**Lab 6 – CS 3340**

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

**Lab Objectives**

1. Learn the term: round-trip, forward and reverse engineering.
2. Use the Class Designer to implement forward and reverse engineering.
3. Implement 1-many and use Session to access it across multiple pages.

**Lab Organization**

There are 7 stages to complete Lab 6.

|  |  |
| --- | --- |
| Stage | Description |
| 0 | OO Modeling |
| 1 | Install Class Designer |
| 2 | Add a Class |
| 3 | Add a 1-many Relationship |
| 4 | Test Classes |
| 5 | Put Person Object In Session |
| 6 | Package Assignment for Submission |

**Stage 0 – OO Modeling**

(Read, no action required)

1. The *Class Designer* allows you to create class diagrams using a visual tool. When you add a class to the diagram, a code file will also be created. As you add classes, properties, methods, *etc,* to the diagram, the code is updated. This is called *forward engineering.* If you modify the code directly, the diagram will automatically be updated. This is called *reverse engineering*.
2. How exactly is *forward engineering* implemented?

* When you add a class, *Foo*, then *Foo.cs* is created.
* When you add a *string Name* property to the diagram, the code for the property (with default get and set) is added to *Foo.cs*.
* When you add a method to a diagram, you supply the full signature. The corresponding code is added to *Foo.cs*.

The Class Designer is weak in its ability to write the code for the *get* in a property. However, it is a minor inconvenience as we simply delete the generated line of code and write it ourselves. I believe there is an editable template that controls how the code is generated which could perhaps fix this issue.

1. What exactly is *reverse engineering*? It is really two things.

* It is simply a concept, that if we modify a man-made object (code), its design (class diagram) will be updated also. The Class Designer does this: once you have a class on your diagram, you can write properties, methods, *etc.* by hand and the diagram will be updated.
* It is a software engineering activity where we have a man-made object (code for a system, component, *etc* composed of classes) and we reveal its design (class diagram). In other words, we have a code base and we want to learn how it is organized.

We will not need to do this in this class, but it is very useful when trying to understand a code base with no documentation. There are many tools for reverse engineering. However, I cannot get the Class Designer to accomplish this activity. In other words, of you have a class diagram and you add a class to the code base, you can’t get it on the diagram! Surely there is a way, though. Thus, you must create the shell of the class in the Class Designer, then you can write all the code by hand if you choose.

1. Taken together, forward and reverse engineering are called [*round-trip engineering*](https://en.wikipedia.org/wiki/Round-trip_engineering)*.*

