|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | *.NET Architecture and Design 2.0* | |  | | |
| |  | | --- | |  | | |  |  | | --- | --- | | [Modules](file:///C:\Users\Pinku%20Surana\Documents\DevelopMentor\Architecture\lab-repo\coursebook.htm) |  | | |  | |

|  |  |
| --- | --- |
| **Lab 1: Architecture**  **Estimated time for completion: 1 hour**  **Goals**   * Organize common classes in logical groups and namespaces * Create a solution structure which represents the games layers and packages * Implement an interface for the Game class   **Overview**  In all of the labs this week, we’re going work with an application named Developoly (develop-oly). It is a partial (incomplete) implementation of a game based upon the popular game Monopoly.  As the labs progress we will refactor this application into a distributed application with client and server pieces.  We will then explore different ways for the client to communicate with the server.  Why use a game?  Well, we have all built typical business applications.  Using a game can make things a little more entertaining while still developing the *same skills* needed to build boring business applications.  Many of you have probably endured many games of Monopoly and know the rules.  If you have never played or just want to refresh your memory, see the file *Game Rules.htm* in this folder for a quick overview.   |  | | --- | | A quick administrative note: Throughout this course, the lab files are divided into “Before” and “After” sets of files.  The “Before” files provide the starting point for the work you will do in the lab.  These are the files that you will edit.  The “After” files are solution files.  They provide an example of how to complete the lab.  The Before/After identifier may be on the folder or on the file name (the Visio files do this). |   In this exercise we are going to move the classes into several namespaces.  The namespaces will help us organize the code.  It highlights the purpose of a set of classes.  It also becomes useful in identifying layers.  A common practice is to have the code files/folders duplicate the class/namespace hierarchy.  We will do this also.    **Step 1:  Explore the Developoly Application**  We do not expect you to write the Developoly game.  There is not enough time this week for that.  Instead, we have provided an implementation to start with.  It is not even close to a complete implementation of the game but it is enough for our purposes.  This is the code we will be working with, more or less, all week.  Your first task is to explore the code.  You’re going to be working with it all week so you may as well start looking at it now.  Note that it is not a distributed application yet.  That will change in later labs.   1. In the Before directory, open the Developoly solution file and make sure the game runs. 2. Play the game.            First add players (with unique names)           When you have added all your players, click Start.           For each player click throw.  For each throw, you are presented with a throw result dialog. In this dialog you should be able to purchase a property, play chances and community chests.           To “pass the dice” you must manually click on the next player.           Play long enough to get a feel for the game.   1. Look at the code.  You can use the UML diagram to get a feel for how things are related.  The diagram can be found at: *01\_Developoly.Before.vsd*   You should note the following:           The Game class is the central “business logic” class.  How does it get the game data?  How would you use a different source for the data?           Who talks to the game class?  Who does the Game class talk to?           If you were to make this a distributed application, where would you break things apart?    **Step 2:  Identifying Layers**  Organizing the code will make it easier to understand and can also help identify logical points to divide the application into pieces.   1. Look at the Before UML diagram. It has been color coded to identify UI classes and business classes.  In a more complex application we may break these down into more specific categories but for our application this will work fine. 2. We know that that we would eventually like to divide the application into pieces that can be distributed.  The two categories that we have so far look like a good start.  However, there are touch points between these categories.  What are the touch points?            (Look for a point where an item of one color has an arrow to an item of another color.) 3. The definitions of the types used at these touch points would need to be available to both sides.  How could we handle that? 4. Let’s define a new category called “Common” to represent the items that need to be available to both sides.  What should we include in this category? 5. We can move items into “Common” from the other categories.  We can also create new items.  In general, if the touch point is more of a data class – no real layer logic – then we can move it into common.  If the class has layer specific logic then we probably do not want to move it into common.  For example, the Game class is a touch point.  This is a business layer class with business logic.  If we put Game in common then we are giving business layer logic to the UI.  We do not want to give the Game class to the UI when we make that application distributed but we have to give the UI something.  How do we handle that? 6. The idea is to separate the client and business code so that the only dependencies are in common.     Think about these questions for a moment and then move on.    **Step 3:  Client Namespace - Presentation**  The first namespace that we are going to define is called “Client” under the “Developoly” namespace that the project uses.   We will create a directory for each namespace and move the classes into it.  This keeps the File/Folder relationship the same as the Class/Namespace relationship.  (See [1] at the bottom of this file for an explanation of the folder names.)   