



SCULPTURE TOOLKIT

LANGUAGE WORKBENCH CHALLENGE 2012

This Paper shows the Sculpture Toolkit solution of the language workbench challenge 2012 assignment.

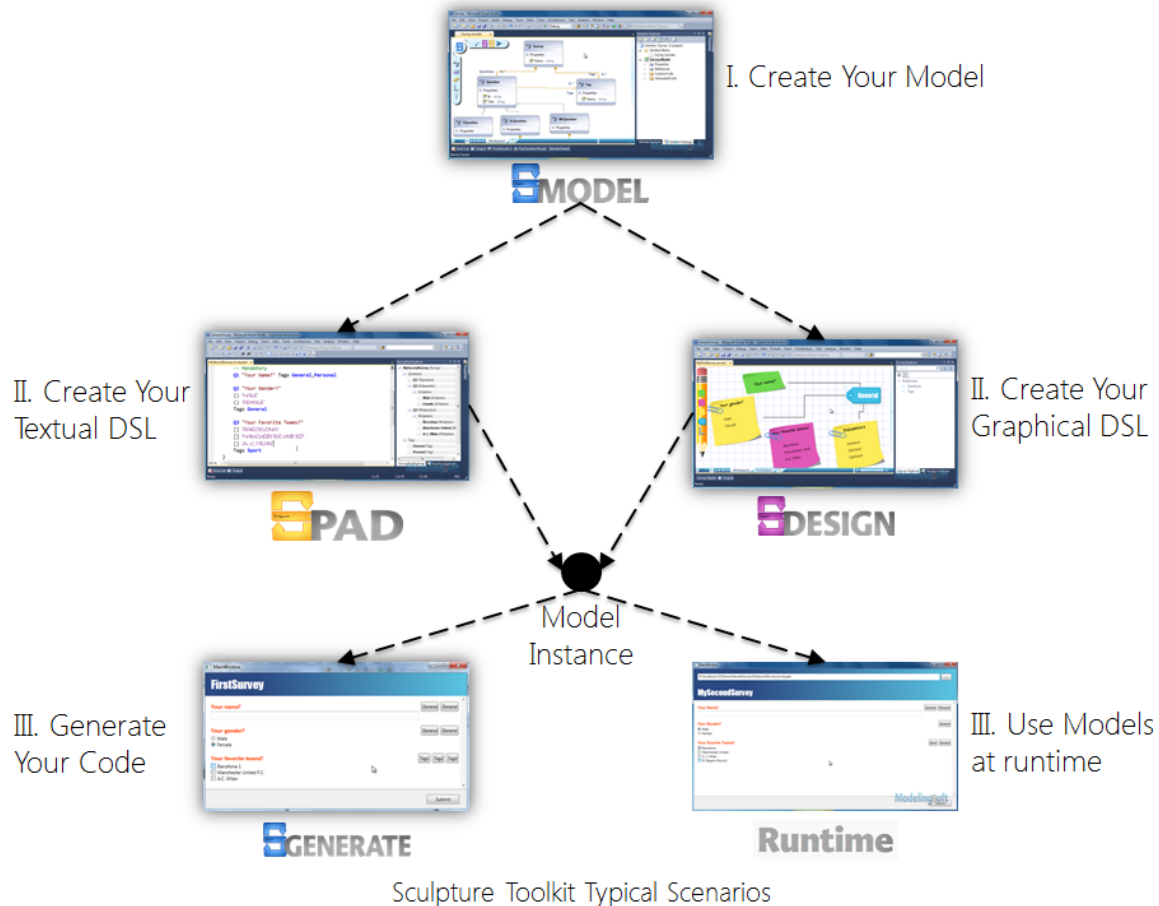
CONTENTS

Introduction to Sculpture Toolkit	2
Assignment.....	3
Create Your Model.....	4
Test your Model.....	10
Create your Designer	12
Generate Artifacts from your Model	21

INTRODUCTION TO SCULPTURE TOOLKIT

Sculpture Toolkit is a modeling platform that contains a set of generative components and runtime infrastructures for developing .NET model-based solutions.

