BallControlFootballTeam : FootballTeam
{
   private ITeamCreationCriteriaAbstractFactory _teamCreationCriteria;

   public BallControlFootballTeam(
      ITeamCreationCriteriaAbstractFactory teamCreationCriteriaAbstractFactory)
      : base(
```

```
            teamCreationCriteriaAbstractFactory.Name,
            teamCreationCriteriaAbstractFactory.HeadCoach,
            teamCreationCriteriaAbstractFactory.RosterSize)
    {
        _teamCreationCriteria = teamCreationCriteriaAbstractFactory;
    }

    protected override void BuildRoster()
    {
        // Build team specific roster here...

        // AddQuarterbacks();
        // AddRunningBacks();
        // etc.
    }
}

public class PassHappyFootballTeam : FootballTeam
{
    private ITeamCreationCriteriaAbstractFactory _teamCreationCriteria;

    public PassHappyFootballTeam(
        ITeamCreationCriteriaAbstractFactory teamCreationCriteriaAbstractFactory)
        : base(
            teamCreationCriteriaAbstractFactory.Name,
            teamCreationCriteriaAbstractFactory.HeadCoach,
            teamCreationCriteriaAbstractFactory.RosterSize)
    {
        _teamCreationCriteria = teamCreationCriteriaAbstractFactory;
    }

    protected override void BuildRoster()
    {
        // Build team specific roster here...

        // AddQuarterbacks();
        // AddRunningBacks();
        // etc.
    }
}
```

Output

```
***Abstract Factory Method Pattern***
[Balanced College Team] is coached by Balanced College Coach and has 85 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[Balanced College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Balanced College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[Balanced NFL Team] is coached by Balanced NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
```

```
[Balanced NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Balanced NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[PassHappy College Team] is coached by PassHappy College Coach and has 85 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[PassHappy College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[PassHappy College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[PassHappy NFL Team] is coached by PassHappy NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[PassHappy NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[PassHappy NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[BallControl College Team] is coached by BallControl College Coach and has 85 players on the
roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[BallControl College Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[BallControl College Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!

[BallControl NFL Team] is coached by BallControl NFL Coach and has 53 players on the roster.
Starting the game.  The score is tied 0-0...
Ran a running play...
Ran a passing play...
[BallControl NFL Team] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[BallControl NFL Team] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
The game is over!
```

The output is the same even though the design changed, but now it is easier to write new concrete classes for new FootballTeam objects and there are fewer classes to maintain.

Advantages

- Factories handle the details of object creation. Using a factory can simplify object creation by reducing the required criteria from client code and letting the factory handle the details of object creation.
- By using a factory to handle the criteria for product creation (i.e. the constructor's criteria), general classes can be utilized resulting in fewer product classes to maintain. For instance, the improved design above was able to reduce the need for specific types of FootballTeams such as CollegeXXXFootballTeam and NflXXXFootballTeam in favor of the more general class "XXXFootballTeam".
- Changes to constructors are encapsulated within the factory and hidden from client code.
- Changes to criteria required by concrete implementations are isolated into the concrete implementations of the abstract factories resulting in efficient maintenance.

Disadvantages

- This design does not address the fact that many potential product classes would need to be handled by the factory potentially involving long if-else or switch statements.
- The abstract factory object(s) providing the criteria ("ingredients") to the product classes does not address the concern of the product classes still being able to mismanage the data provided by the criteria factory object(s).
- The concrete product class implementations now have a dependency on the abstract factory objects.

4. Decorator Design Pattern

4.1 Applied Example – Enhanced Football Simulator

This application is an enhanced version of the simulation present in section 1 for the Strategy pattern demonstration. It enhances that simulation by providing more specific offensive behaviors for use by the simulation and provides a great opportunity to demonstrate the **Decorator** design pattern.

4.1.1 Original Design[1]

The initial design for this application continues where the simulation at the end of the Strategy pattern section left off. The offensive behavior of FootballTeam objects can be changed dynamically to concrete implementations of the IOffensiveBehavior interface. However, this simulation enhances the previous one by creating and providing new implementations of IOffensiveBehavior to account for formations and personnel groupings rather than general offensive behaviors (CollegePistol3Wr1Rb1TeOffense is a more specific behavior as compared to that of BalancedOffense). However, due to many different variations of possible formations and personnel groupings, there would need to be tens to hundreds of different concrete implementations of IOffensiveBehavior to account for them all. The **Decorator** pattern will help address these concerns in the improved design.

Concrete IOffensiveBehavior Classes

These are concrete implementations of the IOffensiveBehavior interface. They are simply additional offensive behavior types that have been added to the ones that existed for the other simulations. In a more realistic scenario, these IOffensiveBehavior objects would perform some simulated action for running a play, but for the context of this project, they simply print an information message to the console in the format of <*Football Level*>.<*Formation*>.{<*Formation Attribute 1*>.<*Formation Attribute 2*>…<*Formation Attribute X*>}.

The problem with this design is that an entirely new class has to be created for each variation of each formation. For instance, if an NFL offensive behavior, NflPistol3Wr1Rb1TeOffense, needed to take on some of the similar characteristics of the CollegePistol3Wr1Rb1TeOffense object then a new class would have to be created that inherits from IOffensiveBehavior or extends CollegePistol3Wr1Rb1TeOffense. Either approach duplicates code and provides for a rigid design. By applying the Decorator pattern, decorators that "combine" to decorate a concrete implementation of IOffensiveBehavior can be used to decorate other concrete IOffensiveBehavior objects while also providing similar functionality and design flexibility.

