SSE 658

Design Problems and Solutions

Project #3

by

Jason Payne

April 25, 2014

TABLE OF CONTENTS

Topics Covered	Topic Examples
Design Patterns	• Null Object • Service Locator / Inversion of Control • Multiton • Memento

Introduction: The GPS Route Planning System (RPS)

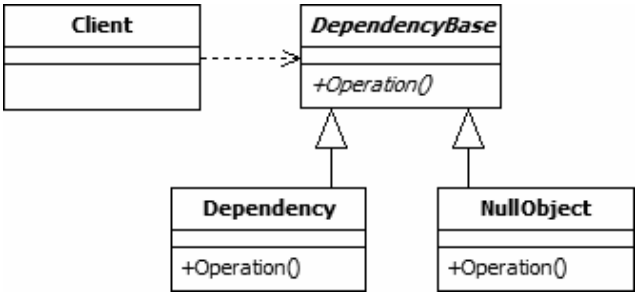
The primary purpose of this project was to illustrate different design patterns – namely, the Null Object, Service Locator, Multiton, and Memento Patterns – within a single system. So for the purposes of this project an application was chosen that would allow users to plan and create routes to be used in a GPS, or similarly functioning, device. The main idea of the system is for it to allow users to create a route by dropping points on a map (graphical) and/or listing points in a data grid (text-based). At the end of their session the user could export the route to their GPS device and/or save the route for later use. For the sake of this project, this system will be referred to as the Route Planning System or "RPS" for short.

The architecture of the system supports plug-in components by providing interfaces (see `IGpsComponent` in Appendix A) to third party developers who wish to extend the capabilities of the system. Since the system is a new application, there needs to be a default planning component in place for users that simply wish to use the RPS out-of-the-box. This default implementation will be developed in the same way as a plug-in to the system which also allows as a good test for the RPS and its framework. In industry, this would be packaged together and defined as the "complete" system, but third party developers would implement their plug-ins in much the same way as the default planning component will be implemented here. Due to the scope of the application and time restrictions, the focus of this system will be on the implementation details of the text-based planning feature.

The project will be presented in Agile-inspired "sprints" where each sprint implements a new feature by utilizing and illustrating a specific design pattern. Each sprint will build off the previous sprint (as it would in industry) and highlight the deficiencies of the current state of the code as it relates to the specifications of the system. The specifications of the system are listed below:

ID	Specifications
1	The system shall allow users to plan their routes in a text-based format such as a data grid. The grid should support blank rows between points (much like Microsoft's Excel application).
2	The system shall use the supported point type color preferences provided by the system framework.
3	As color preferences change, the preferences shall apply to all subsequent route plans opened. Any pre-existing plans will not have their color preferences altered and will continue to use the color preferences that were in place when that plan was created.
4	The system shall support undo operations.

Sprint 1: Null Object Design Pattern



1.1 Applied Example – Blank Data Grid Rows

For the first sprint, the first requirement to address was the ability to display blank (null) rows on a data grid. This would be useful for UI data-binding purposes. Having said that, it adds complexity to such implementations because collection objects do not natively support containment of null, or void, references. To overcome this constraint, the Null Object Pattern was utilized so that the null representation is an actual class object that has a "null implementation" as an empty object.

1.1.1 Implemented Solution

IRoutePoint Interface (DependencyBase)

This is the logical equivalent of the DependencyBase class in the class diagram above. This is a local interface and it provides an interface for objects to implement if they wish to be used by the route planner.The null object implementation must implement this interface to adequately stand in as a "nulled" route point.

```
public interface IRoutePoint
{
   string Id { get; }
   string Description { get; }
   string Lat { get; }
   string Lon { get; }
   Color Color { get; }
   bool IsBlank { get; }
}
```

NullRoutePoint Class (NullObject)

This is the logical equivalent of the NullObject class in the class diagram above. It represents the concrete implementation of the DependencyBase class (`IRoutePoint`). This class will always return true when accessing its `IsBlank` property. This class allows empty rows to be presented in the route planner's data grid view without worrying about null reference exceptions.

```
public class NullRoutePoint : IRoutePoint
{
   public string Id { get; private set; }
   public string Description { get; private set; }
   public string Lat { get; private set; }
   public string Lon { get; private set; }
   public Color Color { get; private set; }

   public bool IsBlank
   {
      get { return true; }
   }
```

```
   public NullRoutePoint()
   {
      Id = "<Blank>";
      Description = Lat = Lon = string.Empty;
      Color = Color.Transparent;
   }

   public override string ToString()
   {
      return string.Format("{0}\t{1}\t\t\t{2}\t{3}\t\t<{4}>",
                           Id,
                           Description,
                           Lat,
                           Lon,
                           Color.Name);
   }
}
```

RoutePoint Class (Dependency)

This is the logical equivalent of the Dependency class in the class diagram above. This represents the "active" representation of `IRoutePoint`s when actual data is used in the row of a data grid. This implementation will always return false when accesing its `IsBlank` property.

It should also be noted that this class is dependent upon the use of the point database and color preferences supplied by the system framework. As it stands now, there is no way for this class to access the system framework in order to gain access to actual implementations of the data it needs from the framework, so it must create local instantiations of the dependencies it is relying on. This is not recommended because it may or may not be the same implementation that is currently provided by the framework. The next sprint will address these issues.

```
public class RoutePoint : IRoutePoint
{
   public string Id { get; private set; }
   public string Description { get; private set; }
   public string Lat { get; private set; }
   public string Lon { get; private set; }
   public Color Color { get; private set; }

   public bool IsBlank
   {
      get { return false; }
   }

   private RoutePointDB PointDatabase { get; set; }

   // Make these static so that they persist for any future RoutePoints that are created.
   private static IPreferences Preferences { get; set; }

   private RoutePoint()
   {
      // The route point database and preferences are provided by the framework, but there
      // is currently no way to access the framework.  So local copies must be created here.
      PointDatabase = new RoutePointDB();
      if (Preferences == null)
         Preferences = new Preferences(Color.Red, Color.Green, Color.Blue);
   }

   public RoutePoint(string id)
```

```
      : this()
   {
      InitializePointFromId(id);
   }

   private void InitializePointFromId(string id)
   {
      var result = PointDatabase.FindPointById(id);
      Id = result.Name;
      Description = result.Description;
      Lat = result.Lat.ToString(CultureInfo.InvariantCulture);
      Lon = result.Lon.ToString(CultureInfo.InvariantCulture);
      Color = Preferences[result.PointType];
   }

   public override string ToString()
   {
      return string.Format("{0}\t{1}\t\t{2}\t\t{3}\t\t<{4}>",
                           Id,
                           Description,
                           Lat,
                           Lon,
                           Color.Name);
   }
}
```

