SSE 658

Design Problems and Solutions

Project #2

by

Jason Payne

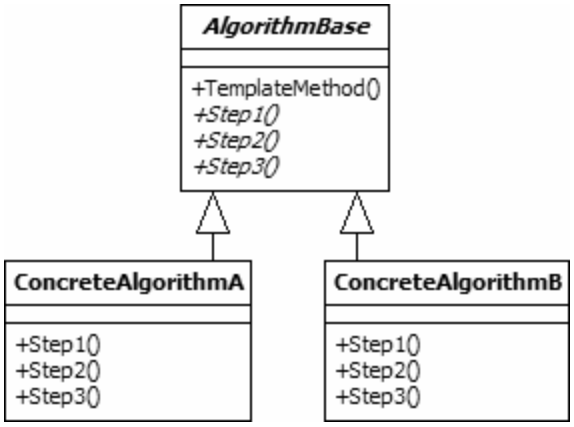
March 17, 2014

TABLE OF CONTENTS

1. Template Design Pattern 4
1.1 Applied Example – Football Game Simulator 4
1.1.1 Original Design 4
1.1.2 Improved Design Using Template Pattern 10
2. Adapter Design Pattern 17
2.1 Applied Example – “Easter Egg” in Football Game 17
2.1.1 Original Design 17
2.1.2 Improved Design Using Adapter Pattern 22
3. Iterator Design Pattern 26
3.1 Applied Example – Iterating Athletic Conferences 26
3.1.1 Original Design 26
3.1.2 Improved Design Using Iterator Pattern 31
4. Composite Design Pattern 37
4.1 Applied Example – Revenue Reporting from Athletic Conferences 37
4.1.1 Original Design 37
4.1.2 Improved Design Using Composite Pattern 42
Appendix A – *Non-Direct Activity Report* 49

Topics Covered	Topic Examples
Design Patterns	• Template • Adapter • Iterator • Composite

1. Template Design Pattern



1.1 Applied Example – Football Game Simulator

This example stems from the application presented in the previous project demonstrating the Strategy Pattern. This application will simulate the behavior of a football team while playing offense. This could be used for artificial intelligence during video game and other related applications. While this is mostly the same simulation, the offensive behavior is enhanced to provide more detail and characteristics when the `IOffensiveBehavior::RunPlay` command is executed for a specific implementation of an offensive behavior.

When on offense, a football team transitions through different stages, or phases, depending on the type of offense that team is currently running. For instance, traditional offenses will form a huddle. During the huddle phase, players are substituted for certain formations and packages, and the play is delivered from the coaches through a player coming into the game or via headsets in the quarterback's helmet. Once that has happened and the players know what their assignment is, the offense transitions to the lineup phase. In this phase, the players lineup at their respective positions on the field and according to the rules (e.g. no more than 4 players are allowed off the line of scrimmage in traditional American football). Also, during this phase, a player can go into motion in order to reset their placement in the formation so as to gain a positional advantage against the defense. Once all of this has happened (and no penalties have happened), the team transitions into the final stage of the play by snapping the ball and executing the designed play. Conversely, a hurry-up offense will skip the huddle in the hopes of being able to execute more plays in a shorter amount of time.

Using the Template Pattern, this transition of states can be factored into a single template class allowing concrete subclasses to implement their own detailed behaviors of the offense. This also provides a simple mechanism to make the behavior of the offenses more robust without impacting the code of the current simulation. All of the changes will happen at the implementation of the offensive behaviors and not at the simulation or prime (`FootballTeam`) logic. The implementations presented here are for demonstration purposes. More realistic implementations would have interactions between an opponent football team and a monitoring officiating crew which would be responsible for calling and enforcing any penalties.

1.1.1 Original Design

The original design utilizes the Strategy Pattern to allow a football team to change their offensive behavior at run-time. For this simulation, a requirement was to make the behavior of the offenses more robust but still only requiring a call to the `FootballClass::RunPlay()` method. Because there are multiple behaviors, it is up to each implementation of `IOffensiveBehavior` to

implement their behaviors in such a way that they all mimic the same process of phase transition when running a play. However, this is problematic in that a mistake in how the methods are ordered in the concrete class's code could lead to erratic and unintended behavior which would obviously be a bug.

Simulator Code

This is the code driving the simulation. Note how the offensive behavior is changed at run-time based on the current conditions of the game.

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

   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);

   Console.WriteLine(
      "{0} is now behind 21-14 with only two minutes left to go!  Time to hurry up...",
      offense.Name);
   offense.OffensiveBehavior = new HurryUpOffense();
   RunOffensiveSeries(offense);
}

private static void RunOffensiveSeries(FootballTeam team)
{
   for (int i = 0; i < 4; i++)
   {
      team.RunOffensivePlay();
      Console.WriteLine("\t---------");
   }
}
```

FootballTeam Class

This class is the logical representation of a football team which is used by the simulation.

```
public class FootballTeam
{
   public int RosterSize { get; private set; }

   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();
   }
}
```

IOffensiveBehavior Interface

This interface defines the behavior that each concrete class should implement in order to be used as an offensive behavior for a `FootballTeam` object.

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

BalancedOffense Class

This is the concrete implementation for a balanced offense. These offenses are more traditional in the sense that they do huddle so that players can be swapped and plays can be sent in by the coaches. From there, they line up in their formation and execute the play. Note how the methods are placed without any real knowledge of the true order they should be in. Although this is the correct order (as will be seen in the improved design), simply moving ANY of the method calls up or down would lead to unintended results. Providing an abstract template class (Template Pattern) that this class (and others) could inherit from would (mostly) guarantee that the correct phases would get executed in the proper order.

```
public class BalancedOffense : IOffensiveBehavior
{
   private bool _timeForRunPlay = true;

   public void RunPlay()
   {
      SubstitutePlayers();
      GetPlaysFromCoaches();
      LineUpInFormation();
      SendPlayerInMotion();
      if (_timeForRunPlay)
      {
         RunBlock();
      }
      else
      {
```

```
            PassBlock();
        }
        _timeForRunPlay = !_timeForRunPlay;
    }

    private void PassBlock()
    {
        Console.WriteLine("\t Block for PASS...");
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void SendPlayerInMotion()
    {
        Console.WriteLine("\t Send player in motion...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }

    private void GetPlaysFromCoaches()
    {
        Console.WriteLine("\t Get the play from the coaches...");
    }

    private void SubstitutePlayers()
    {
        Console.WriteLine("\t Substitute players...");
    }
}
```

BallControlOffense Class

This is the concrete implementation for a run-based ball control offense. These offenses are more simplistic with their behavior since they only run the ball on every play. Note the difference in the number of methods used in the `RunPlay` method versus the number of methods used in the `BalancedOffense::RunPlay` method (above).

```
public class BallControlOffense : IOffensiveBehavior
{
    public void RunPlay()
    {
        SubstitutePlayers();
        LineUpInFormation();
        RunBlock();
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }
```

```
    private void SubstitutePlayers()
    {
        Console.WriteLine("\t Substitute players...");
    }
}
```

HurryUpOffense Class

This is the concrete implementation for a hurry-up offense. These offenses are more hectic in nature because the main goal is to attempt to run as many plays as possible within the allotted time. Because of the urgent nature of these offenses, they do not go through a huddle phase.

```
public class HurryUpOffense : IOffensiveBehavior
{
    private static Random _random = new Random();

    private static bool RandomBoolean
    {
        get { return Convert.ToBoolean(_random.Next(2)); }
    }

    public void RunPlay()
    {
        // Hurry-Up offenses do not huddle
        LineUpInFormation();
        SendPlayerInMotion();
        if (RandomBoolean)
        {
            RunBlock();
        }
        else
        {
            PassBlock();
        }
    }

    private void PassBlock()
    {
        Console.WriteLine("\t Block for PASS...");
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void SendPlayerInMotion()
    {
        Console.WriteLine("\t Send player in motion...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }
}
```

PassHappyOffense Class

This is the concrete implementation for a passing-based offense. Much akin to the `BallControlOffense`, these offenses are more simplistic with their behavior since they only pass the ball on every play.