1. In Solution Explorer create a folder named “Developoly.Client”  (right click on the project, select Add/New folder) 2. Which classes should go in this namespace?  What classes control the UI?  In Solution Explorer, drag-and-drop the class files into the new folder. 3. Did you select the Form classes?  How about the App class? 4. We have moved the files but we still need to move the classes into the namespace. Update the class files to place the classes in the namespace “Developoly.Client”. 5. Moving a class into a namespace causes its name to change.  Anyone using the class must know about the new name.  Add the appropriate using/imports statements to files that use the classes you just moved into a namespace. 6. Make sure that the code compiles.   You now have the presentation classes identified.  You can check your work by looking at the ClassView tab and making sure that the class structure looks similar to the file structure.      **Step 4:  Common Namespace**   The classes in the Client namespace will be part of the client when this becomes a distributed application.  As mentioned above, the touch points between the client and business layers must be on both sides.  The Common namespace will hold these common items.   1. In Solution Explorer create a folder named “Developoly.Common” 2. Which classes should go in this namespace?  What are the touch points between client and Business?  Move the common classes into the new directory. 3. In general, all the touch points should be moved into Common.  However, Game is a touch point and that is definitely a Business class and we would not want that on the client.  We will deal with the Game class in the next step. 4. We have moved the files.  Now we must move the classes into the namespace.  Update the class files to place the classes into the namespace “Developoly.Common”. 5. Add the appropriate using/imports statements to the files that use the classes you just moved into the namespace. 6. Make sure that the code compiles.   You now have common classes identified.      **Step 5: Create Interface IGame**  Remember the note above about Game being a touch point?  How did you decide to handle that?  In this part of the exercise we’re going to decouple the client and business layers by adding an interface. By using an interface we are separating the client from the implementation of the business classes, which will allow the layers to evolve independently.   1. In the Developoly.Common directory, create a new class file called IGame. 2. In the file, define an interface called IGame. 3. IGame should have method signatures that match each of the public methods and properties in the existing Developoly.Game class. 4. This is a common definition so place it in the namespace Developoly.Common 5. Update the Developoly.Game class to implement the interface. 6. The client UI classes need to be updated to use the interface instead of the using the Game class directly.  Find where they are using Game and change the field type to type IGame.  (An easy trick for this is to let the compiler identify these places for you.  Rename the Game class to something like Game2 and recompile.  Look for errors in the client classes and change those to use the interface.  Then change Game2 back to Game.)  ex. Game g = new Game();  becomes IGame g = new Game();   Fix any namespace reference problems and test your work.  As this point the Common namespace should be complete.  Look at the Client classes.  If the only classes that they refer to are in the Client or Common namespaces then this is correct.  (There is one exception to this – there will be one line of code that creates a new Game object.  We will remove this dependency when we make this a distributed app)    **Step 6:  Business Namespace**  The last namespace that we are going to define is called “Business” under the “Developoly” namespace that the project uses.   1. In Solution Explorer create a folder named “Developoly.Business” (right click on the project, select Add/New folder) 2. Identifying classes which are logically business components and place them in the directory.  (You can look at the solution for the ones we selected.) 3. Update the class files to place the classes in the namespace “Developoly.Business”. 4. Add the appropriate using statements to files that use the classes you just moved into a namespace. 5. For C# only: The xml resource file is location dependent.  Look in GameDataResource class for how to update the file path used in the lookup. 6. Make sure that the code compiles.   You now have the business classes identified.  Now that we’ve completed this task review the structure of the developoly game and note that from a logical perspective we’ve refactored the application into what are essentially layers.  As we progress through the labs, the new solution structure is going to help us understand the architecture of our distributed application.   * The After UML diagram shows the new structure.     **[1] Note about folder names**  Naming folders after namespaces is common and useful.  The question is how you handle multiple levels of nesting.  There are two main approaches.  There is no major advantage to either one.  In most cases it boils down to preferences or existing standards.  The first approach is to nest the folders at the same level as the namespace.  For example, if you have class Customer in the namespace Foo.Bar.Baz, then the folder structure would have a folder named Foo, containing a folder named Bar, containing a folder named Baz, containing the class Customer.  So the folders look like this:        The second approach is to have a compound folder name that matches the namespace.  The example above would have a single folder named Foo.Bar.Baz, containing the class Customer.  It would look like this:        For the labs, we will use the second approach.  This keeps the files one click away.  The trade-off is that the tree has more namespaces listed which can make it more cluttered.  That is not a problem in our case. |
| Solutions: [After](file:///C:\Users\Pinku%20Surana\Documents\DevelopMentor\Architecture\lab-repo\work\01%20Architecture\C#_After) |