Sculpture Toolkit enables you to create your own models 'S-MODEL', textual languages 'S-PAD', graphical designers 'S-DESIGN', and code generators 'S-GENERATE'.



Let's start with the first component 'S-MODEL' which enables you to create your own models. With S-Model designer, you can describe your domain model elements such as Domain Types, Domain Classes, Domain Properties, Domain References, and Domain Enumerations. S-Model runtime empowers your models with many out-of-the box functionalities like Meta-Data definitions, Serialization, Validation, Copying, Comparing, and Transaction Management.

Based on your model you can use the second component 'S-DESIGN' to create your own custom graphical designer. Visually configure your designer Shapes, Connectors, and

Toolbox, then generate rich, fully customizable, and extensible WPF designer that works natively inside your Visual Studio.

You can also use the third component 'S-PAD' to create your own custom textual language. Write your language tokens, styles, and syntaxes, then generate grammar, parser, and full customizable language editor with different facilities like intellisense, text colorization, and real-time syntax checking.

Finally you can use the fourth component 'S-GENERATE' to create your own custom code generator. This generator has the ability to understand your model instances and generate artifacts. Visually author your generation process scenario which may have more than one template, each with multiple options. S-Generate utilize and extend Microsoft T4 templates to easily write your templates. S-Generate also provide an API to facilitate Visual Studio core automation like project and project item construction.

With these integrated and unique components you can create full .NET model-based solutions in a very easy and straight forward.

ASSIGNMENT

This paper shows the Sculpture Toolkit solution of the language workbench challenge 2012 assignment.

The assignment can be found at

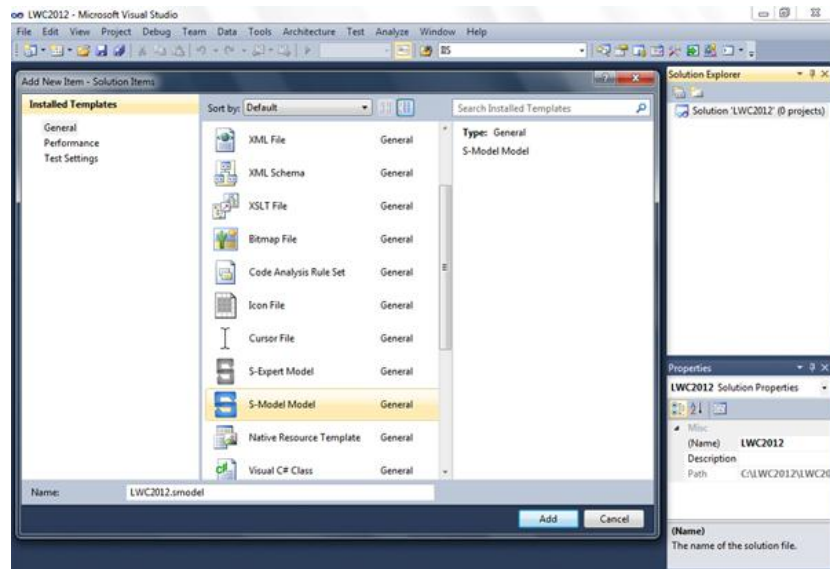
http://www.languageworkbenches.net/index.php?title=LWC_2012#The_assignment

The goal is to enable piping & instrumentation domain experts to build a model of a home heating system.