```
public class PassHappyOffense : IOffensiveBehavior
{
   public void RunPlay()
   {
      SubstitutePlayers();
      LineUpInFormation();
      SendPlayerInMotion();
      PassBlock();
   }

   private void PassBlock()
   {
      Console.WriteLine("\t Block for PASS...");
   }

   private void SendPlayerInMotion()
   {
      Console.WriteLine("\t Send player in motion...");
   }

   private void LineUpInFormation()
   {
      Console.WriteLine("\t Line up in formation...");
   }

   private void SubstitutePlayers()
   {
      Console.WriteLine("\t Substitute players...");
   }
}
```

Output

```
***Pre-Template Pattern***
Starting the game.  The score is tied at zero...
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for RUN...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for RUN...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
```

```
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
[Offense] is now ahead 14-7!  Time to run some clock...
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
[Offense] is now behind 21-14 with only two minutes left to go!  Time to hurry up...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Line up in formation...
    Send player in motion...
    Block for RUN...
   ---------
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
```

1.1.2 Improved Design Using Template Pattern

To improve the design while still meeting the requirements, the Template Pattern is utilized here because of the algorithmic nature of the phases of the RunPlay method. By doing this, it ensures that the different phases of the play will be executed in the proper order by any classes that inherit from template abstract class (`OffensiveBehaviorTemplate`). While this does not eliminate

all possible order of operation errors, it does provide subclasses with better guidance on how their methods should be organized for processing.

OffensiveBehaviorTemplate Class (Algorithm Base)

This is the abstract template class from which all concrete implementations of offensive behavior will be derived. Because this class inherits from `IOffensiveBehavior`, any subclasses will be able to be used anywhere an instance of an `IOffensiveBehavior` is used. For the purposes of this application, the required method from the `IOffensiveBehavior` interface will also serve as the template method that orders the phases of offensive behaviors. When subclasses implement the abstract methods, they are implementing under the notion that they do not have to know what order those methods need to be called because the `OffensiveBehaviorTemplate` class handles the algorithm's processing.

```
public abstract class OffensiveBehaviorTemplate : IOffensiveBehavior
{
    /// <summary>
    /// Template algorithm used to implement the offensive behavior of football teams.
    /// </summary>
    public void RunPlay()
    {
        Huddle();
        LineUp();
        SnapBall();
    }

    /// <summary>
    /// Phase 1 - Huddle.
    /// </summary>
    protected abstract void Huddle();

    /// <summary>
    /// Phase 2 - Line up in formation.
    /// </summary>
    protected abstract void LineUp();

    /// <summary>
    /// Phase 3 - Snap the ball and execute the play.
    /// </summary>
    protected abstract void SnapBall();
}
```

BalancedOffense Class (Concrete Algorithm)

Fundamentally, there is not much difference between the implementations of this class, except that now, the method calls have been refactored into the abstract template methods provided by `OffensiveBehaviorTemplate`. It should be noted that while the Template Pattern has helped to alleviate concern about order of operation errors, there still exists potential for order of operation errors due to the way that they class has been implemented.

```
public class BalancedOffense : OffensiveBehaviorTemplate
{
    private bool _timeForRunPlay = true;

    protected override void Huddle()
    {
        SubstitutePlayers();
        GetPlaysFromCoaches();
    }
```

```
    protected override void LineUp()
    {
        LineUpInFormation();
        SendPlayerInMotion();
    }

    protected override void SnapBall()
    {
        if (_timeForRunPlay)
        {
            RunBlock();
        }
        else
        {
            PassBlock();
        }
        _timeForRunPlay = !_timeForRunPlay;
    }

    private void PassBlock()
    {
        Console.WriteLine("\t Block for PASS...");
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void SendPlayerInMotion()
    {
        Console.WriteLine("\t Send player in motion...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }

    private void GetPlaysFromCoaches()
    {
        Console.WriteLine("\t Get the play from the coaches...");
    }

    private void SubstitutePlayers()
    {
        Console.WriteLine("\t Substitute players...");
    }
}
```

BallControlOffense Class (Concrete Algorithm)

Concrete implementation of the `BallControlOffense` class.

```
public class BallControlOffense : OffensiveBehaviorTemplate
{
    protected override void Huddle()
    {
        SubstitutePlayers();
    }

    protected override void LineUp()
```

```
    {
        LineUpInFormation();
    }

    protected override void SnapBall()
    {
        RunBlock();
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }

    private void SubstitutePlayers()
    {
        Console.WriteLine("\t Substitute players...");
    }
}
```

HurryUpOffense Class (Concrete Algorithm)

Concrete implementation of the `HurryUpOffense` class. Note how the implementation of the template-provided `Huddle` method does nothing since hurry-up offenses never go through a huddle phase in a traditional sense.

```
class HurryUpOffense : OffensiveBehaviorTemplate
{
    private static Random _random = new Random();

    private static bool RandomBoolean
    {
        get { return Convert.ToBoolean(_random.Next(2)); }
    }

    protected override void Huddle()
    {
        // Do nothing
    }

    protected override void LineUp()
    {
        LineUpInFormation();
        SendPlayerInMotion();
    }

    protected override void SnapBall()
    {
        if (RandomBoolean)
        {
            RunBlock();
        }
        else
        {
            PassBlock();
        }
    }
```

```
    private void PassBlock()
    {
        Console.WriteLine("\t Block for PASS...");
    }

    private void RunBlock()
    {
        Console.WriteLine("\t Block for RUN...");
    }

    private void SendPlayerInMotion()
    {
        Console.WriteLine("\t Send player in motion...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }
}
```

PassHappyOffense Class (Concrete Algorithm)

Concrete implementation of the `PassHappyOffense` class.

```
public class PassHappyOffense : OffensiveBehaviorTemplate
{
    protected override void Huddle()
    {
        SubstitutePlayers();
    }

    protected override void LineUp()
    {
        LineUpInFormation();
        SendPlayerInMotion();
    }

    protected override void SnapBall()
    {
        PassBlock();
    }

    private void PassBlock()
    {
        Console.WriteLine("\t Block for PASS...");
    }

    private void SendPlayerInMotion()
    {
        Console.WriteLine("\t Send player in motion...");
    }

    private void LineUpInFormation()
    {
        Console.WriteLine("\t Line up in formation...");
    }

    private void SubstitutePlayers()
    {
        Console.WriteLine("\t Substitute players...");
```

```
   }
}
```

Output

```
***Template Pattern Applied***
Starting the game.  The score is tied at zero...
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for RUN...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for RUN...
   ---------
    Substitute players...
    Get the play from the coaches...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
    Substitute players...
    Line up in formation...
    Send player in motion...
    Block for PASS...
   ---------
[Offense] is now ahead 14-7!  Time to run some clock...
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
    Substitute players...
    Line up in formation...
    Block for RUN...
   ---------
    Substitute players...
    Line up in formation...
    Block for RUN...
```

```
    ---------
     Substitute players...
     Line up in formation...
     Block for RUN...
    ---------
[Offense] is now behind 21-14 with only two minutes left to go!  Time to hurry up...
     Line up in formation...
     Send player in motion...
     Block for PASS...
    ---------
     Line up in formation...
     Send player in motion...
     Block for PASS...
    ---------
     Line up in formation...
     Send player in motion...
     Block for RUN...
    ---------
     Line up in formation...
     Send player in motion...
     Block for PASS...
    ---------
```

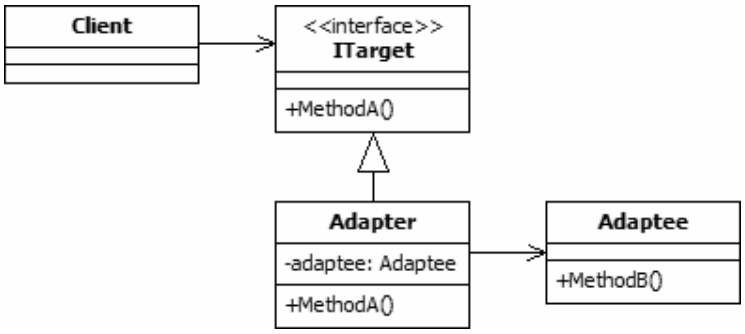
Advantages

- Removes the responsibility from the concrete class for knowing how the algorithm is supposed to be implemented. The concrete class implements the template methods and the abstract class handles the order in which the methods are called.
- If the algorithm ever changes such that the steps of execution are arranged in a different order, it is possible that only the abstract template class would be the only thing required to change involving potentially no changes to the concrete subclasses.

Disadvantages

- Each concrete class of the abstract template class could potentially be duplicating code.
- The intention of the template class could potentially be communicated poorly to subclasses if there are non-private methods and properties on the abstract template class that have little to do with the algorithm that the template class is intended to provide.

2. Adapter Design Pattern



2.1 Applied Example – "Easter Egg" in Football Game

Often times, video games will put in secret features that provide the user with special abilities for the game they are playing. These secret features are commonly called "Easter eggs" due to the similar nature of the concepts. For this section, the football game simulator used in other sections will be altered to provide the ability to play with a football team made up of a team's mascot. When playing as a mascot team, the user would have the ability to run explosive plays for every play.

2.1.1 Original Design

The initial design introduces an `IMascot` type which provides the interface for all mascots to implement. The goal is to be able to provide a mascot team to the `RunSim(IFootballTeam team)` method so that a simulation can be performed for a mascot team. The expected behavior would have the mascot team performing an explosive play every time `IFootballTeam::RunOffensivePlay` is executed.

Ideally, no changes would be required on the concrete `FootballTeam` class while the only noticeable changes would be where the simulation initializes the mascot team and passes it to the `RunSim(IFootballTeam team)` method.

IMascot Interface

These are the interfaces that will be implemented by any mascot objects. The `IEasterEggTeam` interface provides the ability for teams to behave according to an "Easter egg" team. The `IMascot` interface provides the basic methods required for mascots to behave like traditional mascots.