```
public class CollegePistol3Wr1Rb1TeOffense : IOffensiveBehavior
{
   public void RunPlay()
   {
      var msg = "The College.Pistol.3WR.1RB.1TE offense ran a play...";
      Console.WriteLine(msg);
   }
}

public class NflShotgun4Wr1RbOffense : IOffensiveBehavior
{
   public void RunPlay()
   {
      var msg = "The NFL.Shotgun.4WR.1RB offense ran a play...";
```

[1] *The source code for the FootballTeam class and the IOffensiveBehavior interface did not change.*

```
      Console.WriteLine(msg);
   }
}

public class HighSchoolWishbone3Rb2TeOffense : IOffensiveBehavior
{
   public void RunPlay()
   {
      var msg = "The HighSchool.Wishbone.3RB.2TE offense ran a play...";
      Console.WriteLine(msg);
   }
}
```

Simulator Code

The lines highlighted below show the new concrete implementations of the IOffensiveBehavior interface and how they are used in the simulation.

```
public static void Run()
{
   Console.WriteLine("***Non-Decorator Applied***");

   var offensiveBehavior = new CollegePistol3Wr1Rb1TeOffense();
   var omniTeam = new FootballTeam("Omni Team", offensiveBehavior, 11);

   Console.WriteLine("Starting the game.  The score is tied at zero...");
   RunOffensiveSeries(omniTeam);

   Console.WriteLine(
      "{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
      omniTeam.Name);
   omniTeam.OffensiveBehavior = new NflShotgun4Wr1RbOffense();
   RunOffensiveSeries(omniTeam);

   Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...",
                     omniTeam.Name);
   omniTeam.OffensiveBehavior = new HighSchoolWishbone3Rb2TeOffense();
   RunOffensiveSeries(omniTeam);
}

private static void RunOffensiveSeries(FootballTeam team)
{
   for (int i = 0; i < 2; i++)
   {
      team.RunOffensivePlay();
   }
}
```

Output

```
***Non-Decorator Applied***
Starting the game.  The score is tied at zero...
The College.Pistol.3WR.1RB.1TE offense ran a play...
The College.Pistol.3WR.1RB.1TE offense ran a play...
[Omni Team] is now behind 7-0!  Time to get aggressive with the playcalling...
The NFL.Shotgun.4WR.1RB offense ran a play...
The NFL.Shotgun.4WR.1RB offense ran a play...
[Omni Team] is now ahead 14-7!  Time to run some clock...
The HighSchool.Wishbone.3RB.2TE offense ran a play...
The HighSchool.Wishbone.3RB.2TE offense ran a play...
```

4.1.2 Improved Design Using Decorator Pattern

The design in this section simply applies the Decorator pattern to the same simulation in the preceding section. This shows its usefulness, however, by allowing different variations of offensive formation and personnel groupings to be created and assigned to the OffensiveBehavior property of FootballTeam objects dynamically!

Updated IOffensiveBehavior Interface

In order to properly demonstrate the application of the Decorator pattern, the IOffensiveBehavior needed to be slightly modified so that the RunPlay method returned a string instead of performing a void action. This was so that each decorator could show how it was applied and changed at run-time.

```
public interface IOffensiveBehavior
{
   string RunPlay();
}
```

Concrete IOffensiveBehaviors Classes

These classes are the concrete implementations of the IOffensiveBehavior interface. They represent the "first level" implementations of ALL IOffensiveBehavior objects. The concept is that for every developmental level of football that exists, there should be a first level implementation of IOffensiveBehavior that serves as the "root" that all offenses extend from. Working off the basis of the popular levels of American football, there exists three developmental levels of football (High School, Collegiate, and NFL) and as such there should only be a limited amount of these classes that exist for the system.