ICountable Interface

A local interface provided for objects that wish to provide a method for counting the actual non-blank instances in a collection. This is utilized here for providing the number of non-blank rows in the data grid.

```
public interface ICountable
{
   int Count { get; }
}
```

RoutePlannerTextView Class

This class is the logical representation of the text-based planning component of the RPS. This class contains the grid that should be able to contain blank rows in between active points. As the class is first constructed, it starts with 5 blank rows and allows new rows to be added at any row in the grid.

```
public class RoutePlannerTextView : IGpsTextView<IRoutePoint>, ICountable
{
   private List<IRoutePoint> _grid;
   public IList<IRoutePoint> Grid
   {
      get { return _grid.AsReadOnly(); }
   }

   public RoutePlannerTextView()
   {
      var size = 5;
      _grid = new List<IRoutePoint>();
      for (var i = 0; i < size; i++)
      {
         _grid.Add(new NullRoutePoint());
      }
   }
```

```
   public IRoutePoint AddPointById(string id, int index)
   {
      if ((index < 0) || (index >= _grid.Count))
      {
         var msg = string.Format("The text view cannot add items at index {0}", index);
         throw new IndexOutOfRangeException(msg);
      }
      var addedRoutePt = (string.IsNullOrEmpty(id))
                              ? new NullRoutePoint()
                              : new RoutePoint(id) as IRoutePoint;
      _grid[index] = addedRoutePt;
      return addedRoutePt;
   }

   /// <summary>
   /// Returns the size of the grid.
   /// </summary>
   public int Size { get { return _grid.Count; } }

   /// <summary>
   /// Returns the count of non-blank points in the grid.
   /// </summary>
   public int Count
   {
      get { return _grid.Count(pt => !pt.IsBlank); }
   }
}
```

IRoutePlanner Interface

This is a local interface used by objects that wish to provide a method that allows a point to be added by its identification string. For this implementation, the `RoutePlanner` class implements this interface as a means for adding points to its graphical and text-based planning components. This interface will also be used in a later sprint for providing proper context to a collection of service containers.

```
public interface IRoutePlanner
{
   // Local custom interface.
   void AddPointById(string id, int index);
}
```

RoutePlanner Class

This is the logical representation of the default route planner component that would be packaged and shipped with the RPS. By implementing `IGpsComponent`, this is the first object created by the framework when using the default route planner. It also acts as a plug-in to the RPS framework thereby extending the capabilities of the system. By implementing `IGpsDataObject`, this class indicates to the framework that it contains data that will be consumable in all contexts of use by this route planner. In a more complete version, this component would support creating/opening/saving/closing of instances of itself that would be handled through communications with the framework.

```
public class RoutePlanner : IGpsComponent, IGpsDataObject<IRoutePoint>, IRoutePlanner
{
   private RoutePlannerTextView _textView;

   public IGpsFramework Framework { get; set; }
   public IGpsMapView MapView { get; set; }
   public IGpsTextView<IRoutePoint> TextView
   {
```

```
      get { return _textView; }
      set { _textView = value as RoutePlannerTextView; }
   }

   public void Initialize()
   {
      _textView = new RoutePlannerTextView();

      // Initialize other fields here...
   }

   public void AddPointById(string id, int index)
   {
      // Add to text view...
      TextView.AddPointById(id, index);

      // Add to map view...
      // ...
   }
}
```

Simulation

In this simulation, the `RoutePlanner` is registered with and gets initialized by the mock RPS. Then, points are added to the grid (through the route planner) by providing an identifier and row number. A null row is also added to the grid by specifying a null reference for the identifier and at a location that is already occupied by an active point. This effectively acts as blanking a row and validates the support for null/blank rows.

```
public class NullObjectPatternSim
{
   public static void Run()
   {
      Console.WriteLine("***Null Object Pattern***");

      var mockRoutePlannerObj = new RoutePlanner();
      MockGpsRoutePlannerSystem.RegisterComponent(mockRoutePlannerObj);
      var textView = mockRoutePlannerObj.TextView;

      var i = 0;
      Console.WriteLine("{0}Step {1}: Start with an empty grid.", Environment.NewLine,
                        ++i);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in last row.", Environment.NewLine,
                        ++i);
      mockRoutePlannerObj.AddPointById("test", textView.Size - 1);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in first row.",
                        Environment.NewLine, ++i);
      mockRoutePlannerObj.AddPointById("MCN/A", 0);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User blanks the last row.",
                        Environment.NewLine, ++i);
      mockRoutePlannerObj.AddPointById(null, textView.Size - 1);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in middle row.",
                        Environment.NewLine, ++i);
```

```
      mockRoutePlannerObj.AddPointById("dob", textView.Size / 2);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in last row.", Environment.NewLine,
                        ++i);
      mockRoutePlannerObj.AddPointById("GQO", textView.Size - 1);
      PrintGrid(textView);
   }

   private static void PrintGrid(IGpsTextView<IRoutePoint> textView)
   {
      var header = new StringBuilder().Append("ID\t\t")
                                      .Append("Description\t\t\t")
                                      .Append("Lat\t\t")
                                      .Append("Lon\t\t\t")
                                      .Append("Color").AppendLine();
      header.Append("=====\t")
            .Append("===========\t\t\t")
            .Append("=====\t")
            .Append("=====\t\t")
            .Append("=====");
      Console.WriteLine(header.ToString());
      foreach (var routePoint in textView.Grid)
      {
         Console.WriteLine(routePoint);
      }

      // Display the number of points in the current grid.
      Console.WriteLine("=== Number of points in the grid: {0} ===",
                        (textView as ICountable).Count);
   }
}
```

Output

```
***Null Object Pattern***

Step 1: Start with an empty grid.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 0 ===

Step 2: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
.TEST                       0      0         <Transparent>
=== Number of points in the grid: 1 ===

Step 3: User enters data in first row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
```

```
<Blank>                                     <Transparent>
.TEST                       0      0        <Transparent>
=== Number of points in the grid: 2 ===

Step 4: User blanks the last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
=== Number of points in the grid: 1 ===

Step 5: User enters data in middle row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                     <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
=== Number of points in the grid: 2 ===

Step 6: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                     <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                     <Transparent>
GQO/W  Choo Choo            34.57  -85.09    <Blue>
=== Number of points in the grid: 3 ===
```

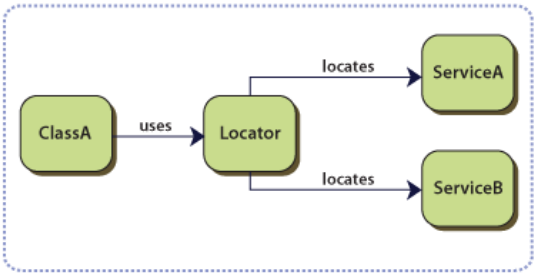
Advantages

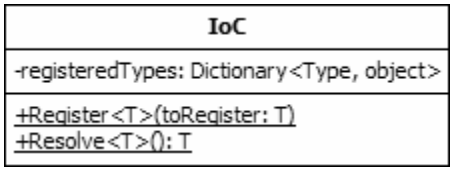
- Removes the need to wrap every possible null reference in try/catch statements, simplifying code.