**Stage 1 – Install Class Designer**

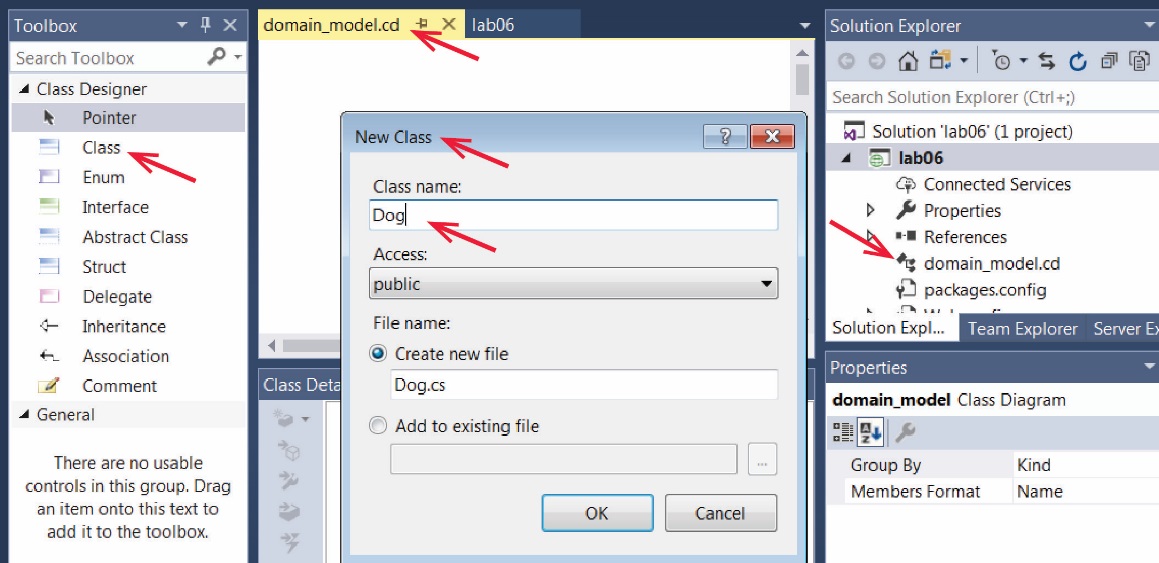
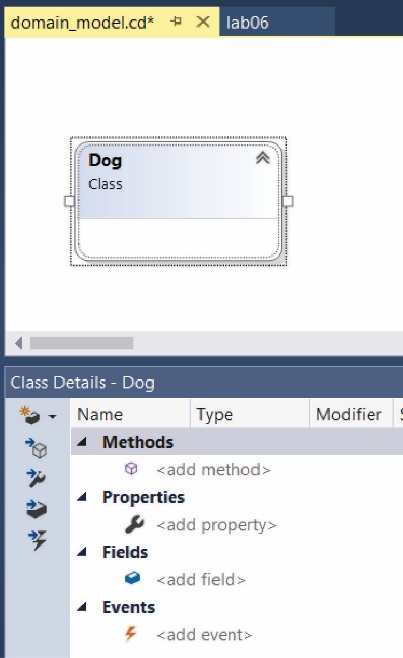
1. Create your *lab06* project (solution folder must be named *lab06\_lastName*).
2. Check to see if you have the Class Designer installed (probably don’t) by doing the following:
3. Right-click the project node in the SE and choose: Add, New Item.
4. In the menu on the left, choose: Visual C#, General.
5. You should see, “Class Diagram”. Search for it if necessary.
6. If you find, “Class Diagram”, press Cancel and move on to Stage 2; if you don’t install it using these instructions:

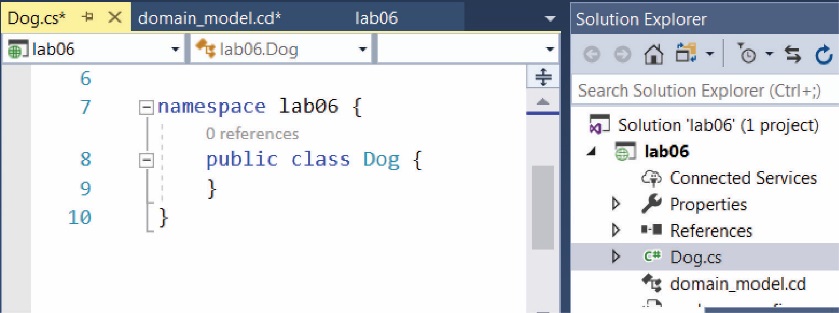
[How to: Add class diagrams to projects](https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019)

Note: all you need to do is the first 3 numbered steps, in the first section, “Install the Class Designer component”, then move on to Stage 2.