```
public class NflOffense : IOffensiveBehavior
{
   public string RunPlay()
   {
      return "NFL";
   }
}

public class CollegeOffense : IOffensiveBehavior
{
   public string RunPlay()
   {
      return "College";
   }
}

public class HighSchoolOffense : IOffensiveBehavior
{
   public string RunPlay()
   {
      return "HighSchool";
   }
}
```

IOffenseBehavior Decorator Classes

The main thought here is that football offenses will have a large number of personnel and formation variations. It would not make good design sense to create a concrete implementation for each possible variation (as was done in the original design) and that is where the Decorator design pattern comes into play. These classes are the "decorator" objects that allow the

functionality and design of concrete IOffensiveBehavior objects to be altered dynamically. The functionality that they provide can be shared across any concrete IOffensiveBehavior object making the multiple variations of formation and personnel groupings easier to implement and maintain.

The functionality is extended by making use of the reference to the IOffensiveBehavior object that is passed into the constructor. Any shared implementation by the decorator is executed directly before or after the field member's functionality is executed.

```
public class _1Te : IOffensiveBehavior
{
   private readonly IOffensiveBehavior _offensiveBehavior;

   public _1Te(IOffensiveBehavior behavior)
   {
      _offensiveBehavior = behavior;
   }

   public string RunPlay()
   {
      return _offensiveBehavior.RunPlay() + ".1TE";
   }
}

public class _2Te : IOffensiveBehavior
{
   private readonly IOffensiveBehavior _offensiveBehavior;

   public _2Te(IOffensiveBehavior behavior)
   {
      _offensiveBehavior = behavior;
   }

   public string RunPlay()
   {
      return _offensiveBehavior.RunPlay() + ".2TE";
   }
}

public class _1Rb : IOffensiveBehavior
{
   private readonly IOffensiveBehavior _offensiveBehavior;

   public _1Rb(IOffensiveBehavior behavior)
   {
      _offensiveBehavior = behavior;
   }

   public string RunPlay()
   {
      return _offensiveBehavior.RunPlay() + ".1RB";
   }
}

public class _3Rb : IOffensiveBehavior
{
   private readonly IOffensiveBehavior _offensiveBehavior;
```

```
    public _3Rb(IOffensiveBehavior behavior)
    {
        _offensiveBehavior = behavior;
    }

    public string RunPlay()
    {
        return _offensiveBehavior.RunPlay() + ".3RB";
    }
}

public class _3Wr : IOffensiveBehavior
{
    private readonly IOffensiveBehavior _offensiveBehavior;

    public _3Wr(IOffensiveBehavior behavior)
    {
        _offensiveBehavior = behavior;
    }

    public string RunPlay()
    {
        return _offensiveBehavior.RunPlay() + ".3WR";
    }
}

public class _4Wr : IOffensiveBehavior
{
    private readonly IOffensiveBehavior _offensiveBehavior;

    public _4Wr(IOffensiveBehavior behavior)
    {
        _offensiveBehavior = behavior;
    }

    public string RunPlay()
    {
        return _offensiveBehavior.RunPlay() + ".4WR";
    }
}

public class Shotgun : IOffensiveBehavior
{
    private readonly IOffensiveBehavior _offensiveBehavior;

    public Shotgun(IOffensiveBehavior behavior)
    {
        _offensiveBehavior = behavior;
    }

    public string RunPlay()
    {
        return _offensiveBehavior.RunPlay() + ".Shotgun";
    }
}

public class Pistol : IOffensiveBehavior
{
```

```
   private readonly IOffensiveBehavior _offensiveBehavior;

   public Pistol(IOffensiveBehavior behavior)
   {
      _offensiveBehavior = behavior;
   }

   public string RunPlay()
   {
      return _offensiveBehavior.RunPlay() + ".Pistol";
   }
}

public class Wishbone : IOffensiveBehavior
{
   private readonly IOffensiveBehavior _offensiveBehavior;

   public Wishbone(IOffensiveBehavior behavior)
   {
      _offensiveBehavior = behavior;
   }

   public string RunPlay()
   {
      return _offensiveBehavior.RunPlay() + ".Wishbone";
   }
}
```

Updated FootballTeam Class

For this example, the FootballTeam class was slightly modified to accommodate the modification to the IOffensiveBehavior interface. Everything else remained the same with this class.