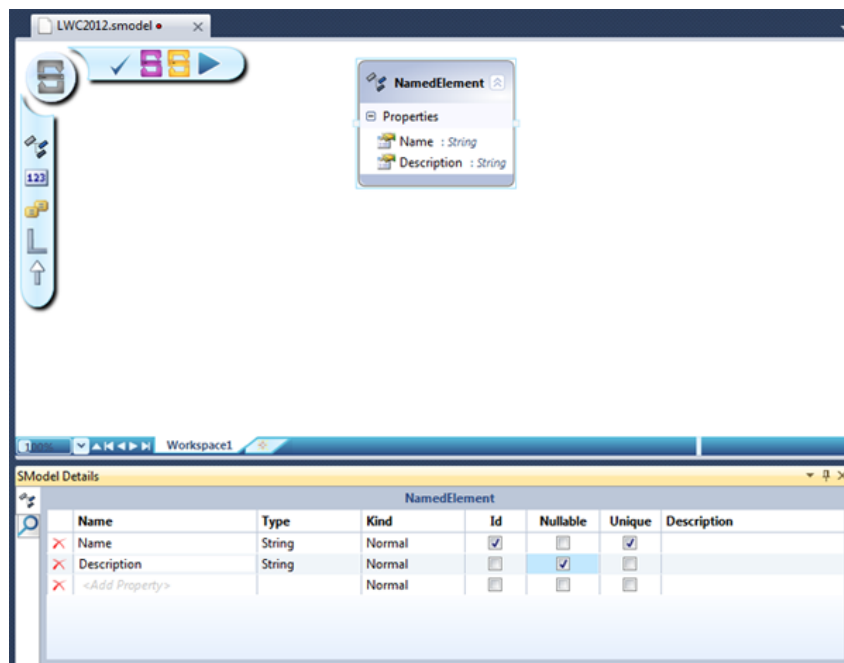
CREATE YOUR MODEL

We will try to keep the domain as simple as possible to concentrate on the tool capabilities. We will start by creating our model.

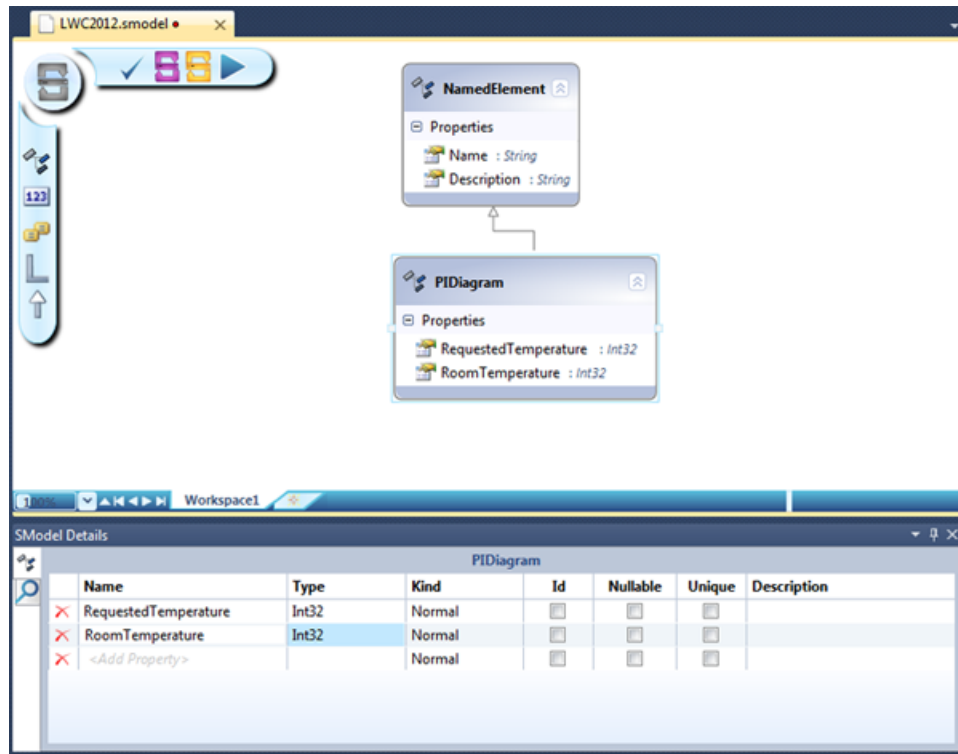
From the Visual Studio 2010, create new Blank Solution, and then add new S-Model.