Disadvantages

- Can hide errors in code where null reference exceptions would typically be throw, which leads to difficult debug/test sessions.

Sprint 2: Service Locator Design Pattern with Inversion of Control (IoC) Container





2.1 Applied Example – Color Preferences

As was noticed in the previous sprint, there is a problem in the `RoutePoint` class because it has a dependency on objects that are provided by the framework (color preferences and a point database), yet there is no framework reference made available to this class. To address this issue, the previous version would simply create its own local instances of the dependencies it needed. This is obviously not an ideal solution as the framework could have entirely different implementations of those dependencies and using local instances could create very confusing results.

This sprint addresses the issue by utilizing the ***Service Locator Pattern*** in conjunction with an ***Inversion of Control (IoC)*** container. By doing this, the IoC container can be accessed as a static instance (since the framework is static itself) which means that any class that has visibility to the `IoC` class can gain access to the services it provides. For this implementation, the framework is configured and placed in the IoC container when the `RoutePlanner` is initialized. Then as `RoutePoint` objects are created, they have access to the framework via the IoC container.

2.1.1 Implemented Solution

IoC Class (Service Locator / Inversion of Control Container)

This class serves as the Service Locator in the class diagram above. Services are added to the container using the `Register` method and dependent objects access those services by calling the `Resolve` method. Because this class is static, any classes with visibility to this class can access services by making static calls of the provided methods.

However, because it is static, as its services change, any classes accessing the services from the container will be getting potentially different values from the service objects. For services that should be persisted among the multiple, independent objects (such as color preferences), this solution will not work in its current state. The next sprint will address these issues.

```
public static class IoC
{
   private static Dictionary<Type, object> _registeredTypes = new Dictionary<Type, object>();

   public static void Register<T>(T toRegister)
   {
      _registeredTypes.Add(typeof (T), toRegister);
   }

   public static T Resolve<T>()
```

```
   {
      return (T) _registeredTypes[typeof (T)];
   }
}
```

RoutePoint Class (Service User)

As can be seen here, the `RoutePoint` class no longer has to instantiate its own local implementations of its dependent classes because it can access the IoC container and retrieve the services from it. Because the previous implementation created its own local instance of `IPreferences`, to maximize the efficiency of the code, the `Preferences` property was made static so that lazy loading with redundancy checking could be utilized from the constructor. Now, because the class has access to the framework, the `Preferences` property can be turned into an instance property allowing for persistence among different `RoutePoint` objects.

It should be noted that while this implementation does fulfill the requirement for getting preferences from the framework, this will not fulfill the other requirements if the preferences were ever changed. If a `RoutePoint` object in route planning session 1 is created with preferences X, and then the preferences are changed to preferences Y, when subsequent RoutePoint objects are created, they will be created with preferences Y which would lead to confusing behavior of the system (not to mention unfulfilled requirements). The next sprint will address these issues.

```
public class RoutePoint : IRoutePoint
{
   public string Id { get; private set; }
   public string Description { get; private set; }
   public string Lat { get; private set; }
   public string Lon { get; private set; }
   public Color Color { get; private set; }

   public bool IsBlank
   {
      get { return false; }
   }

   private RoutePointDB PointDatabase { get; set; }
   private /*static*/ IPreferences Preferences { get; set; }

   private RoutePoint()
   {
      PointDatabase = IoC.Resolve<IGpsFramework>().RoutePointDB;
      //if (Preferences == null)
         //Preferences = new Preferences(Color.Red, Color.Green, Color.Blue);
      Preferences = IoC.Resolve<IGpsFramework>().Preferences;
   }

   public RoutePoint(string id)
      : this()
   {
      InitializePointFromId(id);
   }

   private void InitializePointFromId(string id)
   {
      var result = PointDatabase.FindPointById(id);
      Id = result.Name;
      Description = result.Description;
      Lat = result.Lat.ToString(CultureInfo.InvariantCulture);
      Lon = result.Lon.ToString(CultureInfo.InvariantCulture);
      Color = Preferences[result.PointType];
```

```
   }

   public override string ToString()
   {
      return string.Format("{0}\t{1}\t\t{2}\t\t{3}\t\t<{4}>",
                           Id,
                           Description,
                           Lat,
                           Lon,
                           Color.Name);
   }
}
```

RoutePlanner Class

Since this is the main, top-level component that gets created first – by the RPS – at initialization of the route planning feature, any services required by its sub-components are registered at initialization. Some third-party IoC containers allow for configuration from text files that are parsed and translated into objects. Here, a simple configuration process is used by registering the framework with the container due to the dependency of the `RoutePoint` class on the framework.

```
public class RoutePlanner : IGpsComponent, IGpsDataObject<IRoutePoint>, IRoutePlanner
{
   private RoutePlannerTextView _textView;

   public IGpsFramework Framework { get; set; }
   public IGpsMapView MapView { get; set; }
   public IGpsTextView<IRoutePoint> TextView
   {
      get { return _textView; }
      set { _textView = value as RoutePlannerTextView; }
   }

   public void Initialize()
   {
      _textView = new RoutePlannerTextView();

      // Initialize other fields here...

      // Initialize services container
      IoC.Register(Framework);
   }

   public void AddPointById(string id, int index)
   {
      // Add to text view...
      TextView.AddPointById(id, index);

      // Add to map view...
      // ...
   }
}
```

IRoutePlanner Interface

No change. Refer to the latest `IRoutePlanner`.

ICountable Interface

No change. Refer to the latest `ICountable`.

RoutePlannerTextView Class

No change. Refer to the latest `RoutePlannerTextView`.

NullRoutePoint Class

No change. Refer to the latest `NullRoutePoint`.

IRoutePoint Interface

No change. Refer to the latest `IRoutePoint`.

Simulation

Because the changes to implement the Service Locator were internal in nature, no visible change in the simulation was required. Therefore, the same simulation from Sprint 1 was used here. Please refer to the Simulation used in Sprint 1 for further details.