```
public interface IEasterEggTeam
{
   void RunUnstoppablePlay();
}

public interface IMascot : IEasterEggTeam
{
   string Name { get; }
   void CheerForTeam();
}
```

Mascot Class (Adaptee)

This is the concrete implementation of the `IMascot` interface. These objects are the logical representation of traditional team mascots used for the purposes of this simulation. Using these objects as a football team allows the user to run nothing but explosive plays on offense. However,

because it does not implement the `IFootballTeam` interface, a new type or a new constructor on the `FootballTeam` class will have to be created in order to use mascots as `IFootballTeam` types.

```
public class Mascot : IMascot
{
   private IFootballTeam _associatedTeam;
   private IOffensiveBehavior _offensiveBehavior;

   public string Name { get; private set; }

   public Mascot(IFootballTeam footballTeam)
   {
      _associatedTeam = footballTeam;
      Name = string.Format("{0}, the Juggernaut", _associatedTeam.Name);
      _offensiveBehavior = new EasterEggOffense();
   }

   public void CheerForTeam()
   {
      var msg = string.Format("{0} says: YAY! Go Juggernauts!!!", Name);
      Console.WriteLine(msg);
   }

   // Easter Egg offensive behavior.
   public void RunUnstoppablePlay()
   {
      _offensiveBehavior.RunPlay();
   }
}
```

EasterEggOffense Class

This class provides the special behavior for Easter egg teams. Changing a team's `IOffensiveBehavior` to this type will allow them to always run explosive plays.

```
public class EasterEggOffense : IOffensiveBehavior
{
   public void RunPlay()
   {
      Console.WriteLine("Ran an **UNSTOPPABLE** play...");
   }
}
```

IFootballTeam Interface (ITarget)

This is the interface that is accepted by the `RunSim(IFootballTeam team)` method. Any object that implements this interface can be run through the simulation procedures. So, for this application, any mascot team will need to find a way to behave the same as an `IFootballTeam`.

```
public interface IFootballTeam
{
   string Name { get; }
   int RosterSize { get; }
   IOffensiveBehavior OffensiveBehavior { get; set; }
   IMascot Mascot { get; }
   void RunOffensivePlay();
}
```

IOffensiveBehavior Interface

Same as seen in the other sections.

Other Implementations of IOffensiveBehavior

```
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...");
   }
}
```

FootballTeam Class

This is the concrete implementation of the `IFootballTeam` interface. An object of type `IMascot` has been added to represent the mascot for the team. A more realistic implementation would use a factory to determine – based on the supplied `IFootballTeam` – the matching mascot, but for this example, a simple constructor uses the instance of the `FootballTeam` to construct the mascot.

In order to make it possible for mascots to be created as `IFootballTeams`, a new constructor was added that accepts `IMascot` types. This does allow `IMascot` types to be created as `IFootballTeam` types. Unfortunately, this required a change to the concrete `FootballTeam` class, which was hopefully to be avoided.

```
public class FootballTeam : IFootballTeam
{
   public int RosterSize { get; private set; }

   private string _name;

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

   public IOffensiveBehavior OffensiveBehavior { get; set; }

   public IMascot Mascot { get; private set; }

   private FootballTeam()
```

```
    {
    }

    public FootballTeam(IMascot mascot)
        : this(mascot.Name, 11)
    {
        Mascot = mascot;
        OffensiveBehavior = new EasterEggOffense();
    }

    public FootballTeam(string name, int size)
    {
        Name = name;
        RosterSize = size;
        Mascot = new Mascot(this);
    }

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

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

Original Simulator Code:

As seen here, the only change to the simulator code was to create an instance of a mascot-based `FootballTeam` object in order to pass it to the `RunSim` method. Unfortunately, by implementing the mascot-based team as a `FootballTeam` object, the implementation of that class allows the `IOffensiveBehavior` to change at run-time which also eliminates the Easter egg-based offensive behavior of Easter egg teams. Since this goes against an original requirement, this solution is not acceptable.

```
public class AdapterPatternSim
{
    public static void Run()
    {
        Console.WriteLine("***Pre-Adapter Pattern***");

        var offense = new FootballTeam("Offense", new BalancedOffense(), 11);
        RunSim(offense);

        //RunSim(offense.Mascot); --> ERROR!  Mascot is not an IFootballTeam!!

        // This works but it required a change to the concrete FootballTeam class to create
        // a new constructor that accepts an IMascot type.  Also, by using the concrete
        // implementation of FootballTeam, it allows the OffensiveBehavior to change at
        // run-time.  While that is ideal for typical FootballTeam objects, that does not
        // bode well for the concept of "Easter Egg" teams.  The offensive behavior should
        // never be allowed to change away from an unstoppable offensive behavior (such as
        // EasterEggOffense).
        var mascotTeam = new FootballTeam(offense.Mascot);
        RunSim(mascotTeam);
    }
```

```
    private static void RunSim(IFootballTeam team)
    {
        Console.WriteLine("Starting the game.  The score is tied at zero...");
        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);
    }

    private static void RunOffensiveSeries(IFootballTeam team)
    {
        for (int i = 0; i < 2; i++)
        {
            team.RunOffensivePlay();
        }
        team.Mascot.CheerForTeam();
        Console.WriteLine("------");
    }
}
```

Output:

```
***Pre-Adapter Pattern***
Starting the game.  The score is tied at zero...
Ran a running play...
Ran a passing play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[Offense] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
Starting the game.  The score is tied at zero...
Ran an **UNSTOPPABLE** play...
Ran an **UNSTOPPABLE** play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[[Offense], the Juggernaut] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[[Offense], the Juggernaut] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
```

2.1.2 Improved Design Using Adapter Pattern

To improve upon the faults of the original design, the **Adapter Pattern** is used to create an adapter class that inherits from `IFootballTeam` and also accepts an `IMascot` type in its constructor. As the `IFootballTeam` methods are called on the adapter, it delegates those calls to the `IMascot` object.

This design prevents any changes from being made to the concrete `FootballTeam` class while also allowing the `IOffensiveBehavior` of mascot-based teams to retain their Easter egg behavior on offense.

MascotTeamAdapter Class (Adapter)

This adapter class acts as a wrapper for `IMascot` objects so that they can be accepted as `IFootballTeam` types for anything expecting an `IFootballTeam` type as an input.

Because this implements the `IFootballTeam` interface, it provides a getter/setter for its `IOffensiveBehavior`. In the original design, this caused a problem by allowing the Easter egg behavior to be overwritten. Here, any gets/sets done on the `IOffensiveBehavior` are simply ignored as the call to RunOffensivePlay will always run the implementation of this `IMascot`'s `RunUnstoppablePlay()` method.

```
public class MascotTeamAdapter : IFootballTeam
{
   public string Name { get; private set; }
   public int RosterSize { get; private set; }
   public IOffensiveBehavior OffensiveBehavior { get; set; }
   public IMascot Mascot { get; private set; }

   private MascotTeamAdapter()
   {
   }

   public MascotTeamAdapter(IMascot mascot)
      : this(mascot.Name, 11)
   {
      Mascot = mascot;
   }

   private MascotTeamAdapter(string name, int size)
   {
      Name = name;
      RosterSize = size;
   }

   public void RunOffensivePlay()
   {
      Mascot.RunUnstoppablePlay();
   }
}
```

IFootballTeam Interface (ITarget)

No change.

Other Supporting Interfaces

No change.

Implementations of IOffensiveBehavior

No change.

FootballTeam Class

Even though this is listed as a change, other than adding an `IMascot` object to `IFootballTeam` types, and by removing the highlighted code below, this is the original form of the `FootballTeam` class. This satisfies one of the original requirements by not requiring unnecessary changes to the `FootballTeam` class.

```
public class FootballTeam : IFootballTeam
{
   public int RosterSize { get; private set; }

   private string _name;

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

   public IOffensiveBehavior OffensiveBehavior { get; set; }

   public IMascot Mascot { get; private set; }

   private FootballTeam()
   {
   }

   //public FootballTeam(IMascot mascot)
   //   : this(mascot.Name, 11)
   //{
   //   Mascot = mascot;
   //   OffensiveBehavior = new EasterEggOffense();
   //}

   public FootballTeam(string name, int size)
   {
      Name = name;
      RosterSize = size;
      Mascot = new Mascot(this);
   }

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

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

Mascot Class (Adaptee)

No Change.

Updated Simulator Code (Client)

The change here is a subtle yet important change. Instead of creating the mascot-based team as an instance of `FootballTeam` , it is created as an instance of `MascotTeamAdapter`. In doing this, any changes to the `IFootballTeam::OffensiveBehavior` property get ignored when the `IFootballTeam` is an Easter egg-based team (or mascot-based team in this instance) thereby preserving the "special" nature of the offensive behavior of Easter egg teams.

