CSE 459.24 – Programming in C# **Roger Crawfis**

In-Class Worksheet #2

Overview

For this worksheet, we will create a class library and then use the resulting dll in a console application. This worksheet is a start on your next programming assignment, where we will build a directed acyclic graph (a DAG) and traverse it using an interface (which we will call IVisitor). Each node in the graph will be of type ISceneNode. The goal here is to allow multiple implementations of the IVisitor that can be determined at run-time. It uses method overloading to resolve the scenenode's type at runtime. We will have several different flavors of ISceneNode's and will define interfaces for each of these flavors.

Before You Start

In Programming Assignment #2 you will be implementing a scene graph. I would advise you to take a look at the assignment before you begin this worksheet. Things to understand before you begin:

- What a scene graph is.
- The visitor design pattern. In addition to the resources listed on the assignment you may find the details section of this wiki page helpful. http://en.wikipedia.org/wiki/Visitor_%28design_pattern%29
- A basic understanding of the class hierarchy in the assignment.

Creating a Class Library

You should have Visual Studio 2008 open.

1. Create a new Project either from the shortcut on the Start page or through the file menu,
2. Select the type to be a C# Class Library. This will create a .dll file rather than an .exe file. Dll's are loaded at runtime. Where the runtime finds these dll's is a rather complex set of paths. This was touched upon in the early material for the course, but is not important for this level. Here the .dll will be in the same directory as the .exe file. For more information on dll's go to http://support.microsoft.com/kb/815065. Note, that for the most part dll's are being used in place of statically linked in libraries (.lib files). In C++, we probably would have implemented a library rather than a dll.
3. In the Location textbox, enter your desired hard drive location. Note that this will be on the mounted z: drive in the CSE environment.
4. Name the library ISceneGraph.
5. Select OK.
6. You will notice that Visual Studio has created a file called Class1.cs and opened it in the Code View. Note the namespace used and the class name. Right-click on this file in the Solution Explorer and *Delete* it.
7. Select the menu item *Project->ISceneGraph Properties*. Here you will see that it is building a class library, the name of the library (dll), the environment support it needs, etc.
8. Change the Default Namespace to your lastname.SceneGraph (e.g., Crawfis.SceneGraph). Now any new files added (to this project) will be encapsulated in this namespace. You can always change this in the .cs file to another namespace or class name.
9. Right-click on the Project, ISceneGraph, and select *Add->New Item*. Select *Code* from the left panel. This filters the choices down to make it easier to find the one we are after. Select *Interface*. Name this interface ISceneNode.cs. Microsoft naming convention has all interfaces starting with a Capital I followed by

```
namespace Crawfis.SceneGraph
{
   public interface ISceneNode
   {
         void Accept(IVisitor visitor);
         string Name { get; }
   }
}
```

the name of the class/interface using Pascal case. (Ex: PascalCase)

10. Insert the following code such that your interface looks like the the code on the right. The interface requires all implementations to have a property called Name that is a string, and a method Accept that takes an IVisitor type and has no return value. Recall, that all methods and properties are public in an interface (it is its public interface). So, the property Name has a public getter. There is no requirement for a public setter. The implementor can choose what to provide. The actual interface can be public or internal (only exposed to the current project).
11. Right-click anywhere in the code area and Select *Organizing Usings->Remove and Sort*. This will delete any unneeded using statements (in this case, all of them).
12. Add another interface called IVisitor, with the following methods:

```
void PreVisit(IDrawableNode drawable);
void PreVisit(ITransformNode transform);
void PreVisit(IStateNode state);
void PreVisit(IGroupNode group);
void PostVisit(IDrawableNode drawable);
void PostVisit(ITransformNode transform);
void PostVisit(IStateNode state);
void PostVisit(IGroupNode group);
```

13. We will not use an ISceneNode in the visitor, but the four types shown (IDrawableNode, ITransformNode, IStateNode, and IGroupNode). Each of these will require the ISceneNode interface to also be implemented.
14. Create an IDrawableNode interface as shown. Note that it is a public interface, it requires that all implementers also implement the ISceneNode interface and it adds a Draw method.

```
namespace Crawfis.SceneGraph
{
    public interface IDrawableNode : ISceneNode
    {
        void Draw();
    }
}
```

