Worksheet #2

Overview

For this worksheet, we will create a class library and then use the resulting dynamic link library (DLL) in a console application. This worksheet is a start on your next programming assignment, where we will build a role playing game (RPG) battle simulation and test it in a WPF application. In this worksheet, we will be introducing an interface for the characters in your RPG and another for each character's attack behavior. This will introduce you to the strategy design pattern. For more information, see Head First Design Patterns by Freeman.

Before You Start

This example is based loosely on character battles in the game Final Fantasy I, implemented in 1976. This was a historical title that turned around the fortunes of a game studio and led to a series that is still going strong today.

This worksheet introduces the strategy design pattern. For more information on this design pattern, read chapter 1 of Head First Design Patterns by Freeman.

Creating a Class Library

You should have your favorite version of Visual Studio 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 (called a Dynamic Link Library) 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 text box, enter your desired hard drive location. Note that this will be on the mounted z: drive in the CSE environment.
4. Name the library RPGInterfaces.
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->RPGInterfaces 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.RoleplayingGameInterfaces (e.g., Maung. RoleplayingGameInterfaces). 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, *RPGInterfaces*, 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 ICharacter.cs. Microsoft naming convention has all interfaces starting with a Capital "I" followed by the name of the class/interface using Pascal case. (Ex: PascalCase)

```
namespace Maung.RoleplayingGameLibrary
{
	public interface ICharacter
	{
		//Properties
		string CharacterClass { get; }
		string Name{get;}
		int Health {get;}

		//Methods
		void PerformAttack(ICharacter target);
		void ReceiveAttack(int damage);
	}
}
```

10. Insert the following code such that your interface looks like the the code above. (Your namespace should be different than mine.) There are many things you may wish to model for a character in a full RPG, but here we will only concern ourselves with a simple battle simulation. Each character has a Name and Character class as string properties which describe the character. He/she also has health which indicates how much damage the character can take before dying. There are only two methods for our battle simulation. A character may perform an attack, or be the target of an attack (ReceiveAttack).
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 IAttack, with the following method:

```
void Attack(ICharacter attacker, ICharacter target);
```

13. OK. That is it. We are done. You can do a Build to ensure that it compiles fine.
14. If you try to do a *Debug->Start without Debugging* you will get an error message. 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.
15. OK, let's recap. We created a class library with the interfaces: ICharacter and IAttack. 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. Code written to interfaces will work even if new implementations are created after your code is compiled and delivered! Your Software Engineering courses will hopefully reinforce this.
16. Edit the Assembly Information to provide your name, organization, description, etc.

Creating another project that implements our design

17. 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 RPGTester and leave the directory location alone.
18. Open the new Project's Property page and change the default namespace to *YourLastname.RPGCore*.
19. In Program.cs, change the namespace to *YourLastname.RPGCore*. Add the following global private variable:

```
private IList<ICharacter> playerParty = new List<ICharacter>();
```

Note: I personally prefer to use interfaces even for internal variables, so I used the IList rather than List

for characters. Returned values should generally (but not always) be interface types, input parameters should almost always be interfaces, private variables as you see fit. If you want to use List, then you can replace IList with List or better yet, replace IList<ICharacter> 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!

20. If you do a build now, you will see there is a compiler error indicating that it does not recognize the type ICharacter. 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 (RPGInterfaces) and vice versa. To fix this, we need to add a reference to either the dll or the project.
21. 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 (RPGInterfaces). Select it and click OK. You may notice that RPGInterfaces has now appeared in the references folder for this project.
22. Rebuild the solution. You will still have the compiler error. However, if you click on the type reference (ICharacter) 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., Maung.RPGInterfaces.ICharacter). 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 playerParty 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.

23. There are many possible ways we could define characters for a role playing game. I will help you implement a couple ICharacter classes for your lab and you can add a couple more. We will start with the Warrior and the Mage classes. Additionally, I will give you some examples of the IAttack class and provide you with a basic combat engine on which you can expand.
24. Right-click on the RPGTester project and select *Add->Class*. Name the class *CharacterBase*. We will use this class to provide some default behavior for characters in our role playing game. 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.
25. Specify that CharacterBase should implement the ICharacter 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.
26. Click on the ICharacter 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 ICharacter interface. All of the methods for ICharacter are implemented with default code that just

throws the "NotImplementedException". You will have to replace this code with your own implementation.