```
public class AdapterPatternSim_Applied
{
   public static void Run()
   {
      Console.WriteLine("***Adapter Pattern Applied***");

      var offense = new FootballTeam("Offense", new BalancedOffense(), 11);
      RunSim(offense);

      var mascotTeam = new MascotTeamAdapter(offense.Mascot);
      RunSim(mascotTeam);
   }

   private static void RunSim(IFootballTeam team)
   {
      Console.WriteLine("Starting the game.  The score is tied at zero...");
      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);
   }

   private static void RunOffensiveSeries(IFootballTeam team)
   {
      for (int i = 0; i < 2; i++)
      {
         team.RunOffensivePlay();
      }
      team.Mascot.CheerForTeam();
      Console.WriteLine("------");
   }
}
```

Output

```
***Adapter Pattern Applied***
Starting the game.  The score is tied at zero...
Ran a running play...
Ran a passing play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[Offense] is now behind 7-0!  Time to get aggressive with the playcalling...
Ran a passing play...
Ran a passing play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
```

```
------
[Offense] is now ahead 14-7!  Time to run some clock...
Ran a running play...
Ran a running play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
Starting the game.  The score is tied at zero...
Ran an **UNSTOPPABLE** play...
Ran an **UNSTOPPABLE** play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[Offense], the Juggernaut is now behind 7-0!  Time to get aggressive with the playcalling...
Ran an **UNSTOPPABLE** play...
Ran an **UNSTOPPABLE** play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
[Offense], the Juggernaut is now ahead 14-7!  Time to run some clock...
Ran an **UNSTOPPABLE** play...
Ran an **UNSTOPPABLE** play...
[Offense], the Juggernaut says: YAY! Go Juggernauts!!!
------
```

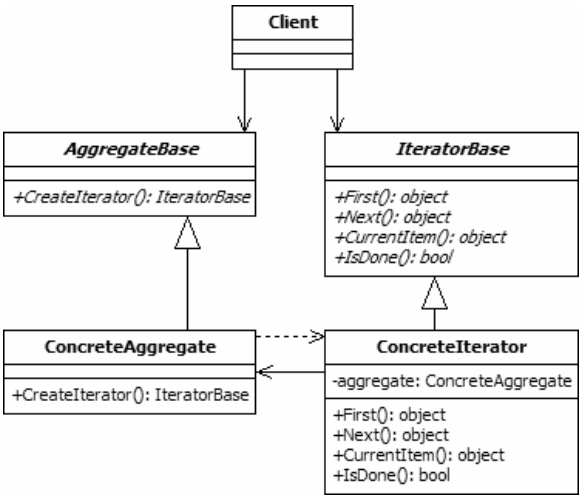
Advantages:

- With respect to a 2-tier (or more) system, when the server requires a specific interface implementation that the client cannot provide, an adapter can provide a solution without requiring changes to client or server.
- Allows previously incompatible types to work together after they have been designed and/or implemented.

Disadvantages

- Using adapters adds complexity to code and makes for potentially troublesome debugging sessions.
- Adds a dependency on the interface of the Adaptee (`IMascot`) at the level of where ever the Adapter (`MascotTeamAdapter`) is implemented.

3. Iterator Design Pattern



3.1 Applied Example – Iterating Athletic Conferences

This application will allow types implementing the same interface, but with different underlying implementation features, to be traversed and iterated in the same manner no matter the details of implementation. For this example, the simulation will build a collection of objects representing athletic teams within an athletic conference. The conference is the top-level object of the hierarchy that provides a container and methods to the objects – representing the athletic teams – to be added to the conference.

3.1.1 Original Design

The main idea is for client code to be presented with a general mechanism by which it can traverse different enumerable types in the same manner and with the same code. For this initial design, standard concrete classes representing athletic conferences are presented to a `ConferenceDisplayController` for basic output of member associations of the conference. The `SecConference` implements its container using an `IList<Team>` while the `AccConference` implements its container using a `Dictionary<string, Team>`. Because these two implementations use different containers, it is impossible to use the same code to traverse each object in the container.

Also, because the containers are different, the client code using these classes must be aware of such differences in order to properly treat each container. It would be best if the containers could be treated generically and without the need for prior knowledge of the implementation details of the concrete classes.

TeamColors Struct

A simple C# struct used to represent an organization's affiliated colors.

```
public struct TeamColors
{
   public Color Primary;
   public Color Secondary;

   public TeamColors(Color primary, Color secondary)
   {
      Primary = primary;
      Secondary = secondary;
   }
```

```
   public override string ToString()
   {
      var primary = ConvertColorStringToColorNameOnly(Primary);
      var secondary = ConvertColorStringToColorNameOnly(Secondary);
      return string.Format("{0} / {1}", primary, secondary);
   }

   private string ConvertColorStringToColorNameOnly(Color color)
   {
      return color.ToString().Split(' ').Last().Trim("[]".ToCharArray());
   }
}
```

Original Team Class

This is the logical representation of athletic teams that can be affiliated with the athletic conferences.

```
public class Team
{
   public string Name { get; private set; }
   public string Nickname { get; private set; }
   public TeamColors Colors { get; private set; }

   private Team()
   {
   }

   public Team(string name, string nickname, TeamColors colors)
   {
      Name = name;
      Nickname = nickname;
      Colors = colors;
   }

   public override string ToString()
   {
      return string.Format("{0} {1} - Team colors: {2}.", Name, Nickname, Colors);
   }
}
```

Original Concrete Classes Representing Athletic Conferences

These classes are the logical representations of athletic conferences. Two instances are created here: one (`AccConference`) utilizes a `Dictionary<string, Team>` and the other (`SecConference`) utilizes an `IList<Team>`. Even though their behaviors are similar in nature, the differences in their respective implementations cause enough problems for client code to not be able to treat them generically (which is what is desired).

```
/// <summary>
/// Represents the collection of teams from the ACC.  Underlying collection implemented using
/// a dictionary, where the key is an abbreviated name of the team and the value is the team.
/// </summary>
public class AccConference
{
   public Dictionary<string, Team> Teams { get; private set; }

   public string Name { get; private set; }

   private AccConference()
   {
```

```
    }

    public AccConference(string name)
    {
        Name = name;
        Teams = new Dictionary<string, Team>
            {
                {
                    "FSU", new Team("Florida St.", "Seminoles",
                                    new TeamColors(Color.Firebrick, Color.Gold))
                },
                {
                    "GT", new Team("Georgia Tech", "Yellow Jackets",
                                   new TeamColors(Color.Gold, Color.White))
                },
                {
                    "WF",
                    new Team("Wake Forest", "Demon Deacons",
                             new TeamColors(Color.Gold, Color.Black))
                }
            };
    }

    public void AddTeam(KeyValuePair<string, Team> kvp)
    {
        if (Teams.ContainsKey(kvp.Key) || Teams.ContainsValue(kvp.Value))
        {
            Console.WriteLine("You cannot add {0} because it already exists in the " +
                              "ACC conference!", kvp.Value.Name);
            return;
        }
        Teams.Add(kvp.Key, kvp.Value);
    }

    // etc.
}

/// <summary>
/// Represents the collection of teams from the SEC.  The underlying collection has been
/// implemented using a generic list of Team objects.
/// </summary>
public class SecConference
{
    private List<Team> _teams;

    public IList<Team> Teams
    {
        get { return _teams.AsReadOnly(); }
        private set { _teams = value.ToList(); }
    }

    public string Name { get; private set; }

    private SecConference()
    {
    }

    public SecConference(string name)
    {
        Name = name;
        _teams = new List<Team>
            {
```

```
            new Team("Alabama", "Crimson Tide",
                    new TeamColors(Color.Crimson, Color.White)),
            new Team("Georgia", "Bulldogs", new TeamColors(Color.Red, Color.Black)),
            new Team("Vanderbilt", "Commodores",
                    new TeamColors(Color.Gold, Color.Black))
        };
    }

    public void AddTeam(KeyValuePair<string, Team> kvp)
    {
        // Ignore the key and just add the value.
        _teams.Add(kvp.Value);
    }

    // etc.
}
```

ConferenceDisplayController Class (Client)

This is the client code who is most concerned with using the instances of the `SecConference` and `AccConference`. As it stands now, the `ConferenceDisplayController` must know details of how both instances are implemented in order to properly use them. This creates a dependency that should not be relegated to this code. Also, because these conference objects do not inherit from any interface, they cannot be treated generically, so if any new conferences are added, in order for the `ConferenceDisplayController` class to handle them appropriately, change to the `ConferenceDisplayController` class will have to be made. Ideally, these could be passed to the `ConferenceDisplayController` class as a collection and the client code could act on them generically.