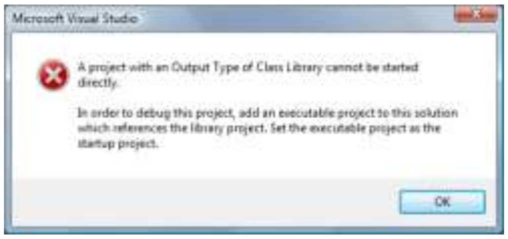
15. Add two more interfaces called IStateNode and ITransformNode that also require ISceneNode to be implemented and have the methods:

```
a.          void Apply();
b.          void UnApply();
```

16. Finally, add the interface IGroupNode. Again it should 1) be public, 2) require ISceneNode to be implemented and 3) require the following additional method:

```
a.          void AddChild(ISceneNode child);
```

17. OK. That is it. We are done. You can do a Build to ensure that it compiles fine.
18. If you try to do a *Debug->Start without Debugging* you will get an error message like the following. Since we selected a project of type Class Library (a .dll file) rather than an application (an .exe file), we have no executable (.exe file) to run. You cannot run a dll by itself. It needs to be loaded by another executable file.



A project with an Output Type of Class Library cannot be started directly.

In order to debug this project, add an executable project to this solution which references the library project. Set the executable project as the startup project.

19. OK, let's recap. We created a class library with the interfaces: IDrawableNode, IGroupNode, ISceneNode, IStateNode, ITransformNode, and IVisitor. There is absolutely no executable code! What good is this? Well, for getting things to work, nothing! For large-scale maintenance and protecting intellectual property, we can ship this contract to our customers (and even our competitors) and let them know that if they also program to this specification we can all work together. Your Software Engineering courses will hopefully reinforce this.
20. Edit the Assembly Information to provide your name, organization, description, etc.

Creating another library that implements our design

21. Right-click on the solution name in the solution explorer and select *Add->New Project* (alternatively, use the menu *File->Add->New Project*). Select a Console Application project. Name the Project *SceneGraph* and leave the directory location alone.
22. Open the new Project's Property page and change the Default Namespace to *YourLastname.SceneGraphCore*.
23. In Program.cs, change the namespace to *YourLastname.SceneGraphCore*.
24. Inside the Main method, create a variable of type ISceneNode as follows:
    a. ISceneNode root;
25. If you do a build now, you will see there is a compiler error indicating that it does not recognize the type ISceneNode. Why is that? Well, there are two reasons. One, the namespaces are different, but more importantly, those types are contained in a different project. This project knows nothing of the other project (ISceneGraph) and vice versa. To fix this, we need to add a reference to either the dll or the project.
26. Select the menu *Project->Add Reference*. This will take awhile, so be patient. There are several tabs on the dialogue for this. The .NET tab provides every registered dll on your system. Note the plethora of locations. We want our own (unregistered – aka non-signed) library. We could use Browse, to find it and add it, but Visual Studio provides a better solution for this common development situation. The Project tab will list the other project (ISceneGraph). Select it and click OK. You may notice that ISceneGraph has now appeared in the references folder for this project.
27. Rebuild the solution. You will still have the compiler error. However, if you click on the type reference (ISceneNode) in Main, you will see a little red underscore box (if not, see the note at the end of this item). This is a smart tab. Hover over it for a second and a list of pop-up options to correct this error will appear. In this case we can either add a using statement to the file, or fully specify the type (e.g., Crawfis.SceneGraph.ISceneNode). Choose to add the using statement. (You can also right click on the error and select resolve.) Now your program should compile. Again, we do not have any implementation and cannot set the variable root to anything, since no concrete implementations exist. Let's fix this. FYI, If you did not get the fix error tab, or your program does not compile, this is probably because you did not make your interface public. Go back and compare it to the code in step 10 above. Remember that if an interface is not made public other ***projects*** within your solution will not be able to inherit that interface.
28. There are many possible designs for scene graphs and how you might implement these. I will help you implement two ISceneNode classes for your lab, the cube class and the group class. Additionally, I will help you implement a simple concrete visitor to print out the DAG. We will add all of our concrete classes to the console application project.
29. Right-click on the SceneGraph project and select *Add->Class*. Name the class *GroupNode*. Some notes: 1) you could also do this from the Project menu, but things are now muddled as there are two projects. Which project will it add it to? The answer is whichever one is active (as if that helps). The Dialogue box lists the project name in the title bar. 2) You could just as easily select *Add->New Item* and select Class from the list of items.
30. Specify that GroupNode should implement the IGroupNode interface. Intellisense will not list it, but after you type it you will get the fix error box again to add the using statement. Add the using statement.
31. ~~Click on the IGroupNode text and you will now see a blue underscore box. This is a smart tab to help you with common tasks. If you hover over it, two options are listed. Select the first one to implement the IGroupNode interface. All of the methods for IGroupNode are stubbed out, including those for ISceneNode and IEnumerable.~~
32. The code also contains some #region pragmas. I personally like to delete the empty white space after the #region's and before the #endregion's. Experiment with the ability to hide or collapse this code by clicking the little minus '-' signs on the left of the window. You can find information on #region at http://msdn.microsoft.com/en-us/library/9a1ybwek.aspx.