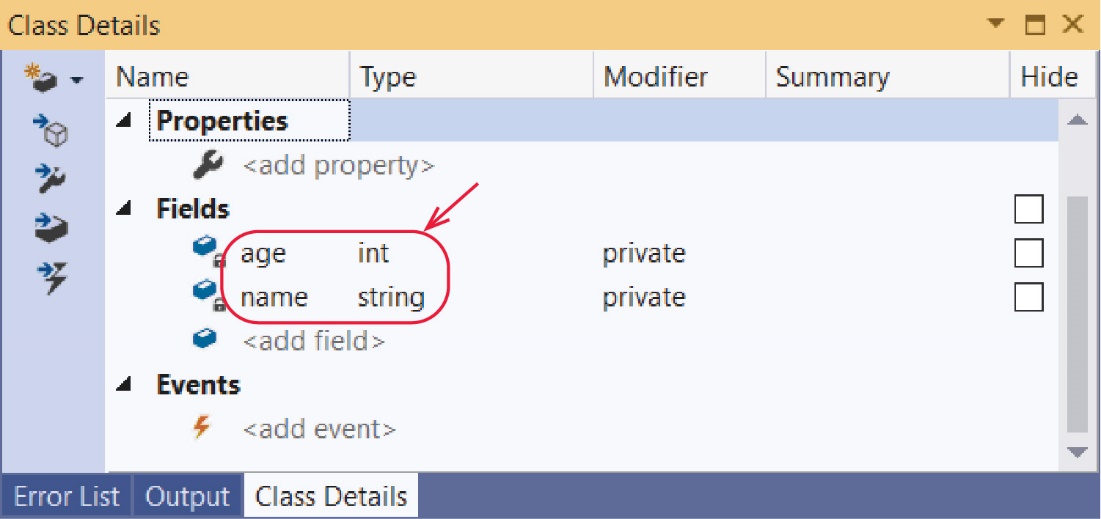
1. OPTIONAL – [How to Use the Class Diagram Tool in Visual Studio](https://www.youtube.com/watch?v=r6ylaQ87SNQ) – 12 minute video.
2. OPTIONAL – [Generate a constructor in Visual Studio](https://docs.microsoft.com/en-us/visualstudio/ide/reference/generate-constructor?view=vs-2017) – This doesn’t have anything to do with class diagrams, nor this lab, but it is a useful technique.

**Stage 2 – Add a Class**

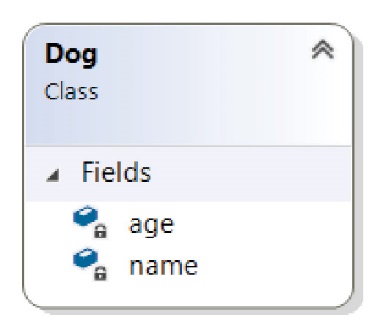
1. Create a class diagram. Do the following:
2. Right-click the project node in the SE and choose: Add, New Item, Visual C#, General.
3. Supply the name: *domain\_model*
4. Choose: Class Diagram and then Add
5. Add a class. Do the following:
6. Drag a class from the Toolbox onto the design surface, provide the name, “Dog”, and choose: OK. Note the items indicated by the arrows.
7. The display should look as shown on the right. Note the *Class Details* pane in the lower part of the window. This is where we will add methods, *etc.*
8. Open the *Dog* class from the SE and verify that the designer has generated some initial code as shown below.



1. Add some fields. Do the following
2. Redisplay the class diagram and select the Dog icon/box. Add *name* and *age* fields in the Class Details pane as shown below.



1. The class diagram should appear as shown below. Save the class diagram.



1. Redisplay the code which should similar as shown below.

namespace lab06\_gibson {

public class Dog {

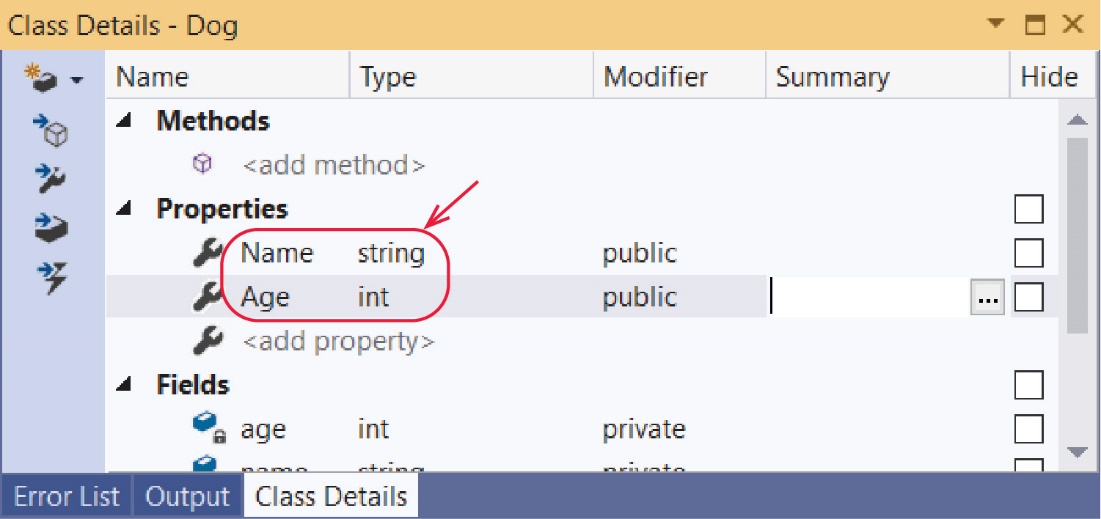
private int age;