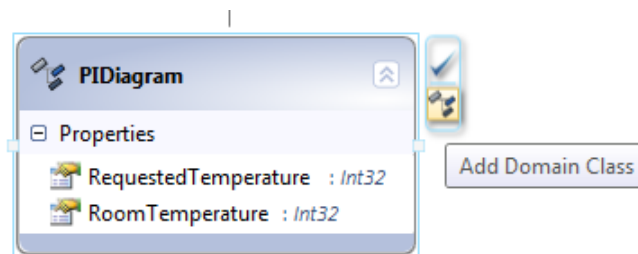
We will start our model with an abstract class **NamedElement**, which contains two properties **Name** and **Description**. The property **Name** is the Identifier property. The property **Description** allows null values.

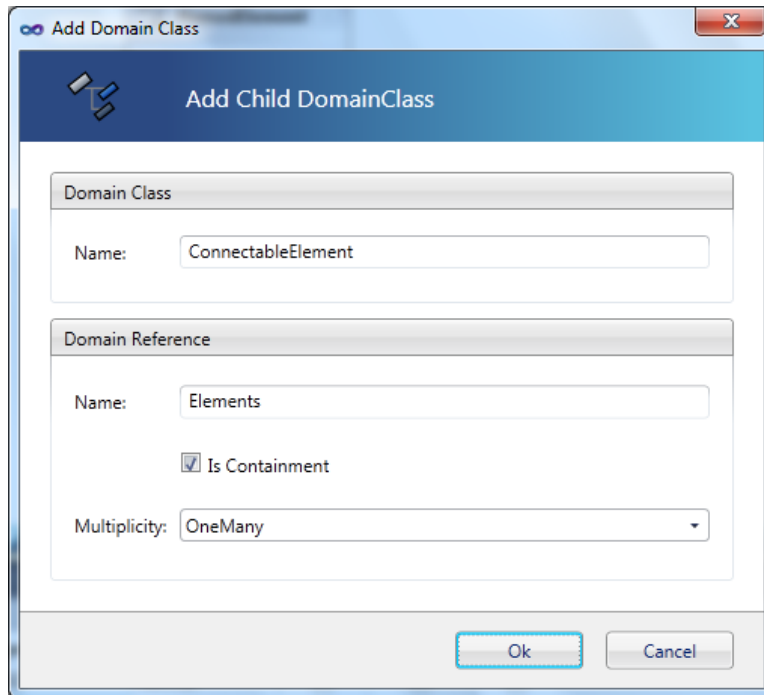


Now we will create the root domain class **PIDiagram** which inherited from **NamedElement**. The domain class is marked as **Root**. The **PIDiagram** contains two properties **RequestedTemperature** and **RoomTemperature**.



From the **PIDiagram** domain class smart commands, press **Add Domain Class** to add a child class called **ConnectableElement**.