33. We will use a new feature with C# 3.0, called automatic properties, for the Name property. Replace the Name property with the following code:

```
public string Name
{
    get;
    private set;
}
```

34. We do not need a set from the interface specification, but we need one with automatic properties. We could make this public, private or protected. For this class we will make it private (for no real good reason). In general, avoid making things public unless there is a compelling reason.

35. Add the following code at the end of the class to keep track of the children for the group:

```
#region Member variables
private IList<ISceneNode> children = new List<ISceneNode>(8);
#endregion
```

36. Note that I personally prefer to use interfaces even for internal variables, so I used the IList rather than List for children. If you want to use List, then you can replace `IList` with `List` or better yet, replace `IList<ISceneNode>` with `var`. You may want to skip ahead to the Generics lecture to understand the <T> notation. I could have had you use ArrayList and the non-generic IList, but these should ***never be used again*** ☺ and teaching them is a waste of time for me and you!
37. The hardwired 8 above is very bad. Shame on me. I will let you fix that by creating a static constant that indicates what the 8 is for. It specifies the initial capacity of the List. Eight sounded like a good number ☺. Note that children is set to an empty list with an initial capacity for eight entries. With C# we do not need to do this initialization in the constructor(s). We can specify it more cleanly in the declaration. It will be created and initialized by the runtime before the constructor is called.
38. We can now implement the IGroupNode members easily:

```
public void AddChild(ISceneNode child)
{
    children.Add(child);
}
```

39. Any errors associated with a bad argument (e.g., child==null), will throw an exception from the List class. This seems reasonable to me. Sometimes you want to catch the exception and re-throw it with a more meaningful error for your implementation. We do not cover exceptions in this course, so … moving on.
40. We still need to implement the Accept method. For this we want to call the IVisitor's PreVisit method, have all of our children accept the visitor and the call the IVisitor's PostVisit method. Here is the code:

```
public void Accept(IVisitor visitor)
{
    visitor.PreVisit(this);
    foreach (ISceneNode child in children)
        child.Accept(visitor);
    visitor.PostVisit(this);
}
```