Output

```
***Service Locator Pattern with IoC Container***

Step 1: Start with an empty grid.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 0 ===

Step 2: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
.TEST                       0      0         <Transparent>
=== Number of points in the grid: 1 ===

Step 3: User enters data in first row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
.TEST                       0      0         <Transparent>
=== Number of points in the grid: 2 ===

Step 4: User blanks the last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 1 ===

Step 5: User enters data in middle row.
ID     Description          Lat    Lon       Color
```

```
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 2 ===

Step 6: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                      <Transparent>
GQO/W  Choo Choo            34.57  -85.09    <Blue>
=== Number of points in the grid: 3 ===
```

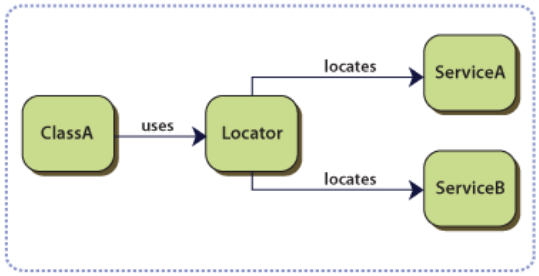
Advantages

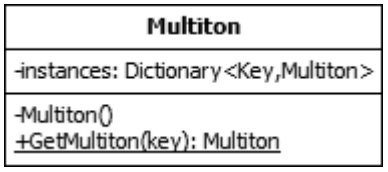
- Simplifies the process of getting access to dependencies in code without coupling the client to the concrete dependency.

Disadvantages

- Can be tougher to test the code b/c the dependencies are injected (from the IoC container) as they are needed in the code and not passed via Constructor Injection.

Sprint 3: Multiton Design Pattern





3.1 Applied Example – Persisted Color Preferences

In the previous sprint, it was noted that changes to preference could lead to unexpected behavior and unfulfilled requirements. To address these concerns, the ***Multiton Pattern*** was utilized. The Multiton is basically a collection of static objects (or Singletons). This provides a system with the capability to persist static data at different states based on a key. As the key changes, so does the context of the collection of static instances. For the purposes of this project, the Multiton was utilized so that the services container (IoC) could have data persisted across for multiple instances of route planning sessions. This was implemented solely for the purpose of persisting color preferences for each route planning object.

3.1.1 Implemented Solution

IoC Class (Multiton Container utilizing Microsoft's Unity Container)

Since creating a more robust IoC is beyond the scope of this project, a third party IoC container was used. Microsoft's Unity Container was used to replace the original IoC container seen in the previous sprint. This container is more robust and provides more built in configuration features that were useful for this sprint. Here, the Multiton is implemented using a `Dictionary` of `IUnityContainer` objects that are keyed by the `IRoutePlanner` interface. By keying off the `IRoutePlanner` interface, the context of the container can adjust as the active route planning object changes, therefore allowing the preferences to remain tied to the instance of the route plan object.

```
public static class IoC
{
    private static readonly Dictionary<IRoutePlanner, IUnityContainer> _instances =
        new Dictionary<IRoutePlanner, IUnityContainer>();

    private static readonly object _lock = new object();

    public static IUnityContainer CreateContainer(IRoutePlanner key)
    {
```

```
        lock (_lock)
        {
            if (!_instances.ContainsKey(key)) _instances.Add(key, new UnityContainer());
        }
        Container = _instances[key];
        return Container;
    }

    public static IUnityContainer UpdateContainer(IRoutePlanner key)
    {
        lock (_lock)
        {
            if (!_instances.ContainsKey(key)) throw new KeyNotFoundException();
        }
        Container = _instances[key];
        return Container;
    }

    public static IUnityContainer Container { get; set; }

    public static T Resolve<T>()
    {
        return TryResolve<T>();
    }

    private static T TryResolve<T>()
    {
        if (Container.IsRegistered<T>())
        {
            return Container.Resolve<T>();
        }
        return default(T);
    }

    public static void Dispose()
    {
        if (Container == null) return;
        Container.Dispose();
        Container = null;
    }

    public static void Dispose(IRoutePlanner key)
```

```
   {
      if ((_instances == null) || (key == null)) return;
      if (_instances.ContainsKey(key))
      {
         _instances[key].Dispose();
         _instances[key] = null;
         _instances.Remove(key);
      }
   }
}
```

RoutePoint Class (Service User)

The notable difference here is that the `Preferences` property is now set based on a call to resolve `IPreferences` from the IoC container (versus accessing the `Preferences` property off the resolved `IGpsFramework` object). It is done this way because the `RoutePlanner` class now configures the framework's preferences to be stored as an instance so that a snapshot (or clone) of the preferences is stored in the services container versus a reference (which would still change as the framework's preferences change).

```
public class RoutePoint : IRoutePoint
{
   public string Id { get; private set; }
   public string Description { get; private set; }
   public string Lat { get; private set; }
   public string Lon { get; private set; }
   public Color Color { get; private set; }

   public bool IsBlank
   {
      get { return false; }
   }

   private RoutePointDB PointDatabase { get; set; }
   private IPreferences Preferences { get; set; }

   private RoutePoint()
   {
      PointDatabase = IoC.Resolve<IGpsFramework>().RoutePointDB;
      //if (Preferences == null)
      //Preferences = new Preferences(Color.Red, Color.Green, Color.Blue);
      Preferences = IoC.Resolve<IPreferences>();
   }
```

```
   public RoutePoint(string id)
      : this()
   {
      InitializePointFromId(id);
   }

   private void InitializePointFromId(string id)
   {
      var result = PointDatabase.FindPointById(id);
      Id = result.Name;
      Description = result.Description;
      Lat = result.Lat.ToString(CultureInfo.InvariantCulture);
      Lon = result.Lon.ToString(CultureInfo.InvariantCulture);
      Color = Preferences[result.PointType];
   }

   public override string ToString()
   {
      return string.Format("{0}\t{1}\t\t{2}\t\t{3}\t\t<{4}>",
                           Id,
                           Description,
                           Lat,
                           Lon,
                           Color.Name);
   }
}
```

IRoutePlanner Interface

This interface now serves multiple purposes. In addition to providing the `AddPointById` method, it now also serves as the context key that is provided to the IoC container when the active route plan changes.

```
public interface IRoutePlanner
{
   // This interface is now also used to identify the context objects that are to be keys
   // for the IoC multiton container.
   void AddPointById(string id, int index);
}
```