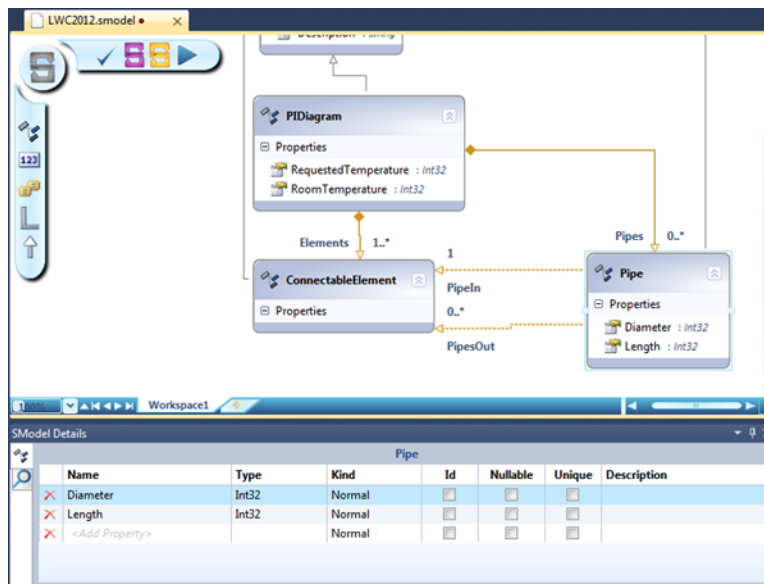




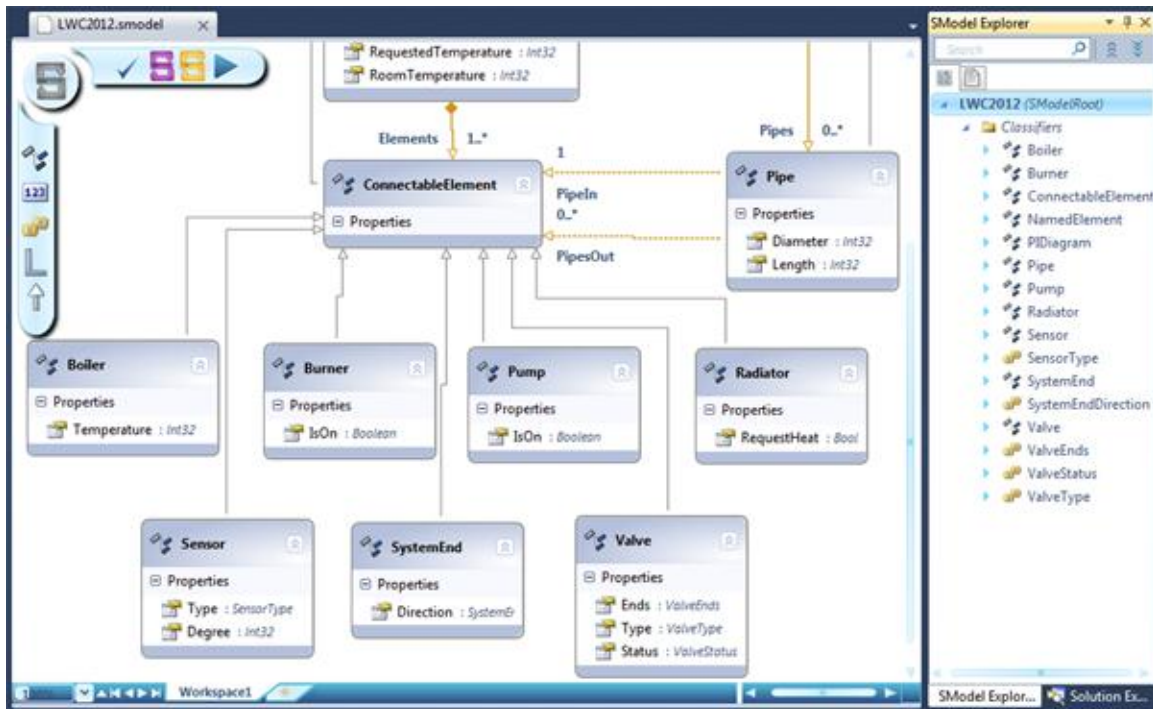
The 'Add Domain Class' dialog box is shown. It has a title bar with a close button. The main area is titled 'Add Child DomainClass'. It contains two sections: 'Domain Class' and 'Domain Reference'. In the 'Domain Class' section, the 'Name' field is set to 'ConnectableElement'. In the 'Domain Reference' section, the 'Name' field is set to 'Elements', the 'Is Containment' checkbox is checked, and the 'Multiplicity' dropdown is set to 'OneMany'. At the bottom, there are 'Ok' and 'Cancel' buttons.