private string name;

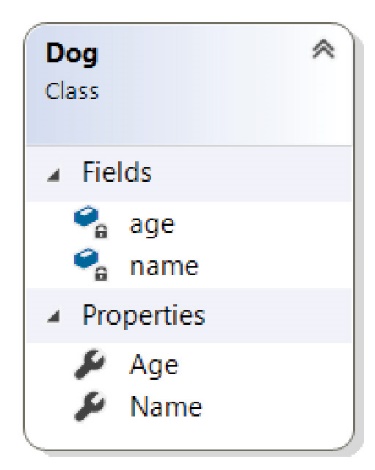
}

}

1. Add some properties. Do the following
2. Redisplay the class diagram and select the Dog icon/box. Add *Name* and *Age* properties as shown below. Save the class diagram.



1. The class diagram should appear as shown below. Save the class diagram.



1. Redisplay the code which should similar as shown below.

|  |  |
| --- | --- |
| public class Dog {  private int age;  private string name;  public string Name {  get => default;  set {  }  }  public int Age {  get => default;  set {  }  }  } | Note:   * A C# *property* encapsulates a *getter* and a *setter*. * The getter is using a lambda expression. |

1. Replace the code in the getter and setter so that they appear as shown below. Note: *value* is a keyword in C# and represents the parameter for the setter.

|  |  |
| --- | --- |
| public string Name {  get => name;  set => name = value;  } | public int Age {  get => age;  set => age = value;  } |

1. Test your class. Do the following
2. Create a web form named, *default.aspx*, add a multi-line textbox named, *txtMessage*.
3. Type this code into *Page\_Load*

Dog d = new Dog();

d.Name = "Chaps";

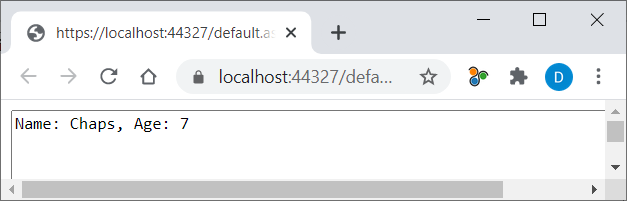
d.Age = 7;

txtMessage.Text = "Name: " + d.Name + ", Age: " + d.Age;

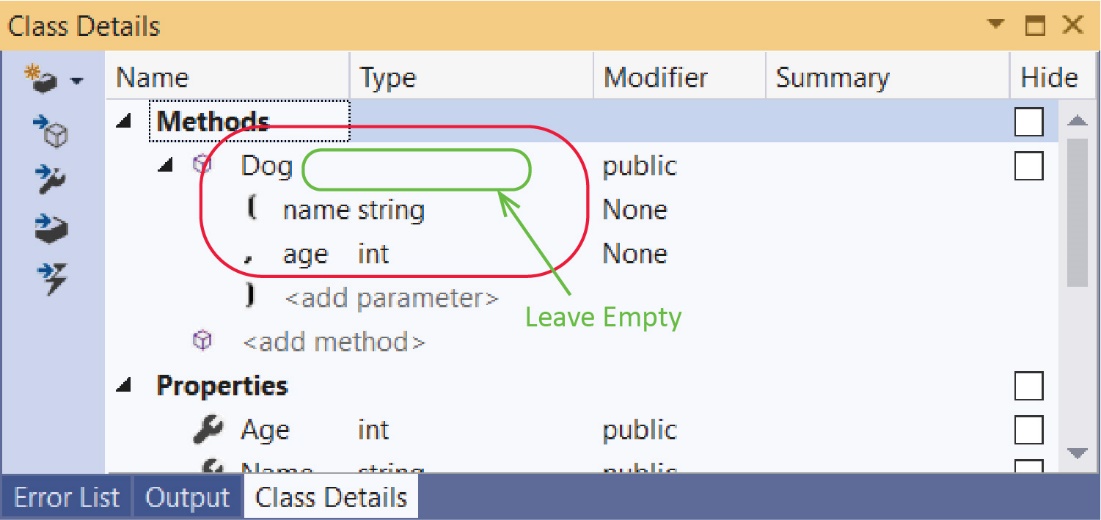
Note:

* d.Name = "Chaps" is invoking the setter for the *Name* property. In Java, it would be: d.setName(“Chaps”)
* d.Name in the output string is invoking the getter for the *Name* property. In Java, it would be: d.getName()

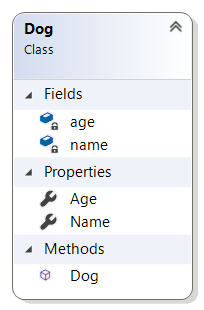
1. Run the page and make sure it works correctly.



1. Add a constructor. Do the following
2. To create a constructor, you add it as a method, but leave the (return) *Type* empty. Supply the values shown below, including the parameters. Leave the “Type” empty for a constructor.



1. The class diagram should appear as shown below. Save the class diagram.



1. Redisplay the code which should be similar to what is shown below.

public Dog(string name, string age) {

throw new System.NotImplementedException();

}

1. Replace the code in the constructor so that it appears as shown below. Note: the two lines are invoking the setters for the *Name* and *Age* properties, respectively.

public Dog(string name, int age) {

Name = name;

Age = age;

}

1. Replace the code in *Page\_Load* with

Dog d = new Dog("Chaps", 6);

d.Age = 7;

txtMessage.Text = "Name: " + d.Name + ", Age: " + d.Age;

1. Run the page and make sure it works correctly.
2. Add a method by writing code. Do the following:
3. Add this method below the properties in *Dog.cs*:

public string Bark(int numTimes) {

string msg = "";

for (int i = 0; i < numTimes; i++) {

msg += "bark ";

}

return msg;

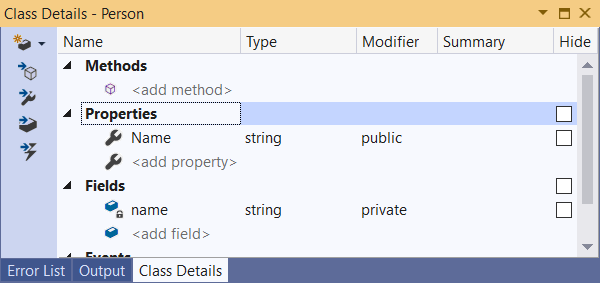
}

1. Open the class diagram and verify that you see the Bark method in the diagram. In the Class Details pane, select and expand the bark and verify its parameter. The ability of the *Class Designer* tool to translate code into the diagram is called *reverse engineering*.

|  |  |
| --- | --- |
|  |  |

**Stage 3 – Add a 1-many Relationship**

1. Do the following:
2. Use the class designer to add a class named *Person*.
3. Add a *name* field and a *Name* property



1. Modify the code for the property:

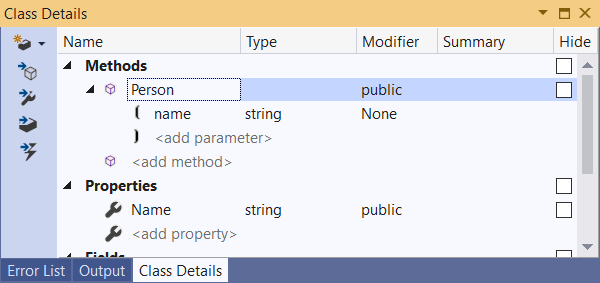
public string Name {

get => name;

set => name = value;

