CSE Programming in C# **Roger Crawfis**

Programming Assignment #3

A SceneGraph Library - Part II: Internal classes, Factories and Serialization.

We will now create two applications, one for creating a scenegraph and the other for reading it in and printing it out to the console. This will allow you to work with reading input from the console, using internal classes that support a public interface, and using .NET's support for serialization to write and read arbitrary graph data structures. Additionally, we will use a couple of Windows Form controls. We will start with your results from Programming Assignment 2. We will perform the following tasks:

1. Create a new Console Application (in the same solution) called CreateScene.
2. Create a new Console Application (again in the same solution) called ReadScene.
3. Change SceneGraph.exe to a class library (i.e., a dll).
4. Move your concrete visitors to the ReadScene project.
5. Add attributes for serialization for all of your classes in the SceneGraph project.
6. Explicitly indicate the protection all of your classes in the SceneGraph project to be **internal**.
7. Add a public Factory interface (ISceneGraphFactory) to ISceneGraph.
8. Add a public Factory class to SceneGraph that implements ISceneGraphFactory. This should not be marked as serializable.
9. Add a class called SearchVisitor to CreateScene that implements IVisitor.
10. Have IGroupNode require implementation of IEnumerable<ISceneGraphNode> and update your concrete implementation to support this. This will allow you to loop through the children. You can simply return the internal container's enumerator. Create a test case for this.

Additional Details for ISceneGraphFactory and SceneGraphFactory

The ISceneGraphFactory should have the following methods.

```
IDrawableNode CreateDrawableNode(string name, string DrawableType, object drawableData);
IGroupNode CreateGroupNode(string name, string groupType, object groupData);
IStateNode CreateStateNode(string name, string stateType, object stateData);
ITransformNode CreateTransformNode(string name, string transformType, object transformData);
```

For your concrete implementation, the last argument will be ignored and you can pass in null for its value when you make the calls to it. It would be used for setting values in the specific type created (e.e, cube size, building type, etc.). The second argument will be a string indicating what type of object to create (e.g., a cube, a terrain, etc.). You are to use a switch statement inside your code to process this argument. If an invalid string is passed in as the argument, then you are to throw an `ArgumentOutOfRangeException` exception. Here is an example for the IDrawableNode:

```
default:
    throw new ArgumentOutOfRangeException("The type " + DrawableType + " is not a recognized
    IDrawableNode.");
```

Additional Details for the SearchVisitor

- SearchVisitor will have a public static method called Find, which takes the search string and an ISceneNode, and returns an ISceneNode (if found) or *null*. This is an example of a single-serving visitor pattern (http://en.wikipedia.org/wiki/Single-serving_visitor_pattern). It will hide any complexity in using the visitor and allow for a single call to search (as opposed to several calls and checking if successful). The implementation will do all of this and hide it to the user of the class. Here is the signature for the Find method.

```
static public ISceneNode Find(string targetName, ISceneNode root)
```

- Nice example of readonly and static methods. Use a readonly variable for the internal target string. This implies that you will need to pass it into the constructor. The constructor should be private or internal (not public). Likewise a private or internal property should be created to hold the result. Find can query this and return it. Here is an internal method that I have all PreVisit methods call. I will let you figure out the rest.

```
private void CompareToTarget(ISceneNode node)
{
    if (notFound && targetName == node.Name)
    {
       notFound = false;
       this.Result = node;
    }
}
```

Additional Details for CreateScene

- CreateScene will enter a loop prompting for input from the console and exiting when the string “**Exit**” is entered. The only other keyword needed would be the string “**Add**”. If “Add” is entered, prompt for the type of ISceneNode to create, the string to be used for its name and the name of the group node to add it to. Start off by creating a group node called “Root” before you enter the loop. If the group node with the entered name does not exist (using the SearchVisitor) then an error should be written to the console and the current command should be cancelled.
- Likewise, if the type entered does not exist, print an error message. To accomplish this you could catch the exception, but for this assignment, create a List<string> and populate it with the known scene node types. You can then use the Contains method on the List to see if a legal string was entered.
- Once “Exit” is entered, you should break out of the loop. You will now write the scenegraph to a file. For this, Add the Reference System.Windows.Forms and create a `System.Windows.Forms.SaveFileDialog` object. You will use the ShowDialog method to display a window to the user to enter a file. Here is an example:

```
System.Windows.Forms.SaveFileDialog dialog = new System.Windows.Forms.SaveFileDialog();
if (dialog.ShowDialog() == System.Windows.Forms.DialogResult.OK)
{
  Stream stream = File.Open(dialog.FileName, FileMode.Create);
  BinaryFormatter formatter = new BinaryFormatter();
  using (stream)
  {
     formatter.Serialize(stream, root);
  }
}
```

- You will need to put the [STAThread] attribute before Main.
- Set the StartUp project to this project and Run it. Alternatively, you can Right-Click the project and select the Debug menu to start a project.
- If you want you can copy the PrintVisitor over to this project as well (it will be a separate file / class) and print the scene out to the console before you exit.

Additional Details for ReadScene

- ReadScene will be simple. It will prompt for a file to read in, use the PrintVisitor to print the scene to the Console and exit.

- To prompt for a file, you will need the `System.Windows.Forms.OpenFileDialog` class. Use it similar to the SaveFileDialog example above.
- You will need to typecast the return value from Deserialize to an ISceneNode.
- You will need to put the [STAThread] attribute before Main.
- Again to Run this from Visual Studio you will need to change the startup project or use the menu on the project.
- You can make the file type any extension you want. I used .osu.

Additional Tasks

1. In the Properties folder, Open the AssemblyInfo.cs file and fill in the Title through the Copyright information. This information will be displayed if you look at the properties of your .exe file.
2. Provide comments explaining your logic. Also, add a block comment to the beginning of the file listing your name, the course and a description of the lab (from above but in a completed tense).
3. Do a ***Build Clean,*** zip up your solution (no .exe files are allowed) and then submit your assignment using submit c459ae directory.