27. 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;
        protected set;
}
```

The interface only requires a **get** property for Name. Internally, we need to be able to set the Name, so we added a set and made its access protected. We could have made this public or private as well. For this class we made it protected, so classes derived from CharacterBase can set it. (In general, avoid making things public unless there is a compelling reason.)

28. Let us create a method for handling anonymous characters that were created without a name. Add the following lines to the top of CharacterBase:

```
protected const string AnonymousName = "Anonymous";
protected static int anonymousCounter = 0;
```

The first string is constant, so it cannot be changed during execution. The static counter holds a single value for all instances of the CharacterBase class (and any classes derived from it). This will allow us to create character names like Anonymous1, Anonymous2, ….

29. Repeat the process for CharacterClass, and Health interface members:

```
public string CharacterClass
{
    get;
    protected set;
}

public int Health
{
    get;
    protected set;
}
```

30. Now, let us fill in the details for the two interface Methods. First, we will fill in PerformAttack. Remember, we are using the Strategy design pattern for attacks, so we will defer the attack to a special instance of IAttack to handle this. Don't worry, it will all come together at the end. Lets create a protected member to hold this class:

```
protected IAttack attackBehavior;
```

Note: I like to have class member methods and variables in certain order for readability:

1. Constants first
2. Private and protected members
3. Properties

4. Constructors
5. Methods

You should develop your own organization and keep your code organized as well.

31. Now we can implement the PerformAttack method:

```
public virtual void PerformAttack(ICharacter target)
{
     attackBehavior.Attack(this, target);
}
```

That's it! As soon as attackBehavior is set, it will implement the appropriate attack. We'll get back to this soon enough.

32. For now, lets move on to implementing some default behavior for receiving an attack. Here is my default implementation:

```
public virtual void ReceiveAttack(int damage)
{
    if (randomNumbers.Next(GameConstants.Instance.DodgeDifficulty) != 0)
    {
        Console.WriteLine(this.Name + " takes " + damage + " damage.");
        Health -= damage;
        if (Health < 0) Health = 0;
    }
    else
    {
        Console.WriteLine(this.Name + " dodges the Attack!");
    }
}
```

This method gives a chance to dodge the attack. If the user dodges, a message is written to the console reflecting this; otherwise, a message indicating the damage taken is written. The damage is subtracted from the character's health. If the health goes below zero, it is set to zero, indicating the character is dead.

There are a couple of things new here. First, we have "GameConstants.Instance.DodgeDifficulty". This is an example of using a singleton class to hold program wide variables. Lets implement it real quick.

33. Create a new class called GameConstants.
34. Now, create a member variable:

```
private const int dodgeDifficulty = 5; // Chance is 1/DodgeDifficulty, so increasing
                                       numbers are more difficult.
```

35. Create a property that returns the dodge difficulty.

```
public int DodgeDifficulty
{
    get { return dodgeDifficulty; }
}
```

Now we have a class where we can set the dodge difficulty, but we don't have a singleton yet.

36. Make the default constructor for this class private:

```
private GameConstants()
{
}
```

37. Make a **static** member variable "instance" to hold the one and only instance of GameConstants:

```
private static GameConstants instance = new GameConstants();
```

38. Create a **static** member property called Instance which returns the one and only instance:

```
public static GameConstants Instance
{
    get { return instance; }
}
```

Now, the only way to get a new GameConstants class is to call Instance, which returns the one and only instance of GameConstants – a singleton!

39. Ok, the last thing is to give CharacterBase a random number generator to determine if the character dodges the attack. (Note: For a cleaner design, you could create a singleton random number generator for your whole program, so all random numbers use the same seed.) Add the following member variable to CharacerBase:

```
protected Random randomNumbers = new Random(); // Quick and dirty random number
                                               // generator for dodge.
```

40. Finally, lets override the Object.ToString method of CharacterBase to give characters a default output. Here is mine:

```
public override string ToString()
{
    return String.Format("{0} the {1} has {2} health.", Name, CharacterClass, Health);
}
```

41. Ok, lets go back to that concept of the Strategy pattern. We need to implement many types of attacks for our various classes. A warrior will attack with a sword and the mage will attack with fire. First we will introduce again the concept of a default attack behavior. Child classes can override this if they have more specifics.
42. Add a new class called NormalAttack and have it implement the IAttack interface:

```
public class NormalAttack : IAttack
{
    protected Random randomNumbers = new Random(); // Simple random number generator
                                                   // for attacks.

    public virtual void Attack(ICharacter attacker, ICharacter target)
```

```
    {
        int damage = GameConstants.Instance.DamageBonus +
                randomNumbers.Next(GameConstants.Instance.DamageRange);
        target.ReceiveAttack(damage);
    }
}
```

43. Go ahead and implement DamageBonus and DamageRange in your GameConstants class just like you have implemented DodgeDifficulty. I have set their values to 5 and 10 respectively (making the default damage "1d10+5" in D20 speak).
44. Now, we need to implement our specific attacks, so create two classes called SwordAttack and FireAttack and have them inherit from NormalAttack.
45. For each of these attacks, we will override the base attack behavior. In a more complex role playing game scenario, each attack may have different damage and different chance to hit, etc., but here with our quick simulation, we are happy with the base class damage, so we can call the base class implementation. Here is my sword attack:

```
public class SwordAttack : NormalAttack
{
    public override void Attack(ICharacter attacker, ICharacter target)
    {
        Console.WriteLine(attacker.Name + " swings a sword at " + target.Name +
                "!");
        base.Attack(attacker, target);
    }
}
```

The difference in attacks is represented here by a Console.WriteLine describing the attack. Again, this is a very simple example. Fill in your fire attack similarly and remember to call the base class implementation for damage.

Now we need to implement some actual classes.

46. Add two new classes to your project: Warrior and Mage.
47. Make both of these classes inherit from CharacterBase.
48. Make a constructor for warrior that takes a name and health as parameters:

```
public Warrior(string name, int health)
{
    this.CharacterClass = "Warrior";
    this.attackBehavior = new SwordAttack();
    this.Name = name;
    this.Health = health;
}
```

A couple of things to notice here. 1) We set the attackBehavior for the warrior to an instance of SwordAttack. If we had other types of warriors, we could create a MaceAttack or HammerAttack, etc. 2) We never created a constructor for CharacterBase and just rely on the default constructor as we don't intend to actually instantiate any instances of CharacterBase by themselves.

49. Go ahead and create a similar constructor for your mage class, switching "Warrior" to "Mage" and SwordAttack to FireAttack.

50. For an API that is easier to use, lets create some more constructors with fewer parameters:

```
public Warrior(string name)
    : this(name, GameConstants.Instance.PlayerHitpoints)
{

}

public Warrior()
    : this(CharacterBase.AnonymousName + (++CharacterBase.anonymousCounter).ToString(),
      GameConstants.Instance.PlayerHitpoints)
{

}
```

Go ahead and implement PlayerHitpoints as an integer in your GameConstants class. I used a value of 50. Note the use of the keyword **this** to call the previous constructor we created. Here we also finally make use of our AnonymousName and anonymousCounter properties. Make sure to repeat this for the Mage class.

51. Now compile and build your program and fix any errors you encounter.

Testing your application

Lets implement a simple test main to see how things work:

```
static void Main(string[] args)
{
    ICharacter player1 = new Mage("Gandalf");
    ICharacter player2 = new Warrior("Boromir");

    Console.WriteLine(player1);
    Console.WriteLine(player2);

    player1.PerformAttack(player2);

    Console.WriteLine(player1);
    Console.WriteLine(player2);

    player2.PerformAttack(player1);

    Console.WriteLine(player1);
    Console.WriteLine(player2);

    Console.ReadLine();
}
```

Your output should look something similar to this:

Gandalf the Mage has 50 health.
Boromir the Warrior has 50 health.
Gandalf launches a fireball at Boromir!
Boromir takes 8 damage.

Gandalf the Mage has 50 health.
Boromir the Warrior has 42 health.
Boromir swings a sword at Gandalf!
Gandalf takes 8 damage.
Gandalf the Mage has 42 health.
Boromir the Warrior has 42 health.