This domain class will be the parent for all elements whom connected by pipes. Of course it is also inherited from the **NamedElement** domain class.

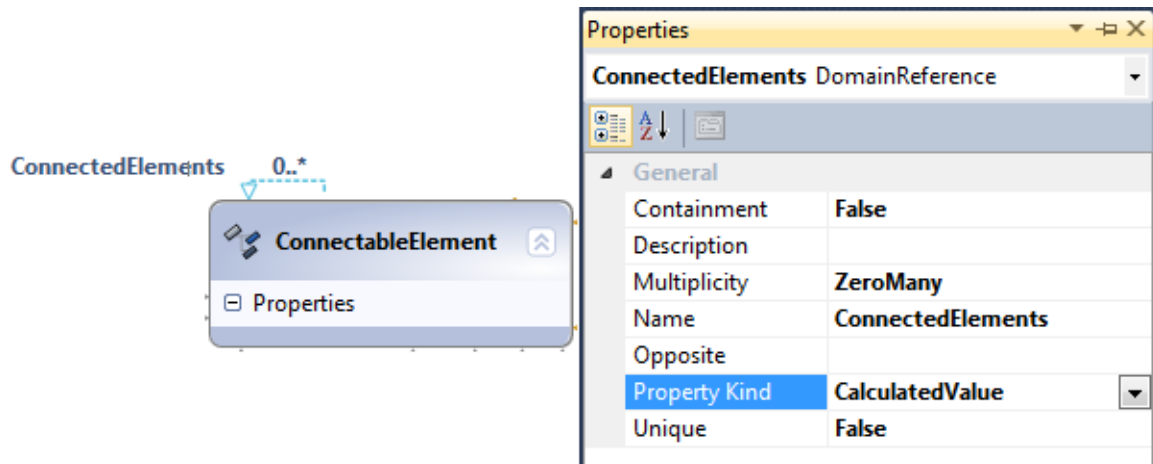
Now we will add the **Pipe** domain class, From the **PIDiagram** domain class smart commands, press **Add Domain Class** to add a child class called **Pipe**. Add **Diameter** and **Length** properties to the **Pipe** domain class. The **Pipe** has two reference relationships with **ConnectableElement**. The first is **PipeIn** with **One** for Multiplicity; the second is **PipesOut** with **ZeroMany** for Multiplicity, where the pipe has one input and multiple outputs.



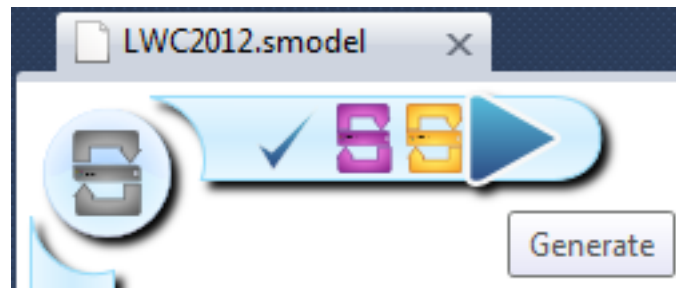
Now we will create our concrete elements which inherited from **ConnectableElement**. Starting from the **Boiler** with **Temperature** property, **Burner** with **IsOn** property, **Radiator** with **RequestHeat** property, **SystemEnd** with **Direction** property from type **SystemEndDirection** Enumeration with **Source** and **Exhaust** literals, **Pump** with **IsOn** property, Plus **Sensor** with **Type** and **Degree** properties, and **Valve** with **Ends**, **Type**, and **Status** properties.



The final thing we need to model is a calculated self-reference which will calculate the connected elements of a particular element.



Now we can validate our model and then generate the model code, from the designer toolbar press **Validate**, and then **Generate**.