```
public class ConferenceDisplayController
{
    private SecConference _sec;
    private AccConference _acc;

    public ConferenceDisplayController(SecConference sec, AccConference acc)
    {
        _sec = sec;
        _acc = acc;
    }

    public void PrintTeams()
    {
        var secTeams = _sec.Teams;
        var accTeams = _acc.Teams;

        var tab = "\t";
        Console.WriteLine("{0} Teams:", _sec.Name);

        // Client code must know that this is a List<>
        for (int i = 0; i < secTeams.Count; i++)
        {
            Console.WriteLine(tab + secTeams[i]);
        }

        Console.WriteLine("{0} Teams:", _acc.Name);

        // Client code must know that this is a Dictionary<>
        foreach (var key in accTeams.Keys)
        {
            Console.WriteLine(tab + accTeams[key]);
        }
```

```
   }
}
```

Original Simulator Code

This is the original simulation code. Several problems exist with this implementation in that it is possible to write code that successfully compiles, yet crashes at run-time. The highlighted comments give more details into these problems. Ideally, these instances of conferences would implement the same interface allowing them to be treated generically.

The standard execution of this simulation builds the conference membership of each conference, and then passes that to the client (`ConferenceDisplayController`), which displays the members of each conference.

```
public class IteratorPatternSim
{
   public static void Run()
   {
      Console.WriteLine("***Pre-Iterator Pattern***");

      var secConference = new SecConference("SEC");
      //var secTeams = secConference.Teams;
      // The following line throws a run-time exception b/c the list is returned as a
      // read-only list!!  The client code has no indication from the compiler that this is
      // a problem!
      //secTeams.Add(new Team("Kentucky", "Wildcats",
      //                      new TeamColors(Color.Blue, Color.White)));
      var team = new Team("Kentucky", "Wildcats",
                          new TeamColors(Color.Blue, Color.White));
      secConference.AddTeam(new KeyValuePair<string, Team>(team.Name, team));
      secConference.AddTeam(new KeyValuePair<string, Team>(team.Name, team));

      var accConference = new AccConference("ACC");
      var accTeams = accConference.Teams;
      // The following line throws a run-time exception b/c the dictionary already contains
      // this key (and team).  In order to avoid this exception, the client must know
      // which teams have been added during object initialization.
      //accTeams.Add("GT", new Team("Georgia Tech", "Yellow Jackets",
      //                            new TeamColors(Color.Gold, Color.White)));
      team = new Team("Georgia Tech", "Yellow Jackets",
                      new TeamColors(Color.Gold, Color.White));
      accConference.AddTeam(new KeyValuePair<string, Team>("GT", team));

      accTeams.Add("UNC",
                   new Team("North Carolina", "Tarheels",
                            new TeamColors(Color.LightSkyBlue, Color.White)));

      var confDisplayController = new ConferenceDisplayController(secConference,
                                                                   accConference);
      confDisplayController.PrintTeams();
   }
}
```

Output

```
***Pre-Iterator Pattern***
You cannot add Georgia Tech because it already exists in the ACC conference!
SEC Teams:
   Alabama Crimson Tide - Team colors: Crimson / White.
   Georgia Bulldogs - Team colors: Red / Black.
   Vanderbilt Commodores - Team colors: Gold / Black.
```

```
    Kentucky Wildcats - Team colors: Blue / White.
    Kentucky Wildcats - Team colors: Blue / White.
ACC Teams:
    Florida St. Seminoles - Team colors: Firebrick / Gold.
    Georgia Tech Yellow Jackets - Team colors: Gold / White.
    Wake Forest Demon Deacons - Team colors: Gold / Black.
    North Carolina Tarheels - Team colors: LightSkyBlue / White.
```

3.1.2 Improved Design Using Iterator Pattern

In the previous design, there were numerous problems ranging from unnecessary dependencies on implementation details by the client code to rigid design. By applying the **Iterator Pattern**, these dependencies on implementation details are removed and the design becomes accommodating to new conferences that may be added at a later time.

ConferenceTeamsEnumerator (Concrete Iterator)

This class is the concrete iterator that will be used by the client code (`ConferenceDisplayController`) to treat the conference objects as generic and apply the same code to them for any implementation requiring traversal of the underlying containers. This class implements the .NET-provided `IEnumerator` interface which provides the necessary behavior required for collection traversal. It also has the added benefit of automatically working with foreach-loops (and any other .NET types that require `IEnumerator` types).

As seen below, it does not matter how each conference implements its underlying container, just that it implements the `IEnumerator` interface. By doing this, any client code is no longer required to know the implementation details; the client code can treat any object implementing the `IEnumerator` interface generically.

```
public class ConferenceTeamsEnumerator : IEnumerator
{
   private IList<Team> _teams;
   private int _position = -1;

   private ConferenceTeamsEnumerator()
   {
   }

   public ConferenceTeamsEnumerator(IDictionary<string, Team> teams)
   {
      _teams = teams.Values.ToList();
   }

   public ConferenceTeamsEnumerator(IList<Team> teams)
   {
      _teams = teams;
   }

   public bool MoveNext()
   {
      _position++;
      return _position < _teams.Count;
   }

   public void Reset()
   {
      _position = -1;
   }

   object IEnumerator.Current
```

```
   {
      get { return Current; }
   }

   public Team Current
   {
      get
      {
         try
         {
            return _teams[_position];
         }
         catch (IndexOutOfRangeException)
         {
            throw new InvalidOperationException();
         }
      }
   }
}
```

IAthleticConference Interface (Aggregate Base)

This is the interface that all concrete aggregate objects (`AccConference`, `SecConference`) must implement in order to be properly used by the client (`ConferenceDisplayController`). This interface also inherits from the .NET-provided `IEnumerable` interface which provides all the necessary behavior required by objects that need to provide a custom or .NET-provided `IEnumerator` type.

```
public interface IAthleticConference : IEnumerable
{
   string Name { get; }
   void AddTeam(KeyValuePair<string, Team> kvp);
}
```

Implementations of IAthleticConference (Concrete Aggregate)

These classes are the concrete implementations of the `IAthleticConference` interface and also represent the concrete aggregate in the iterator design pattern. The most noticeable difference with this version is that these classes now provide a method (`GetEnumerator`) that allows its iterator (or enumerator) to be accessed from any client code for container traversal. Now, it no longer matters to the client code how the underlying containers are implemented, just that the clients can access the enumerator.

```
/// <summary>
/// Represents the collection of teams from the ACC.  Underlying collection implemented using
/// a dictionary, where the key is an abbreviated name of the team and the value is the team.
/// </summary>
public class AccConference : IAthleticConference
{
   private Dictionary<string, Team> _teams;

   public string Name { get; private set; }

   private AccConference()
   {
   }

   public AccConference(string name)
   {
      Name = name;
      _teams = new Dictionary<string, Team>
```

```
        {
            {
                "FSU", new Team("Florida St.", "Seminoles",
                                new TeamColors(Color.Firebrick, Color.Gold))
            },
            {
                "GT", new Team("Georgia Tech", "Yellow Jackets",
                               new TeamColors(Color.Gold, Color.White))
            },
            {
                "WF",
                new Team("Wake Forest", "Demon Deacons",
                         new TeamColors(Color.Gold, Color.Black))
            }
        };
    }

    public void AddTeam(KeyValuePair<string, Team> kvp)
    {
        if (_teams.ContainsKey(kvp.Key) || _teams.ContainsValue(kvp.Value))
        {
            Console.WriteLine("You cannot add {0} because it already exists in the " +
                              "ACC conference!", kvp.Value.Name);
            return;
        }
        _teams.Add(kvp.Key, kvp.Value);
    }

    public IEnumerator GetEnumerator()
    {
        return new ConferenceTeamsEnumerator(_teams);
    }

    // etc.
}

/// <summary>
/// Represents the collection of teams from the SEC.  The underlying collection has been
/// implemented using a generic list of Team objects.
/// </summary>
public class SecConference : IAthleticConference
{
    private List<Team> _teams;

    public string Name { get; private set; }

    private SecConference()
    {
    }

    public SecConference(string name)
    {
        Name = name;
        _teams = new List<Team>
            {
                new Team("Alabama", "Crimson Tide",
                         new TeamColors(Color.Crimson, Color.White)),
                new Team("Georgia", "Bulldogs", new TeamColors(Color.Red, Color.Black)),
                new Team("Vanderbilt", "Commodores",
                         new TeamColors(Color.Gold, Color.Black))
            };
    }
```

```
   public void AddTeam(KeyValuePair<string, Team> kvp)
   {
      // Ignore the key and just add the value.
      _teams.Add(kvp.Value);
   }

   public IEnumerator GetEnumerator()
   {
      return new ConferenceTeamsEnumerator(_teams);
   }

   // etc.
}
```

Team Class (Object)

No change.

TeamColors Struct

No change.

ConferenceDisplayController Class (Client)

This version of `ConferenceDisplayController` now accepts a collection of `IAthleticConference` types and uses the `GetEnumerator` method on each one to get its iterator/enumerator and use generic code to print out each conference's members. This addresses the rigid design from the previous solution and requires no change to this class if more conferences are added in the future.