RoutePlanner Class

Using the new capabilities of the redesigned IoC container, the `RoutePlanner` class now configures the IoC container with instances of the framework and the framework's preferences. This allows for persistence of preferences in route plan objects.

```
public class RoutePlanner : IGpsComponent, IGpsDataObject<IRoutePoint>, IRoutePlanner
{
   private RoutePlannerTextView _textView;

   public IGpsFramework Framework { get; set; }
   public IGpsMapView MapView { get; set; }
   public IGpsTextView<IRoutePoint> TextView
   {
      get { return _textView; }
      set { _textView = value as RoutePlannerTextView; }
   }

   public void Initialize()
   {
      _textView = new RoutePlannerTextView();

      // Initialize other fields here...

      // Initialize services container
      IoC.CreateContainer(this);
      IoC.Container.RegisterInstance(Framework, new ExternallyControlledLifetimeManager());

      // Must store the preferences as an instance since the Framework's preferences can
      // change at any point after this registration.
      IoC.Container.RegisterInstance(typeof(IPreferences), Framework.Preferences,
                                     new ExternallyControlledLifetimeManager());
   }

   public void AddPointById(string id, int index)
   {
      // Add to text view...
      TextView.AddPointById(id, index);

      // Add to map view...
      // ...
   }
}
```

ICountable Interface

No change. Refer to the latest `ICountable`.

RoutePlannerTextView Class

No change. Refer to the latest `RoutePlannerTextView`.

NullRoutePoint Class

No change. Refer to the latest `NullRoutePoint`.

IRoutePoint Interface

No change. Refer to the latest `IRoutePoint`.

Simulation

In this simulation, two route plan objects are created. One is created with the default color preferences (Red, Green, Blue) that are set at application start-up. The other route plan is created after color preferences have been changed (Yellow, Black, White). As the simulation runs and points are added to the grid, their colors now reflect the color preferences that were established at instance construction versus the dynamic preferences of the system.

```
public class MultitonPatternSim
{
   public static void Run()
   {
      Console.WriteLine("***Multiton Pattern***");

      // Simulate creating two independent route plans within the system.
      var routePlanDefaultPrefs = new RoutePlanner();
      MockGpsRoutePlannerSystem.RegisterComponent(routePlanDefaultPrefs);

      // Change the preferences in the system.  All objects created after this point will
      // receive these preferences.
      MockGpsRoutePlannerSystem.ChangePreferences(Color.Yellow, Color.Black, Color.White);

      var routePlanChangedPrefs = new RoutePlanner();
      MockGpsRoutePlannerSystem.RegisterComponent(routePlanChangedPrefs);

      Console.WriteLine("Creating route plan 1 with default preferences...");
      IoC.UpdateContainer(routePlanDefaultPrefs);
      RunRoutePlanSim(routePlanDefaultPrefs);
```

```
      Console.WriteLine("Creating route plan 2 with changed preferences...");
      IoC.UpdateContainer(routePlanChangedPrefs);
      RunRoutePlanSim(routePlanChangedPrefs);
   }

   private static void RunRoutePlanSim(RoutePlanner routePlanObj)
   {
      var textView = routePlanObj.TextView;

      var i = 0;
      Console.WriteLine("{0}Step {1}: Start with an empty grid.", Environment.NewLine,
                        ++i);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in first row.",
                        Environment.NewLine, ++i);
      routePlanObj.AddPointById("MCN/A", 0);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in middle row.",
                        Environment.NewLine, ++i);
      routePlanObj.AddPointById("dob", textView.Size / 2);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in last row.", Environment.NewLine,
                        ++i);
      routePlanObj.AddPointById("GQO", textView.Size - 1);
      PrintGrid(textView);
   }

   private static void PrintGrid(IGpsTextView<IRoutePoint> textView)
   {
      var header = new StringBuilder().Append("ID\t\t")
                                      .Append("Description\t\t\t")
                                      .Append("Lat\t\t")
                                      .Append("Lon\t\t\t")
                                      .Append("Color").AppendLine();
      header.Append("=====\t")
            .Append("===========\t\t\t")
            .Append("=====\t")
            .Append("=====\t\t")
            .Append("=====");
```

```
        Console.WriteLine(header.ToString());
        foreach (var routePoint in textView.Grid)
        {
            Console.WriteLine(routePoint);
        }

        // Display the number of points in the current grid.
        Console.WriteLine("=== Number of points in the grid: {0} ===",
                          (textView as ICountable).Count);
    }
}
```

Output

routePlanDefaultPrefs

```
Creating route plan 1 with default preferences...

Step 1: Start with an empty grid.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 0 ===

Step 2: User enters data in first row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 1 ===

Step 3: User enters data in middle row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                      <Transparent>
```

routePlanChangedPrefs

```
Creating route plan 2 with changed preferences...

Step 1: Start with an empty grid.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 0 ===

Step 2: User enters data in first row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Yellow>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 1 ===

Step 3: User enters data in middle row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Yellow>
<Blank>                                      <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Black>
<Blank>                                      <Transparent>
```

routePlanDefaultPrefs

```
<Blank>                                    <Transparent>
=== Number of points in the grid: 2 ===

Step 4: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                    <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Green>
<Blank>                                    <Transparent>
GQO/W  Choo Choo            34.57  -85.09    <Blue>
=== Number of points in the grid: 3 ===
```

routePlanChangedPrefs

```
<Blank>                                    <Transparent>
=== Number of points in the grid: 2 ===

Step 4: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Yellow>
<Blank>                                    <Transparent>
DOB/N  Dobbins              33.54  -84.3     <Black>
<Blank>                                    <Transparent>
GQO/W  Choo Choo            34.57  -85.09    <White>
=== Number of points in the grid: 3 ===
```

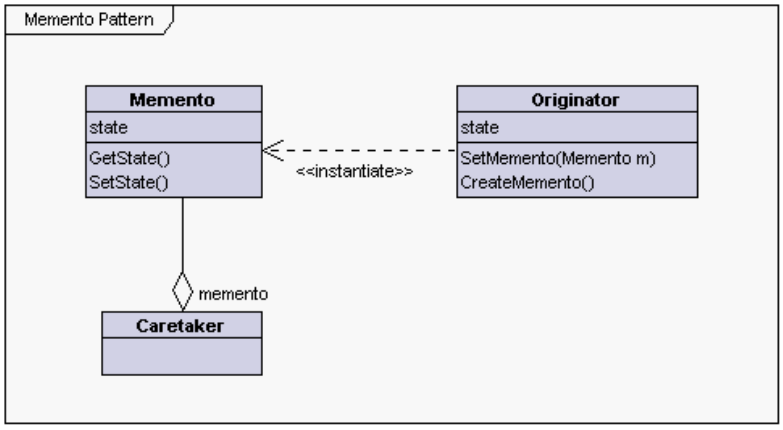
Advantages

- Allows multiple Singletons to exist based on different contexts (implemented in the form of keys).