}

1. Add a constructor that takes a name (you will need to modify the code once it is generated)



1. Modify the code for the constructor:

public Person(string name) {

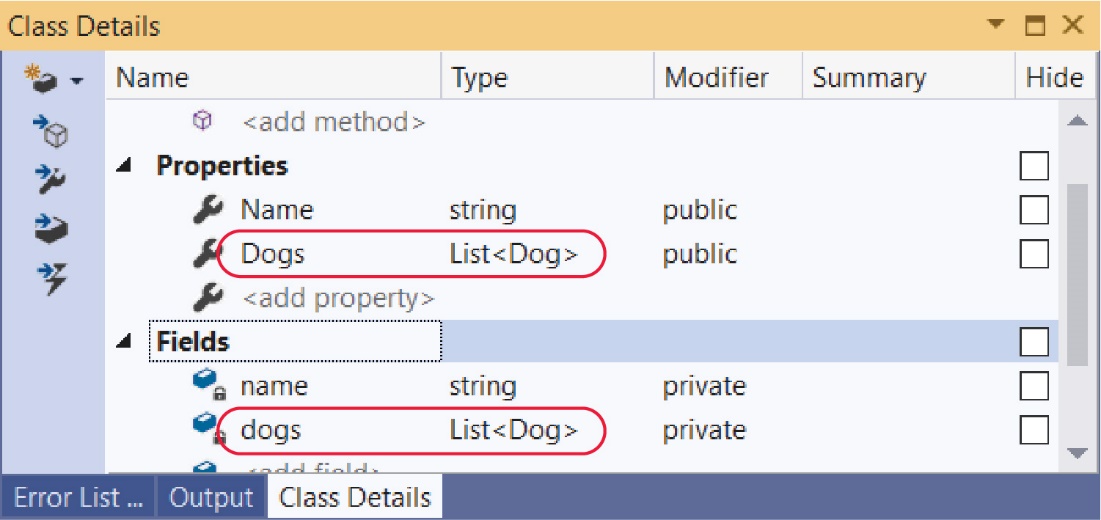
Name = name;

}

1. Next, we will make a 1-many relationship: A *Person* will have many *Dog* instances. Do the following:
2. Add this *using* statement to the top of *Person.cs*

using System.Collections.Generic;

1. Add a *dogs* field of type, *List<Dog>* and a *Dogs* property of type, *List<Dog>* as shown below:



1. Open the *Person* class and edit the *Dogs* property so that it looks as shown below. For this property we only define a getter.

public List<Dog> Dogs {

get => dogs;

}

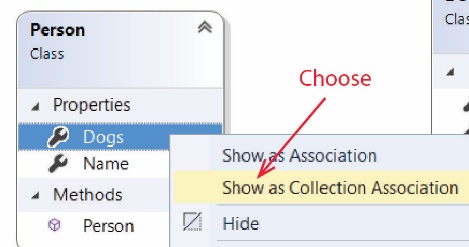
1. Modify the code in the constructor so that it initializes *Dogs*:

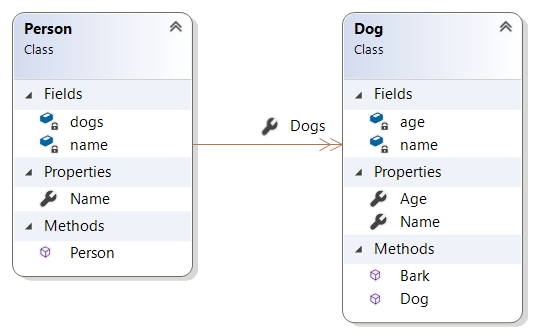
public Person(string name) {

Name = name;

dogs = new List<Dog>();