```
public class ConferenceDisplayController
{
   private readonly IList<IAthleticConference> AthleticConferences;

   public ConferenceDisplayController(IList<IAthleticConference> athleticConferences)
   {
      AthleticConferences = athleticConferences;
   }

   public void PrintTeams()
   {
      foreach (var athleticConference in AthleticConferences)
      {
         PrintTeams(athleticConference as IAthleticConference);
      }
   }

   private static void PrintTeams(IAthleticConference conference)
   {
      var tab = "\t";
      Console.WriteLine("{0} Teams:", conference.Name);

      // Using custom iterator to traverse objects even though they are contained in
      // different types of containers.
      var confTeamsEnumerator = conference.GetEnumerator();
      while (confTeamsEnumerator.MoveNext())
      {
         Console.WriteLine(tab + confTeamsEnumerator.Current);
      }

      // Could have just as easily gone with the following code to do the same thing:
```

```
      //foreach (var team in conference)
      //{
      //   Console.WriteLine(tab + team);
      //}
   }
}
```

Updated Simulator Code

Because an iterator/enumerator is now provided by the concrete `IAthleticConference` classes, the underlying container is no longer required to be exposed, so the only way to add (and remove if it were necessary) is through the methods/properties supplied by the `IAthleticConference` interface (such as `AddTeam`). This prevents unexpected results that were seen as run-time errors in the previous solution and this provides common behavior from the client's perspective (even though the underlying implementation is different).

```
public class IteratorPatternSim_Applied
{
   public static void Run()
   {
      Console.WriteLine("***Iterator Pattern Applied***");

      var secConference = new SecConference("SEC");
      var team = new Team("Kentucky", "Wildcats",
                          new TeamColors(Color.Blue, Color.White));
      secConference.AddTeam(new KeyValuePair<string, Team>(team.Name, team));
      secConference.AddTeam(new KeyValuePair<string, Team>(team.Name, team));

      var accConference = new AccConference("ACC");
      team = new Team("Georgia Tech", "Yellow Jackets",
                      new TeamColors(Color.Gold, Color.White));
      accConference.AddTeam(new KeyValuePair<string, Team>("GT", team));

      team = new Team("North Carolina", "Tarheels",
                      new TeamColors(Color.LightSkyBlue, Color.White));
      accConference.AddTeam(new KeyValuePair<string, Team>("UNC", team));

      var conferences = new List<IAthleticConference> {secConference, accConference};

      var confDisplayController = new ConferenceDisplayController(conferences);
      confDisplayController.PrintTeams();
   }
}
```

Output

```
***Iterator Pattern Applied***
You cannot add Georgia Tech because it already exists in the ACC conference!
SEC Teams:
    Alabama Crimson Tide - Team colors: Crimson / White.
    Georgia Bulldogs - Team colors: Red / Black.
    Vanderbilt Commodores - Team colors: Gold / Black.
    Kentucky Wildcats - Team colors: Blue / White.
    Kentucky Wildcats - Team colors: Blue / White.
ACC Teams:
    Florida St. Seminoles - Team colors: Firebrick / Gold.
    Georgia Tech Yellow Jackets - Team colors: Gold / White.
    Wake Forest Demon Deacons - Team colors: Gold / Black.
    North Carolina Tarheels - Team colors: LightSkyBlue / White.
```

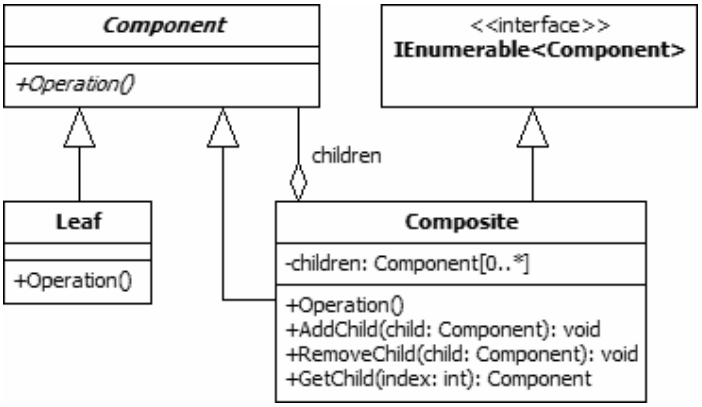
Advantages

- Hides implementation details of how the Aggregate class (`IAthleticConference`) stores its objects, while still allowing traversal of the objects.
- Removes the responsibility of object traversal from the Aggregate class (`IAthleticConference`) and places it with the iterator object (`ConferenceTeamsEnumerator`).

Disadvantages

- If done carelessly, implementing a custom iterator could lead to buggy code with several memory mismanagement issues.

4. Composite Design Pattern



4.1 Applied Example – Revenue Reporting from Athletic Conferences

This application is a continuation from the final solution provided in section 3. It adds the ability for each team and conference to report their revenues at the conference-level down to the sport-level in a tree-like format. This is accomplished using the Composite Design pattern.

4.1.1 Original Design

As previously mentioned, the original design solution was a continuation from the final solution provided in section 3. The biggest difference being that the custom iterator was replaced with a simple call to the .NET-provided `GetEnumerator` method that is available on all .NET containers.

The primary problem encountered with this design is that it does not offer a viable solution to the problem of being able to add "sub-conferences" to conferences. For instance, as revenues are reported, the desire is that the revenues could be shown at different levels of granularity. Requesting revenues at the conference level should show all revenues for all members and all of their sports individually. Athletic conferences have their member schools, and each member school has men's and women's sports (where each could be considered a sub-conference containing their own sports). Then there are different sports for men than there are women (i.e. football for men, softball and gymnastics for women). Then some sports can contain several sub-teams (such as Track & Field who may have a sprinting team, long-distance team, etc.). Simply put, without writing complex code, this design simply offers no viable solution for the given requirements.

Interfaces

The biggest difference here is that a new interface, `IReportRevenues`, has been added for providing the necessary behavior required by conferences and teams for reporting their revenues. Also, what was formerly known as the `IAthleticConference` is now the `IAthleticTeamConference` and has added a property (`Colors`) for affiliating colors with types that implement the `IAthleticTeamConference` interface.

```
public interface IReportRevenues
{
   int Revenues { get; }
}

public interface IAthleticTeamConference : IReportRevenues, IEnumerable
{
   string Name { get; }
   TeamColors Colors { get; }
```

```
   void AddTeam(KeyValuePair<string, Team> team);
}
```

Original Team Class

This is essentially the same class with the only change being the added property and updated message required to report its revenues.

```
public class Team : IReportRevenues
{
   public string Name { get; private set; }
   public string Nickname { get; private set; }
   public TeamColors? Colors { get; set; }
   public int Revenues { get; private set; }

   private Team()
   {
      Revenues = 100;
   }

   public Team(string name, string nickname, TeamColors? colors = null)
      : this()
   {
      Name = name;
      Nickname = nickname;
      Colors = colors;
   }

   public override string ToString()
   {
      return string.Format("{0} {1} reported ${2} in revenues for the {3}.",
                           Name, Nickname, Revenues, Colors);
   }
}
```

TeamColors Struct

No change from implementation in previous section.

TeamConference Abstract Class

This represents the biggest change from the solution in section 3 as all common features and implementation required by the concrete `IAthleticTeamConference` classes have been "brought up" into an abstract class for the concrete types to inherit from. This class also makes use of the Template Design pattern (`AddTeamWithColors`) for the default behavior required to add teams to the conference and the Factory Method Pattern (`GetEnumerator`) for the behavior for fetching an objects `IEnumerator`.

The biggest problem with this design solution does exist here in the `AddTeam` method. Since this method only supports the addition of `Team` objects, `TeamConference` types cannot be added as "sub-conferences" which prevents this design from being a viable solution for the given requirements.

```
public abstract class TeamConference : IAthleticTeamConference
{
   public string Name { get; protected set; }
   public TeamColors Colors { get; protected set; }
   public abstract int Revenues { get; }

   protected TeamConference()
   {
```

```
   }

   protected TeamConference(string name, TeamColors colors)
   {
      Name = name;
      Colors = colors;
   }

   public void AddTeam(KeyValuePair<string, Team> team)
   {
      // As a default behavior, for any teams without specified colors, assign them the
      // conference's colors.
      if (!team.Value.Colors.HasValue)
      {
         team.Value.Colors = new TeamColors(Colors);
      }
      AddTeamWithColors(team);
   }

   protected abstract void AddTeamWithColors(KeyValuePair<string, Team> team);

   public abstract IEnumerator GetEnumerator();
}
```

Concrete Implementations of TeamConference

These are the concrete implementations of the `TeamConference` abstract class. They are essentially the same as before except for revenue reporting features and using .NET's `GetEnumerator` method on its underlying container (as opposed to returning the custom enumerator that was seen in the section 3 solution).