Disadvantages

- Because the root of the Multiton is still based on the concept of Singleton objects, unit testing such designs can be very difficult.
- Becomes easier to introduce memory leaks to the system due to the inherent nature static references.

Sprint 4: Memento Design Pattern



4.1 Applied Example – Undo Support

The final sprint will address the specification for providing support for undo operations. To accomplish this, the ***Memento Pattern*** was utilized. This was realized by implementing the `IMementoOriginator` interface and using the `UndoStack` that is provided by the RPS.

4.1.1 Implemented Solution

RoutePlanner Class (Caretaker)

This class is the logical representation of the Caretake object in the diagram above. For this implementation, the `RoutePlanner` class is responsible for deciding when to push/pop the objects to/from the stack. In a more realistic scenario, the framework would respond to a user event (context menu, button click, etc.) and notify this class to initiate the undo transaction.

```
public class RoutePlanner : IGpsComponent, IGpsDataObject<IRoutePoint>, IRoutePlanner
{
   private RoutePlannerTextView _textView;

   public IGpsFramework Framework { get; set; }
   public IGpsMapView MapView { get; set; }
   public IGpsTextView<IRoutePoint> TextView
   {
      get { return _textView; }
      set { _textView = value as RoutePlannerTextView; }
   }

   public void Initialize()
   {
      _textView = new RoutePlannerTextView();

      // Initialize other fields here...

      // Initialize services container
      IoC.CreateContainer(this);
      IoC.Container.RegisterInstance(Framework, new ExternallyControlledLifetimeManager());
      IoC.Container.RegisterInstance(typeof(IPreferences), Framework.Preferences,
                                     new ExternallyControlledLifetimeManager());
   }

   public void AddPointById(string id, int index)
   {
```

```
        // Push the current text view state to the stack BEFORE editing.
        Framework.UndoStack.Push(_textView);

        // Add to text view...
        TextView.AddPointById(id, index);

        // Add to map view...
        // ...
    }

    public void Undo()
    {
        // Restore the last undo point.
        Framework.UndoStack.Pop(_textView);
    }
}
```

RoutePlannerTextView Class (Originator)

This class is the logical representation of the Originator object in the diagram above. By implementing `IMementoOriginator`, this class is now supported by the framework-provided undo capabilities. When this object is pushed to the framework's `UndoStack`, it will call the `CreateMemento` method. When it is popped off the `UndoStack`, it will call the `RestoreMemento` method.

```
public class RoutePlannerTextView : IGpsTextView<IRoutePoint>, ICountable, IMementoOriginator
{
    private List<IRoutePoint> _grid;

    public IList<IRoutePoint> Grid
    {
        get { return _grid.AsReadOnly(); }
    }

    public RoutePlannerTextView()
    {
        var size = 5;
        _grid = new List<IRoutePoint>();
        for (var i = 0; i < size; i++)
        {
            _grid.Add(new NullRoutePoint());
        }
    }

    public IRoutePoint AddPointById(string id, int index)
    {
        if ((index < 0) || (index >= _grid.Count))
        {
            var msg = string.Format("The text view cannot add items at index {0}", index);
            throw new IndexOutOfRangeException(msg);
        }
        var addedRoutePt = (string.IsNullOrEmpty(id))
                               ? new NullRoutePoint()
                               : new RoutePoint(id) as IRoutePoint;
        _grid[index] = addedRoutePt;
        return addedRoutePt;
    }

    /// <summary>
    /// Returns the size of the grid.
    /// </summary>
```

```
   public int Size { get { return _grid.Count; } }

   /// <summary>
   /// Returns the count of non-blank points in the grid.
   /// </summary>
   public int Count
   {
      get { return _grid.Count(pt => !pt.IsBlank); }
   }

   public dynamic CreateMemento()
   {
      return new TextViewGridMemento(_grid);
   }

   public void RestoreMemento(dynamic memento)
   {
      _grid = new List<IRoutePoint>(memento.Grid);
   }
}
```

TextViewGridMemento Class (Memento)

This class is the logical representation of the Memento object in the diagram above. It is responsible for storing the state of the `RoutePlannerTextView` and is the object pushed and popped to/from the framework's `UndoStack`. For this implementation, it is only concerned with storing the state of the grid.

```
public class TextViewGridMemento
{
   private readonly IList<IRoutePoint> _grid;

   public TextViewGridMemento(IEnumerable<IRoutePoint> grid)
   {
      _grid = new List<IRoutePoint>(grid);
   }

   public IList<IRoutePoint> Grid
   {
      get { return _grid; }
   }
}
```

IRoutePlanner Interface

No change. Refer to the latest `IRoutePlanner`.

ICountable Interface

No change. Refer to the latest `ICountable`.

RoutePoint Class

No change. Refer to the latest `RoutePoint`.

NullRoutePoint Class

No change. Refer to the latest `NullRoutePoint`.

IRoutePoint Interface

No change. Refer to the latest `IRoutePoint`.

IoC Class

No change. Please refer to latest `IoC`.

Simulation

For this simulation, a `RoutePlanner` object is created and several points are added. Based on the current implementation, as points are added, a snapshot of the state of the grid is stored on the `UndoStack`. The next sequence undoes the previous steps of adding the points by initiating repeated undo calls on the `RoutePlanner` object. As each undo is performed, the grid is brought to its next previous state.

```
public class MementoPatternSim
{
   public static void Run()
   {
      Console.WriteLine("***Memento Pattern***");

      var mockRoutePlannerObj = new RoutePlanner();
      MockGpsRoutePlannerSystem.RegisterComponent(mockRoutePlannerObj);

      Console.WriteLine("Creating route plan with default preferences...");
      int i;
      RunRoutePlanSim(mockRoutePlannerObj, out i);
      UndoAll(mockRoutePlannerObj, ref i);
   }

   private static void UndoAll(RoutePlanner routePlanObj, ref int i)
   {
      var textView = routePlanObj.TextView;

      Console.WriteLine("{0}Step {1}: User undoes last action: (add to middle row).",
                        Environment.NewLine, ++i);
      routePlanObj.Undo();
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User undoes last action: (add to last row).",
                        Environment.NewLine, ++i);
      routePlanObj.Undo();
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User undoes last action: (add to first row).",
                        Environment.NewLine, ++i);
      routePlanObj.Undo();
      PrintGrid(textView);
   }

   private static void RunRoutePlanSim(RoutePlanner routePlanObj, out int i)
   {
      var textView = routePlanObj.TextView;

      i = 0;
      Console.WriteLine("{0}Step {1}: Start with an empty grid.", Environment.NewLine,
                        ++i);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in first row.",
                        Environment.NewLine, ++i);
      routePlanObj.AddPointById("MCN/A", 0);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in last row.", Environment.NewLine,
```

```
                        ++i);
      routePlanObj.AddPointById("GQO", textView.Size - 1);
      PrintGrid(textView);

      Console.WriteLine("{0}Step {1}: User enters data in middle row.",
                        Environment.NewLine, ++i);
      routePlanObj.AddPointById("dob", textView.Size/2);
      PrintGrid(textView);
   }

   private static void PrintGrid(IGpsTextView<IRoutePoint> textView)
   {
      var header = new StringBuilder().Append("ID\t\t")
                                      .Append("Description\t\t\t")
                                      .Append("Lat\t\t")
                                      .Append("Lon\t\t\t")
                                      .Append("Color").AppendLine();
      header.Append("=====\t")
            .Append("===========\t\t\t")
            .Append("=====\t")
            .Append("=====\t\t")
            .Append("=====");
      Console.WriteLine(header.ToString());
      foreach (var routePoint in textView.Grid)
      {
         Console.WriteLine(routePoint);
      }

      // Display the number of points in the current grid.
      Console.WriteLine("=== Number of points in the grid: {0} ===",
                        (textView as ICountable).Count);
   }
}
```