41. The appropriate method is called (i.e., PreVisit(IGroupNode) since the type of **this** is resolved at compile time.
42. ~~For the GetEnumerator, simply return the List's enumerator, `children.GetEnumerator();`.~~
43. We should have a fully functional GroupNode now. However we need some initialization. Let's add a constructor that takes in a string:

```
public GroupNode(string name)
{
        this.Name = name;
    }
```
44. IDrawableNode is the interface for many classes, such as a Cube, Sphere, Terrain and Building. To support our concrete implementations, and to reinforce the learning of abstract classes, we will create a base class that ~~derives~~, no ~~implements~~, no partially implements and then redefines … well let's just look at the code.
45. Create a new class and call it DrawableNodeBase.
46. Have it implement the IDrawableNode interface.
47. So for our/my design, the visitor will call the Draw if that is what it wants to do. The IDrawableNode objects will all be leaf nodes of our scenegraph, so they will not have children. Hence the Accept method will simply call the PreVisit method followed by the PostVisit method:

```
    public void Accept(IVisitor visitor)
    {
        visitor.PreVisit(this);
        visitor.PostVisit(this);
    }
```
48. The Name property will be very similar to our GroupNode except we will make the setter protected. This will allow our derived classes to also change or set the Name. Why, no good reason!

```
    public string Name
    {
        get;
        protected set;
    }
```
49. Add a constructor for DrawableNodeBase that takes a string and sets the Name property.
50. The only thing left to do is to implement the Draw method. But we have no idea how to draw an abstract DrawableNodeBase. Not even sure what this thing is! Not like it is a cube or Boeing 747. This is where abstract classes and methods come into play. With an interface, everything is abstract. There is no implementation. For an abstract class we provide implementation that is either a) common to all derived classes or b) a default implementation that other classes can specialize if needed. If there are methods that we cannot determine any intelligent way to do either of these, then we can leave the implementation undefined and force the derived classes to provide one. If we have at least one of these methods, then we have an abstract class and must mark it as such.
51. Implement the Draw method as such:

```
    public abstract void Draw();
```
52. There is no implementation and it is marked as abstract. We must now also mark the class as abstract (kind of a safety check to make sure you know what you are doing). Change the class definition to:

```
 abstract class DrawableNodeBase : IDrawableNode
```
53. Now, we can implement Cube very easily. Create a new class called Cube and have it derive from DrawableBaseNode:

```
 class Cube : DrawableNodeBase
```
54. We do not need to indicate that it also implements the IDrawableNode interface, or ISceneNode interface as DrawabelNodeBase indicates that. If you were to do that, you would have to implement those interfaces explicitly.
55. We are not going to actually implement the Draw method in this class. Simply have it write a message to the Console that it was called:

```
    public override void Draw()
    {
        System.Console.WriteLine("Drawing a cube...");  ;
        }
```
56. Note you could have used the smart tag again to get the method stub (and region) for this. The override keyword indicates that it is replacing the implementation of the base class.
57. Finally, we need a Constructor. If you try to build it right now you will get a compiler error. Go ahead and try it out. Since we defined a constructor for DrawableNodeBase, C# no longer provides a built in default constructor. That is a constructor that takes no arguments. So we need to create a constructor for our cube class that references the constructor we made in DrawableNodeBase. Here is what we will do for this:

```
public Cube(string name)
    : base(name)
{
}
```

58. This just passes the string to the base class. Recall that a class can have only one base class, so the keyword base is used to indicate that. In this case, it is the DrawableNodeBase class.
59. So there you have seen a direct implementation of an interface with the GroupNode and a class hierarchy example with the DrawableBaseNode and Cube classes. The Terrain, Sphere and Building would/could also derive from the DrawableBaseNode.
60. Okay, let's implement a simple visitor and then I will leave the rest of the lab to you. Create a new class and call it PrintVisitor. Specify that it implements the IVisitor interface and use the smart tag to generate the method stubs. All of these methods will be implemented similarly.
61. Add some member variables to keep track of the indentation level:

```
#region Member Variables
string currentIndentation = "";
string indentIncrement = "    ";
#endregion
```

62. We will add an indentation level (indentIncrement) with each PreVisit call and return to the previous indentation level with each PostVisit call. Here is the implementation for the IDrawableNode.

```
public void PreVisit(IDrawableNode drawable)
{
    System.Console.WriteLine(currentIndentation
        + "A Drawable of type "
        + drawable.GetType().ToString()
        + " with name: " + drawable.Name);
    currentIndentation += indentIncrement;
}

public void PostVisit(IDrawableNode drawable)
{
    currentIndentation = currentIndentation.Substring(0,
       currentIndentation.Length - indentIncrement.Length);
}
```

63. I added some information on the type of the node here, so this is your PrintNameType visitor.
64. Provide similar implementations for the other methods, but change the Drawable in the first string to something more meaningful.
65. Finally, we need to test this. Here is a simple test to get you started. For your lab you will need a more substantial test:

```
static void Main(string[] args)
{
    ISceneNode root;
    root = new GroupNode("Root");
    IVisitor printGraph = new PrintVisitor();
    root.Accept(printGraph);
    ISceneNode node = new Cube("cube");
    (root as GroupNode).AddChild(node);
    node = new Cube("cube2");
    (root as GroupNode).AddChild(node);
    GroupNode snowman = new GroupNode("snowman");
    (root as GroupNode).AddChild(snowman);
    node = new Cube("Bottom");
    snowman.AddChild(node);
    node = new Cube("Middle");
    snowman.AddChild(node);
    node = new Cube("Top");
    snowman.AddChild(node);

    root.Accept(printGraph);
}
```