```
/// <summary>
/// Represents the collection of teams from the ACC.  Underlying collection implemented using
/// a dictionary, where the key is an abbreviated name of the team and the value is the team.
/// </summary>
public class AccConference : TeamConference
{
   private readonly Dictionary<string, Team> _teams;

   public override int Revenues
   {
      get { return _teams.Values.Aggregate(0, (current, team) => current + team.Revenues); }
   }

   private AccConference()
   {
   }

   public AccConference(string name, TeamColors colors)
      : base(name, colors)
   {
      _teams = new Dictionary<string, Team>
         {
            {
               "FSU", new Team("Florida St.", "Seminoles",
                               new TeamColors(Color.Firebrick, Color.Gold))
            },
            {
               "GT", new Team("Georgia Tech", "Yellow Jackets",
                              new TeamColors(Color.Gold, Color.White))
            },
            {
```

```
                "WF",
                new Team("Wake Forest", "Demon Deacons",
                         new TeamColors(Color.Gold, Color.Black))
            }
        };
    }

    protected override void AddTeamWithColors(KeyValuePair<string, Team> team)
    {
        if (_teams.ContainsKey(team.Key) || _teams.ContainsValue(team.Value))
        {
            Console.WriteLine("You cannot add {0} because it already exists in the " +
                              "ACC conference!", team.Value.Name);
            return;
        }
        _teams.Add(team.Key, team.Value);
    }

    public override IEnumerator GetEnumerator()
    {
        return _teams.Values.GetEnumerator();
    }

    // etc.
}

/// <summary>
/// Represents the collection of teams from the SEC.  The underlying collection has been
/// implemented using a generic list of Team objects.
/// </summary>
public class SecConference : TeamConference
{
    private List<Team> _teams;

    public override int Revenues
    {
        get { return _teams.Aggregate(0, (current, team) => current + team.Revenues); }
    }

    private SecConference()
    {
    }

    public SecConference(string name, TeamColors colors)
        : base(name, colors)
    {
        _teams = new List<Team>
            {
                new Team("Alabama", "Crimson Tide",
                         new TeamColors(Color.Crimson, Color.White)),
                new Team("Georgia", "Bulldogs", new TeamColors(Color.Red, Color.Black)),
                new Team("Vanderbilt", "Commodores",
                         new TeamColors(Color.Gold, Color.Black))
            };
    }

    protected override void AddTeamWithColors(KeyValuePair<string, Team> team)
    {
        // Ignore the key and just add the value.
        _teams.Add(team.Value);
    }
```

```
   public override IEnumerator GetEnumerator()
   {
      return _teams.GetEnumerator();
   }

   // etc.
}
```

Original ConferenceDisplayController Class

This is the same implementation as before except for a new message for reporting revenues.

```
public class ConferenceDisplayController
{
   private readonly IList<IAthleticTeamConference> AthleticConferences;

   public ConferenceDisplayController(IList<IAthleticTeamConference> athleticConferences)
   {
      AthleticConferences = athleticConferences;
   }

   public void PrintTeams()
   {
      foreach (var athleticConference in AthleticConferences)
      {
         PrintTeams(athleticConference);
      }
   }

   private static void PrintTeams(IAthleticTeamConference conference)
   {
      var tab = "\t";
      Console.WriteLine("{0} Teams had total revenues of ${1}:",
                        conference.Name, conference.Revenues);

      foreach (var team in conference)
      {
         Console.WriteLine(tab + team);
      }
   }
}
```

Original Simulator Code

For a single level container solution, this is a good solution. For the purposes of this application, however, this is not a good solution due to the need for the ability to have multiple levels in the container (e.g. a tree-like container).

```
public class CompositePatternSim
{
   public static void Run()
   {
      Console.WriteLine("***Pre-Composite Pattern***");

      var secConference = new SecConference("SEC",
                                            new TeamColors(Color.DarkBlue, Color.Yellow));
      var team = new Team("Kentucky", "Wildcats",
                          new TeamColors(Color.Blue, Color.White));
      secConference.AddTeam(new KeyValuePair<string, Team>(team.Name, team));

      var accConference = new AccConference("ACC",
                                            new TeamColors(Color.Blue, Color.LightGray));
```

```
      team = new Team("North Carolina", "Tarheels",
                      new TeamColors(Color.LightSkyBlue, Color.White));
      accConference.AddTeam(new KeyValuePair<string, Team>("UNC", team));


      var conferences = new List<IAthleticTeamConference> {secConference, accConference};
      var confDisplayController = new ConferenceDisplayController(conferences);
      confDisplayController.PrintTeams();
   }
}
```

Output

```
***Pre-Composite Pattern***
SEC Teams had total revenues of $400:
   Alabama Crimson Tide reported $100 in revenues for the Crimson and White.
   Georgia Bulldogs reported $100 in revenues for the Red and Black.
   Vanderbilt Commodores reported $100 in revenues for the Gold and Black.
   Kentucky Wildcats reported $100 in revenues for the Blue and White.
ACC Teams had total revenues of $400:
   Florida St. Seminoles reported $100 in revenues for the Firebrick and Gold.
   Georgia Tech Yellow Jackets reported $100 in revenues for the Gold and White.
   Wake Forest Demon Deacons reported $100 in revenues for the Gold and Black.
   North Carolina Tarheels reported $100 in revenues for the LightSkyBlue and White.
```

4.1.2 Improved Design Using Composite Pattern

To address the given requirements, the original design was re-implemented using the **Composite Pattern**. This allows for multiple levels in the container made up of node (`TeamConference`) and leaf (`Team`) objects. This is accomplished by providing an interface that both the node and the leaf inherit from. All nodes in the tree are the composite objects, meaning they can contain other nodes or leaves, whereas leaves are considered the end of the "branch" in the container without the ability to contain other objects.

TeamConferenceComponent Class (Component)

This is the Component object in the Composite Pattern. This class provides the common interface for both the node and leaf objects to inherit from. The methods and properties are grouped into "composite" and "operation" where each is most likely to be implemented by the composite and leaf objects, respectively.

```
public abstract class TeamConferenceComponent
{
   protected TeamConferenceComponent()
   {
   }

   protected TeamConferenceComponent(string name, string surname, TeamColors? colors)
   {
      Name = name;
      Surname = surname;
      Colors = colors;
   }

   #region Composite Methods/Properties

   public TeamConferenceComponent AddComponent(
      TeamConferenceComponent teamConferenceComponent)
   {
      if (!teamConferenceComponent.Colors.HasValue)
      {
         teamConferenceComponent.Colors = new TeamColors(Colors.Value);
```

```
        }
        if (string.IsNullOrWhiteSpace(teamConferenceComponent.Name))
        {
            teamConferenceComponent.Name = Name;
        }

        // The only way that the colors could be null or the name be empty is if the
        // component is a Team object.
        return AddTeam(teamConferenceComponent);
    }

    protected virtual TeamConferenceComponent AddTeam(TeamConferenceComponent team)
    {
        throw new NotImplementedException();
    }

    public virtual TeamConferenceComponent this[int index]
    {
        get { throw new NotImplementedException(); }
    }

    #endregion

    #region Operation Methods/Properties

    public string Name { get; protected set; }
    public TeamColors? Colors { get; protected set; }
    public string Surname { get; protected set; }

    public abstract int Revenues { get; protected set; }
    public abstract override string ToString();

    #endregion
}
```

TeamConference Class (Composite)

This is the Composite object in the Composite Pattern. This class is considered a node in the tree which means it can contain other node and/or leaf objects. The `AddTeam` method was slightly altered in this solution so that a chain of calls could be made in such a way that it mimicked a tree structure when written out in code.

```
public class TeamConference : TeamConferenceComponent
{
    private readonly List<TeamConferenceComponent> _components;

    public override int Revenues
    {
        get
        {
            return _components.Aggregate(0,
                                         (current, component) => current + component.Revenues);
        }
        protected set { throw new ApplicationException(); }
    }

    private TeamConference()
    {
    }

    public TeamConference(string name)
        : this(name, new TeamColors(Color.Black, Color.Black))
```

```
    {
    }

    public TeamConference(string name, TeamColors colors, string surname = null)
        : base(name, surname, colors)
    {
        _components = new List<TeamConferenceComponent>();
    }

    protected override TeamConferenceComponent AddTeam(TeamConferenceComponent team)
    {
        _components.Add(team);
        return this;
    }

    public override TeamConferenceComponent this[int index]
    {
        get { return _components[index]; }
    }

    public override string ToString()
    {
        var result = new StringBuilder();

        var msg = (string.IsNullOrWhiteSpace(Surname))
                    ? string.Format("{0} Teams had total revenues of ${1}:",
                                    Name, Revenues)
                    : string.Format("{0} {1} Teams had total revenues of ${2}:",
                                    Name, Surname, Revenues);

        result.AppendLine(msg);
        foreach (var component in _components)
        {
            result.Append(TabSpaces).Append(component);
            Tabs--;
        }
        return result.ToString();
    }

    private static int Tabs;

    private static string TabSpaces
    {
        get
        {
            var result = new StringBuilder();
            var tab = "\t";
            Tabs++;
            for (int i = 0; i < Tabs; i++)
            {
                result.Append(tab);
            }
            return result.ToString();
        }
    }
}
```

Updated Team Class (Leaf)

This is the Leaf object in the Composite Pattern. This is outermost level of the container tree. These objects cannot contain other components (unlike composite, or node, objects).