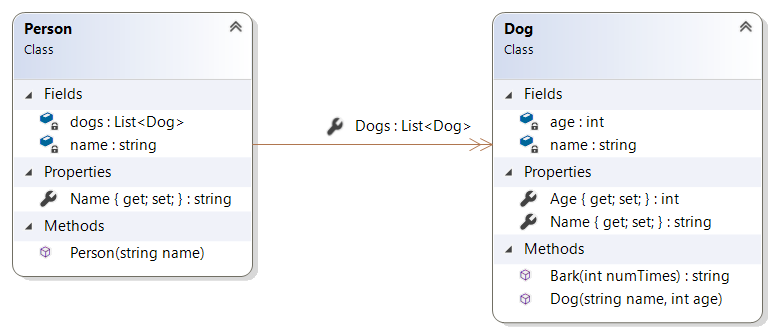
}

1. The *Dogs* property can be visualized as an *association* by showing a directed line from the *Person* class to the *Dog* class*.* Do this:
2. Right-click *Dogs* and choose: *Show as Collection Association*
3. The result will appear as shown below:



The *Dogs* property is now displayed as a 1-many association and is removed from the list of properties. Note, the notation for “many” (-->>) is different than the UML standard (-->\*).

1. The full signature of methods and the data types for fields can be displayed in the class diagram.Do the following:
2. Display the class diagram, and click anywhere in the whitespace outside any class.
3. Right-click, and choose: Change Members Format, Display Full Signature
4. The result will appear as shown below (you will need to stretch the boxes):



1. Add a comment to your diagram with your name. The Comment selector is in the Toolbox.
2. Export your image. Do the following:
3. Right-click an empty space on diagram and choose: Export Diagram as Image
4. The default *Export Location* is your project folder.

Note: Oddly, the dialog does not let you specify a filename! The name of the image is the same as the name of the class diagram file, except with a different extension, *e.g. domain\_model.jpg*. So, if you want to make two versions of a diagram, as we are about to, you need to rename the one we just created (we will do that shortly).

1. Choose: Export
2. Add image to the SE and rename. Do the following:
3. Right-click the Project node in the SE, and choose: Add, Existing Item
4. If necessary, navigate to the image, select it, and choose: Add
5. Rename the image in the SE by adding, “\_1” to the end of the file name, *e.g domain\_model\_1.png*
6. Add the image to the *default* page. Do the following:
7. Display *default.aspx*
8. Select the image in the SE and drag to below the multiline textbox.
9. View the Source for the age and not the HTML *img* tag

<img alt="class diagram" class="auto-style1" src="domain\_model\_1.png" />

**Stage 4 – Test Classes**

1. Create a web form named, *default2.aspx*, add a multi-line textbox named, *txtMessage*.
2. Open *default2.aspx.cs* and replace the code in *Page\_Load* with:

Dog d1 = new Dog("Chaps", 6);

Dog d2 = new Dog("Gigi", 4);

Dog d3 = new Dog("Zoro", 11);

Person p = new Person("Jackson");

p.Dogs.Add(d1);

p.Dogs.Add(d2);

p.Dogs.Add(d3);

foreach (Dog dog in p.Dogs) {

txtMessage.Text += dog.Name + ", " + dog.Age + "\n";

}

1. Run the page and verify that the output is correct.

**Stage 5 – Put Person Object In Session**

We are going to create two pages: *create* and *display*. The *create* page allows the user: (a) to create a person, (b) create a dog and add it to the person, repeatedly. The person will be put in session so that it properly keeps track of each new dog that is added. The *create* page also has a button to “Display All on Other Page”, which redirects to the *display* page. There, the person is retrieved from session and displayed.

1. Create a web form named, *create.aspx*, and replace all except the top line with:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title></title>

</head>

<body>

<form id="form1" runat="server">

<div>

<p><asp:Label ID="Label1" runat="server" Text="Name"></asp:Label>&nbsp;

<asp:TextBox ID="txtNamePerson" runat="server"></asp:TextBox>&nbsp;

<asp:Button ID="btnCreatePerson" runat="server" Text="Create Person" OnClick="btnCreatePerson\_Click" />

</p>

<p><asp:Label ID="Label2" runat="server" Text="Name"></asp:Label>&nbsp;

<asp:TextBox ID="txtNameDog" runat="server"></asp:TextBox>&nbsp;

<asp:Label ID="Label3" runat="server" Text="Age"></asp:Label>&nbsp;