Output

```
***Memento Pattern***

Creating route plan with default preferences...

Step 1: Start with an empty grid.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 0 ===

Step 2: User enters data in first row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
MCN/A  Macon Regional       32.69  -83.64    <Red>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
<Blank>                                      <Transparent>
=== Number of points in the grid: 1 ===

Step 3: User enters data in last row.
ID     Description          Lat    Lon       Color
=====  ===========          =====  =====     =====
```

```
MCN/A  Macon Regional      32.69  -83.64    <Red>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
GQO/W  Choo Choo           34.57  -85.09    <Blue>
=== Number of points in the grid: 2 ===

Step 4: User enters data in middle row.
ID     Description         Lat    Lon       Color
=====  ===========         =====  =====     =====
MCN/A  Macon Regional      32.69  -83.64    <Red>
<Blank>                                     <Transparent>
DOB/N  Dobbins             33.54  -84.3     <Green>
<Blank>                                     <Transparent>
GQO/W  Choo Choo           34.57  -85.09    <Blue>
=== Number of points in the grid: 3 ===

Step 5: User undoes last action: (add to middle row).
ID     Description         Lat    Lon       Color
=====  ===========         =====  =====     =====
MCN/A  Macon Regional      32.69  -83.64    <Red>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
GQO/W  Choo Choo           34.57  -85.09    <Blue>
=== Number of points in the grid: 2 ===

Step 6: User undoes last action: (add to last row).
ID     Description         Lat    Lon       Color
=====  ===========         =====  =====     =====
MCN/A  Macon Regional      32.69  -83.64    <Red>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
=== Number of points in the grid: 1 ===

Step 7: User undoes last action: (add to first row).
ID     Description         Lat    Lon       Color
=====  ===========         =====  =====     =====
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
<Blank>                                     <Transparent>
=== Number of points in the grid: 0 ===
```

Advantages

- Decouples the process of saving the state from the Originator and places the responsibility on the Memento.
- Accomplishes saving of the Originator's state without breaking encapsulation.

Disadvantages

- Saving and restoring state can be taxing to a system's resources.

Appendix A – Mock GPS Route Planning System (RPS)

This section lists the code used to represent the mock GPS route planning system (RPS) utilized throughout this project. Each item will be presented with a brief description of its representation and purpose in the system.

IGpsComponent

This is a required interface for any component that wishes to plug-in to the RPS.

```
public interface IGpsComponent
{
   IGpsFramework Framework { get; set; }
   void Initialize();
}
```

IGpsDataObject

This is a required interface for any component that wishes to have its data displayed by the RPS. The RPS has two views that can display data; one for a graphical representation and another for a text-based representation. For the purposes of this project, the focus will be on implementing a text-based representation for a particular RPS data object.

```
public interface IGpsDataObject<T>
{
   IGpsMapView MapView { get; set; }
   IGpsTextView<T> TextView { get; set; }
}
```

IGpsFramework

This interface is implemented by the RPS and is provided as a service object to any plug-in components that wish to retrieve session data that is made available by the system at run-time (such as preferences, databases, etc.).

```
public interface IGpsFramework
{
   RoutePointDB RoutePointDB { get; }
   Preferences Preferences { get; }
   UndoStack UndoStack { get; }
}
```

IGpsMapView

This interface is provided for illustration purposes. A realistic implementation of this interface could possibly provide drawing and mapping services for client objects to use during map view planning sessions.

```
public interface IGpsMapView
{
   // GpsMapView interface...
}
```

PointType

This enumerated type is provided by the RPS for internal and client use. The types listed are a subset of different point types regularly seen in navigational contexts.

```
public enum PointType
{
   User,
```

```
   Airport,
   Navaid,
   Waypoint,
}
```

IGpsTextView

This interface is required for plug-in components that wish to have their data shown in the text-based view of the RPS.

```
public interface IGpsTextView<T>
{
   IList<T> Grid { get; }
   int Size { get; }
   T AddPointById(string id, int index);
}
```

IPreferences

This interface is used by the RPS as a default representation for preferences. In a realistic implementation, different plug-in data objects would require different preferences and this interface would not be used other than scenarios where the data object needs to know the system's preferences. For this project, this interface will be used by the plug-in object to attain the color map for the different point types that is specified by the RPS.

```
public interface IPreferences
{
   Color this[PointType type] { get; }
}
```

Preferences

This is the concrete implementation of `IPreferences`. This is implemented by the RPS to provide a basic color map for the supported point types.

```
public class Preferences : IPreferences
{
   private readonly Dictionary<PointType, Color> _colors =
      new Dictionary<PointType, Color>();

   private Preferences()
   {
   }

   public Preferences(Color airportColor, Color navaidColor, Color waypointColor)
   {
      _colors.Add(PointType.Airport, airportColor);
      _colors.Add(PointType.Navaid, navaidColor);
      _colors.Add(PointType.Waypoint, waypointColor);
   }

   public Color this[PointType type]
   {
      get { return _colors.ContainsKey(type) ? _colors[type] : Color.Transparent; }
   }
}
```

IGpsPoint

This interface is used by the RPS to represent GPS-based points in the system.

```
public interface IGpsPoint
{
   string Name { get; }
   string Description { get; }
   double Lat { get; }
   double Lon { get; }
   PointType PointType { get; }
}
```