```
public class FootballTeam
{
   protected readonly int _rosterSize;

   private string _name;

   public string Name
   {
      get { return string.Format("[{0}]", _name); }
      private set { _name = value; }
   }

   public IOffensiveBehavior OffensiveBehavior { get; set; }

   private FootballTeam()
   {
   }

   public FootballTeam(string name, int size)
   {
      Name = name;
      _rosterSize = size;
   }

   public FootballTeam(string name, IOffensiveBehavior offense, int size)
      : this(name, size)
```

```
    {
        OffensiveBehavior = offense;
    }

    public void RunOffensivePlay()
    {
        var msg = OffensiveBehavior.RunPlay();
        Console.WriteLine("The {0} offense ran a play...", msg);
    }
}
```

Updated Simulator Code

This is the updated code for the simulation after the **Decorator** pattern has been applied. Note how the constructors that create the concrete IOffensiveBehavior objects are implemented. The "nested" new-statements are the decorators being applied at run-time while the "root" of the nested constructor always ends with a concrete IOffensiveBehavior object. Also notice how one of the decorators (_1Rb) is used to instantiate different objects at different levels in the nested constructors. This is the **Decorator** pattern at work!

```
public static void Run()
{
    Console.WriteLine("***Decorator Pattern***");
    IOffensiveBehavior offensiveBehavior;

    offensiveBehavior = new _1Te(new _1Rb(new _3Wr(new Pistol(new CollegeOffense()))));
    var omniTeam = new FootballTeam("Omni Team", offensiveBehavior, 11);

    Console.WriteLine("Starting the game.  The score is tied at zero...");
    RunOffensiveSeries(omniTeam);

    Console.WriteLine(
        "{0} is now behind 7-0!  Time to get aggressive with the playcalling...",
        omniTeam.Name);
    offensiveBehavior = new _1Rb(new _4Wr(new Shotgun(new NflOffense())));
    omniTeam.OffensiveBehavior = offensiveBehavior;
    RunOffensiveSeries(omniTeam);

    Console.WriteLine("{0} is now ahead 14-7!  Time to run some clock...",
                      omniTeam.Name);
    offensiveBehavior = new _2Te(new _3Rb(new Wishbone(new HighSchoolOffense())));
    omniTeam.OffensiveBehavior = offensiveBehavior;
    RunOffensiveSeries(omniTeam);
}

private static void RunOffensiveSeries(FootballTeam team)
{
    for (int i = 0; i < 2; i++)
    {
        team.RunOffensivePlay();
    }
}
```

Output

```
***Decorator Pattern***
Starting the game.  The score is tied at zero...
The College.Pistol.3WR.1RB.1TE offense ran a play...
The College.Pistol.3WR.1RB.1TE offense ran a play...
```

```
[Omni Team] is now behind 7-0!  Time to get aggressive with the playcalling...
The NFL.Shotgun.4WR.1RB offense ran a play...
The NFL.Shotgun.4WR.1RB offense ran a play...
[Omni Team] is now ahead 14-7!  Time to run some clock...
The HighSchool.Wishbone.3RB.2TE offense ran a play...
The HighSchool.Wishbone.3RB.2TE offense ran a play...
```

Advantages

- "Decorator" objects can be used to decorate any concrete instantiation as long as the base class of the concrete object and decorator object are of the same type.
- Concrete objects can have their functionality extended dynamically while also providing a very flexible design for future modifications without the need to change the base classes.
-

Disadvantages

- The number of decorator objects could become large in number making it hard to maintain.
- Without applying other design patterns, such as Factory, object instantiation could become complex and very nested ("new Type1(new Type2(new Type3(new Type4())))").
- The order of execution for the code that is shared between the concrete and decorator objects becomes dependent upon how the decorators are applied at object creation time. Decorating an object in the wrong order, at object instantiation, could introduce bugs that prove difficult to track down.

Appendix A – *Non-Direct Activity Report*

Date	Duration (minutes)	Specific Task / Activity
6-Jan-2014	60	Class syllabus review, class text review
7-Jan-2014	30	Setup of Direct/Non-Direct Activity Reports
16-Jan-2014	91	Setup of Direct/Non-Direct Activity Reports, project research
19-Jan-2014	195	Reading listserv, research for Project #1, chapter 1 of text
20-Jan-2014	80	Reading listserv, research for Project #1, chapter 1 of text
21-Jan-2014	22	Reading listserv, research for Project #1, chapter 1 of text
22-Jan-2014	65	Reading listserv, research for Project #1, chapter 1 of text, updating activity reports
23-Jan-2014	11	Reading listserv
26-Jan-2014	197	Reading listserv, research for Project #1, writing code
27-Jan-2014	62	Writing code for Project #1
3-Feb-2014	384	Improvements to activity report worksheet, research for Project #1, writing code
4-Feb-2014	199	Research for Project #1
5-Feb-2014	155	Research & write code/report for Project #1
6-Feb-2014	135	Research & write code/report for Project #1
7-Feb-2014	145	Research & write code/report for Project #1
8-Feb-2014	752	Research & write code/report for Project #1
9-Feb-2014	720	Research & write code/report for Project #1
10-Feb-2014	375	Research & write code/report for Project #1
Sum for Report #1	**3678**	**/ 1500 (5 weeks @ 300/wk)**