<asp:TextBox ID="txtAgeDog" runat="server"></asp:TextBox>&nbsp;

<asp:Button ID="btnCreateDog" runat="server" Text="Create Dog" OnClick="btnCreateDog\_Click" />

</p>

<p>&nbsp;

<asp:Button ID="btnDisplayAll" runat="server" Text="Display All" OnClick="btnDisplayAll\_Click" />&nbsp;<asp:Button ID="btnDisplayAllOtherPage" runat="server" Text="Display All Other Page" OnClick="btnDisplayAllOtherPage\_Click" />&nbsp;<asp:Button ID="btnStartOver" runat="server" OnClick="btnStartOver\_Click" Text="Start Over" />

</p>

<asp:TextBox ID="txtMessage" runat="server" Height="175px" TextMode="MultiLine" Width="558px"></asp:TextBox>

</div>

</form>

</body>

</html>

1. Open the code behind file and replace the *create* class (everything below the namespace, except the last brace) with:

public partial class create : System.Web.UI.Page {

private Person p;

protected void Page\_Load(object sender, EventArgs e) {

if (!Page.IsPostBack) {

if(Session["person"]==null) {

txtMessage.Text = "Create a Person";

}

else {

// If returning from display page, pull Person from Session

p = (Person)Session["person"];

}

}

else {

// If postback, pull Person from Session

p = (Person)Session["person"];

}

}

protected void btnCreatePerson\_Click(object sender, EventArgs e) {

// Create Person object

Person p = new Person(txtNamePerson.Text);

// Save object to Session.

Session.Add("person", p);

txtMessage.Text = "";

txtNameDog.Focus();

}

protected void btnCreateDog\_Click(object sender, EventArgs e) {

// Make sure there is a Person object

if(p==null) {

txtMessage.Text = "No person has been created yet";

return;

}

// Create Dog and add to Person

int age = Convert.ToInt32(txtAgeDog.Text);

Dog d = new Dog(txtNameDog.Text, age);

p.Dogs.Add(d);

// Must update Session after change to Person

Session.Add("person", p);

// Clean up UI

txtNameDog.Text = "";

txtNameDog.Focus();

txtAgeDog.Text = "";

}

protected void btnDisplayAll\_Click(object sender, EventArgs e) {

foreach (Dog dog in p.Dogs) {

txtMessage.Text += dog.Name + ", " + dog.Age + "\n";

}

}

protected void btnStartOver\_Click(object sender, EventArgs e) {

Session.Clear();

Response.Redirect(Request.RawUrl);

}

protected void btnDisplayAllOtherPage\_Click(object sender, EventArgs e) {

Response.Redirect("display.aspx");

}

}

1. Run the page. Everything should work except the button to go to the other page.
2. Create a web form named, *display.aspx*, and replace all except the top line with:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title></title>

</head>

<body>

<form id="form1" runat="server">

<div>

Me &amp; My Dogs

<asp:Button ID="btnCreateMore" runat="server" OnClick="btnCreateMore\_Click" Text="Create More Dogs" />

<br />

<asp:TextBox ID="txtMessage" runat="server" Height="245px" TextMode="MultiLine" Width="481px"></asp:TextBox>

<br />

</div>

</form>

</body>

</html>

1. Open the code behind file and replace the *display* class (everything below the namespace, except the last brace) with:

public partial class display : System.Web.UI.Page {

private Person p;

protected void Page\_Load(object sender, EventArgs e) {

if (Session["person"] == null) {

txtMessage.Text = "Didn't find a Person";

}

else {

p = (Person)Session["person"];

foreach (Dog dog in p.Dogs) {

txtMessage.Text += dog.Name + ", " + dog.Age + "\n";

}

}

}

protected void btnCreateMore\_Click(object sender, EventArgs e) {

Response.Redirect("create.aspx");

}

}

1. Run the page. Everything should work including the button to go to the other page. Test thoroughly.

**Stage 6 – Package Assignment for Submission**

1. Close VS and zip your *lab06\_lastName* project folder and submit on Blazeview in the *Lab 06* dropbox.