```
public sealed class Team : TeamConferenceComponent
{
   public override int Revenues { get; protected set; }

   private Team()
   {
   }

   public Team(string surname, string name = null, TeamColors? colors = null)
      : base(name, surname, colors)
   {
      Surname = surname;
      Revenues = 100;
   }

   public override string ToString()
   {
      var msg = string.Format("{0} {1} reported ${2} in revenues for the {3}.",
                              Name, Surname, Revenues, Colors);
      return new StringBuilder().AppendLine(msg).ToString();
   }
}
```

TeamColors Struct

No change.

ConferenceDisplayController Class (Client)

Due to the re-implementation and new design, the `ConferenceDisplayController` has been greatly simplified and altered to accept the Component (`TeamConferenceComponent`) object. This means that the `ConferenceDisplayController` class can accept either a Composite (`TeamConference`) or a Leaf (`Team`) object and respond accordingly. Also, because the Composite and Leaf objects are the same type, the `ConferenceDisplayController` class can treat them generically when displaying them. All of the work is done by the Component objects.

```
/// <summary>
/// Client code that actually uses the Composite objects.
/// </summary>
public class ConferenceDisplayController
{
   private TeamConferenceComponent _allTeams;

   public ConferenceDisplayController(TeamConferenceComponent allTeams)
   {
      _allTeams = allTeams;
   }

   public void PrintTeams()
   {
      Console.Write(_allTeams);
   }
}
```

Updated Simulator Code

The simulation was changed to take advantage of the new `AddComponent` method which accepts `Team` or `TeamConference` objects and can keep calling AddComponent in such a way that is visually depicts the manner in which the components are stored in the tree. The only disadvantage to this is if a mistake is made by trying to add a component to a `Team` object (which is a Leaf object), then an exception would be thrown and it could be difficult to track down.

Beyond that, the rest of the simulation is similar to before; the teams are loaded into the conferences (some with “sub-conferences” and some with none) and the `ConferenceDisplayController` class prints them out in a tree-like manner while revealing their revenues at each level.

```
public class CompositePatternSim_Applied
{
   private static void InitializeTeams(TeamConferenceComponent teamConference)
   {
      teamConference
         .AddComponent(new TeamConference("Men's", teamConference.Colors.Value, "Sports")
                           .AddComponent(new Team("Baseball"))
                           .AddComponent(new Team("Basketball"))
                           .AddComponent(new Team("Football")))
         .AddComponent(new TeamConference("Women's", teamConference.Colors.Value, "Sports")
                           .AddComponent(new Team("Gymnastics"))
                           .AddComponent(new Team("Softball"))
                           .AddComponent(new TeamConference("Women's",
                                                            teamConference.Colors.Value,
                                                            "Olympic Sports")
                                                 .AddComponent(new Team("Swimming"))
                                                 .AddComponent(new Team("Runners"))));
   }

   private static TeamConference InitializeComponents()
   {
      var allConferences = new TeamConference("NCAA");

      // Add conferences
      var secConference = new TeamConference("SEC",
                                             new TeamColors(Color.DarkBlue, Color.Yellow));
      var accConference = new TeamConference("ACC",
                                             new TeamColors(Color.Blue, Color.LightGray));
      allConferences.AddComponent(secConference);
      allConferences.AddComponent(accConference);

      // Add SEC teams
      var alabama = new TeamConference("Alabama", new TeamColors(Color.Crimson, Color.White),
                                       "Crimson Tide");
      InitializeTeams(alabama);
      secConference.AddComponent(alabama);
      secConference.AddComponent(new TeamConference("Georgia",
                                                    new TeamColors(Color.Red, Color.Black),
                                                    "Bulldogs").AddComponent(
                                                       new Team("Football")));
      secConference.AddComponent(new TeamConference("Kentucky",
                                                    new TeamColors(Color.Blue, Color.White),
                                                    "Wildcats").AddComponent(
                                                       new Team("Basketball")));
      secConference.AddComponent(new TeamConference("Vanderbilt",
                                                    new TeamColors(Color.Gold, Color.Black),
                                                    "Commodores"));
      // Add ACC teams
      accConference.AddComponent(new TeamConference("Florida St.",
                                                    new TeamColors(Color.Firebrick,
                                                                   Color.Gold),
                                                    "Seminoles").AddComponent(
                                                       new Team("Football")));
      accConference.AddComponent(new TeamConference("Georgia Tech",
                                                    new TeamColors(Color.Gold, Color.White),
                                                    "Yellow Jackets"));
```

```
        var carolinaColors = new TeamColors(Color.LightSkyBlue, Color.White);
        accConference
           .AddComponent(new TeamConference("North Carolina", carolinaColors, "Tarheels")
                         .AddComponent(new TeamConference("Women's", carolinaColors,
                                                          "Sports")
                                                .AddComponent(new Team("Soccer"))
                                                .AddComponent(new Team("Basketball"))));
        accConference.AddComponent(new TeamConference("Wake Forest",
                                                      new TeamColors(Color.Gold, Color.Black),
                                                      "Demon Deacons"));
        return allConferences;
    }

    public static void Run()
    {
        Console.WriteLine("***Composite Pattern Applied***");

        var allConferences = InitializeComponents();

        // Print all schools.
        var confDisplayController = new ConferenceDisplayController(allConferences);
        confDisplayController.PrintTeams();

        Console.WriteLine("------");

        // Print ONLY ACC schools.
        confDisplayController = new ConferenceDisplayController(allConferences[1]);
        confDisplayController.PrintTeams();
    }
}
```

Output

```
***Composite Pattern Applied***
NCAA Teams had total revenues of $1200:
   SEC Teams had total revenues of $900:
      Alabama Crimson Tide Teams had total revenues of $700:
         Men's Sports Teams had total revenues of $300:
            Men's Baseball reported $100 in revenues for the Crimson and White.
            Men's Basketball reported $100 in revenues for the Crimson and White.
            Men's Football reported $100 in revenues for the Crimson and White.
         Women's Sports Teams had total revenues of $400:
            Women's Gymnastics reported $100 in revenues for the Crimson and White.
            Women's Softball reported $100 in revenues for the Crimson and White.
            Women's Olympic Sports Teams had total revenues of $200:
               Women's Swimming reported $100 in revenues for the Crimson and White.
               Women's Runners reported $100 in revenues for the Crimson and White.
      Georgia Bulldogs Teams had total revenues of $100:
         Georgia Football reported $100 in revenues for the Red and Black.
      Kentucky Wildcats Teams had total revenues of $100:
         Kentucky Basketball reported $100 in revenues for the Blue and White.
      Vanderbilt Commodores Teams had total revenues of $0:
   ACC Teams had total revenues of $300:
      Florida St. Seminoles Teams had total revenues of $100:
         Florida St. Football reported $100 in revenues for the Firebrick and Gold.
      Georgia Tech Yellow Jackets Teams had total revenues of $0:
      North Carolina Tarheels Teams had total revenues of $200:
         Women's Sports Teams had total revenues of $200:
            Women's Soccer reported $100 in revenues for the LightSkyBlue and White.
            Women's Basketball reported $100 in revenues for the LightSkyBlue and White.
      Wake Forest Demon Deacons Teams had total revenues of $0:
------
ACC Teams had total revenues of $300:
```

```
Florida St. Seminoles Teams had total revenues of $100:
   Florida St. Football reported $100 in revenues for the Firebrick and Gold.
Georgia Tech Yellow Jackets Teams had total revenues of $0:
North Carolina Tarheels Teams had total revenues of $200:
   Women's Sports Teams had total revenues of $200:
      Women's Soccer reported $100 in revenues for the LightSkyBlue and White.
      Women's Basketball reported $100 in revenues for the LightSkyBlue and White.
Wake Forest Demon Deacons Teams had total revenues of $0:
```

Advantages

- Uses recursion to accomplish common tasks.
- The Component's (`TeamConferenceComponent`) interface provides a convenient mechanism for client code to interact with the Composite (`TeamConference`) objects.

Disadvantages

- Lose underlying custom container implementation from original implementation that maybe has already been extensively tested.
- Both leaf (`Team`) and composite (`TeamConference`) classes must potentially inherit methods/properties from the component (`TeamConferenceComponent`) class that they will have no interest in using or implementing.
- Using recursion can lead to difficult debugging sessions and potential memory management and performance issues.

Appendix A – *Non-Direct Activity Report*

Date	Duration (minutes)	Specific Task / Activity
10-Feb-2014	29	Updating preparing activity sheet for next activity period.
22-Feb-2014	113	Research for project #2
23-Feb-2014	68	Project #2 setup/initialization.
2-Mar-2014	210	Research & write code/report for Project #2
9-Mar-2014	362	Research & write code/report for Project #2
10-Mar-2014	545	Research & write code/report for Project #2
16-Mar-2014	720	Research & write code/report for Project #2
Sum for Report #2	**2047**	**/ 1500 (5 weeks @ 300/wk)**