GpsPoint

This is the concrete implementation of the

IGpsPoint interface.

```
public struct GpsPoint : IGpsPoint
{
   public string Name { get; internal set; }
   public string Description { get; internal set; }
   public double Lat { get; internal set; }
   public double Lon { get; internal set; }
   public PointType PointType { get; internal set; }

   public GpsPoint(string name, string description, double lat, double lon, PointType type)
      : this()
   {
      Name = name;
      Description = description;
      Lat = lat;
      Lon = lon;
      PointType = type;
   }
}
```

RoutePointDB

This class is a mock representation of a database of navigation aid point data that is provided by the RPS for use by client objects.

```
/// <summary>
/// Mock representation of a navigational aid database.
/// </summary>
public class RoutePointDB
{
   private readonly List<IGpsPoint> _points = new List<IGpsPoint>();

   public RoutePointDB()
   {
      _points = new List<IGpsPoint>
         {
            // Airports
            new GpsPoint("MCN/A", "Macon Regional", 32.69, -83.64, PointType.Airport),
            new GpsPoint("WRB/A", "Robins AFB", 32.65, -83.60, PointType.Airport),
            new GpsPoint("ATL/A", "Atlanta Int'l", 33.64, -84.40, PointType.Airport),
            // Navaids
            new GpsPoint("DOB/N", "Dobbins", 33.54, -84.30, PointType.Navaid),
```

```
            new GpsPoint("MGM/N", "Montgomery", 32.13, -86.19, PointType.Navaid),
            new GpsPoint("VAD/N", "Moody", 30.57, -83.11, PointType.Navaid),
            // Waypoints
            new GpsPoint("VUZ/W", "Vulcan", 33.40, -86.53, PointType.Waypoint),
            new GpsPoint("VNA/W", "Vienna", 32.12, -83.29, PointType.Waypoint),
            new GpsPoint("GQO/W", "Choo Choo", 34.57, -85.09, PointType.Waypoint),
         };
   }

   public IGpsPoint FindPointById(string id)
   {
      if (id.Contains('/')) return SearchByFullQualifier(id.ToUpperInvariant());
      return SearchByPartialQualifier(id.ToUpperInvariant());
   }

   private IGpsPoint SearchByFullQualifier(string id)
   {
      var subsetId = id.Substring(id.LastIndexOf('/')).Last();

      PointType subsetType;
      switch (subsetId)
      {
         case 'A':
            subsetType = PointType.Airport; // Search airports.
            break;
         case 'N':
            subsetType = PointType.Navaid; // Search navaids.
            break;
         case 'W':
            subsetType = PointType.Waypoint; // Search waypoints.
            break;
         default:
            throw new ApplicationException("Unsupported point type.");
      }
      var pointsSubset = _points.Where(p => p.PointType == subsetType);

      return pointsSubset.FirstOrDefault(pt => pt.Name.Equals(id)) ??
               GetDefaultPoint(id);
   }

   private IGpsPoint SearchByPartialQualifier(string id)
   {
      return
         _points.FirstOrDefault(pt =>
                                  pt.Name.Substring(0,
pt.Name.LastIndexOf('/')).Equals(id)) ??
         GetDefaultPoint(id);
   }

   private IGpsPoint GetDefaultPoint(string id)
   {
      return new GpsPoint
         {
            Name = string.Format(".{0}", id.ToUpperInvariant()),
            Description = string.Empty,
            Lat = 0.0,
            Lon = 0.0,
            PointType = PointType.User
```

```
        };
    }
}
```

IMementoOriginator

This interface is an optional interface meant for objects wishing to support undo within the RPS. It allows objects to act as the Originator object in the ***Memento Pattern***.

```
public interface IMementoOriginator
{
    dynamic CreateMemento();
    void RestoreMemento(dynamic memento);
}
```

UndoStack

This is logical representation of the undo stack utilized by the RPS. Objects that wish to support undo operations, must implement the `IMementoOriginator` interface so that it can be passed to the system's undo stack.

```
public class UndoStack
{
    private Stack<dynamic> _stack;

    internal UndoStack(int capacity)
    {
        _stack = new Stack<dynamic>(capacity);
    }

    public void Push(IMementoOriginator mementoOriginator)
    {
        _stack.Push(mementoOriginator.CreateMemento());
    }

    public void Pop(IMementoOriginator mementoOriginator)
    {
        mementoOriginator.RestoreMemento(_stack.Pop());
    }
}
```

MockGpsRoutePlannerSystem

This represents the mock implementation of the RPS.

```
/// <summary>
/// Mock representation of a GPS route planning system.
/// </summary>
public static class MockGpsRoutePlannerSystem
{
    static MockGpsRoutePlannerSystem()
    {
        Framework = new MockGpsFramework();
        Components = new List<IGpsComponent>();
    }

    private static List<IGpsComponent> Components { get; set; }
    private static IGpsFramework Framework { get; set; }

    public static void RegisterComponent(IGpsComponent component)
```

```
   {
      Components.Add(component);
      component.Framework = Framework;
      component.Initialize();
   }

   public static void ChangePreferences(Color airportColor, Color navaidColor,
                                        Color waypointColor)
   {
      (Framework as MockGpsFramework).Preferences = new Preferences(airportColor,
                                                                    navaidColor,
                                                                    waypointColor);
   }

   /// <summary>
   /// Mock representation of the framework supplied by the GPS route planning system.
   /// </summary>
   private class MockGpsFramework : IGpsFramework
   {
      public RoutePointDB RoutePointDB { get; private set; }
      public Preferences Preferences { get; internal set; }
      public UndoStack UndoStack { get; private set; }

      public MockGpsFramework()
      {
         RoutePointDB = new RoutePointDB();
         Preferences = new Preferences(Color.Red, Color.Green, Color.Blue);
         UndoStack = new UndoStack(5);
      }
   }
}
```

Appendix B – Non-Direct Activity Report

Date	Duration (minutes)	Specific Task / Activity
17-Mar-2014	475	Write report for Project #2, setup & research for Project #3
30-Mar-2014	88	Research for Project #3
20-Apr-2014	254	Research for Project #3
21-Apr-2014	170	Research & write code/report for Project #3
22-Apr-2014	385	Research & write code/report for Project #3
24-Apr-2014	182	Research & write code/report for Project #3
25-Apr-2014	500	Research & write code/report for Project #3
Sum for Report #3	**2054**	**/ 1500 (5 weeks @ 300/wk)**
Cumulative Sum for the Course	**7779**	**/ 4500**