We need to add some validation rules like preventing the **SystemEnd** from connecting to more or less than one **Pipe**. From the domain class smart command press **Add Validation Rule**. This command automatically generates **SystemEnd** partial class with the skeleton of **GetViolationRules** method.

```
namespace LWC2012.Model
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using Modelingsoft.Sculpture.SModel.Common;

    internal partial class SystemEnd
    {
        protected override IEnumerable<ViolationRule> GetViolationRules()
        {
            List<ViolationRule> violationRules = new List<ViolationRule>(base.GetViolationRules());

            //violationRules.Add(new ViolationRule(this, "Error Message", ViolationType.Error));

            return violationRules;
        }
    }
}
```

First we will query about all connected Pipes, whether **PipeIn** or **PipesOut**. If the count of pipes not equal one, then add new violation rules indicates that this **SystemEnd** must has 1 connection point.

```
protected override IEnumerable<ViolationRule> GetViolationRules()
{
    List<ViolationRule> violationRules = new List<ViolationRule>(base.GetViolationRules());

    var connectedPipes = this.DomainModel.Locate<IPipe>().
        Where(P => P.PipeIn == this || P.PipesOut.Contains(this));

    if (1 != connectedPipes.Count())
    {
        violationRules.Add(new ViolationRule(this,
            string.Format("System End '{0}' must has 1 connection point.", this.Name),
            ViolationType.Error));
    }

    return violationRules;
}
```

Finally we need to add the implementation of the calculated reference **ConnectedElements**. In the custom code folder, add new **ConnectableElement** partial class. Override **OnGetConnectedElements** Method. Let's add a snippet of code that grab all connected pipes and add the opposite side to a list. Now we can use the property **ConnectedElements** to browse all connected elements through pipes.

```
internal partial class ConnectableElement
{
    protected override ReadOnlyDomainCollection<IConnectableElement> OnGetConnectedElements()
    {
        DomainCollection<IConnectableElement> connectedElements = new DomainCollection<IConnectableElement>();

        var connectedPipes = this.DomainModel.Locate<IPipe>().
            .Where(P => P.PipeIn == this || P.PipesOut.Contains(this));

        foreach (var pipe in connectedPipes)
        {
            this.AddElement(pipe.PipeIn, connectedElements);
            foreach (var element in pipe.PipesOut)
            {
                this.AddElement(element, connectedElements);
            }
        }

        return new ReadOnlyDomainCollection<IConnectableElement>(connectedElements);
    }

    private void AddElement(IConnectableElement element, DomainCollection<IConnectableElement> connectedElements)
    {
        if (element != this && false == connectedElements.Contains(element))
        {
            connectedElements.Add(element);
        }
    }
}
```

TEST YOUR MODEL

At this moment, we are having a complete model, let's try to consume it from a console application.

Add new console project, then add the required assemblies. Now let's create a new instance of our model called **P&I Network**, and retrieve the model factory.

```
namespace LWC2012.Console
{
    using System;
    using System.Linq;
    using LWC2012.Model;

    class Program
    {
        static void Main(string[] args)
        {
            var diagram = LWC2012DomainModel.CreateNewInstance().Root as IPIDiagram;
            diagram.Name = "P&I Network";
            var factory = diagram.DomainModel.DomainFactory;
        }
    }
}
```

Create a new **SystemEnd** called **Cold Water** then add it to the diagram's elements.

```
var diagram = LWC2012DomainModel.CreateNewInstance().Root as IPIDiagram;
diagram.Name = "P&I Network";
var factory = diagram.DomainModel.DomainFactory;

var systemEnd = factory.Create<ISystemEnd>();
systemEnd.Name = "Cold Water";
diagram.Elements.Add(systemEnd);
```

By the same way create a new **Boiler** and a new **Pipe**. Add the boiler to the **Pipesout** side of the pipe. Let the **PipeIn** side empty and try to print any validation errors of our model.

```

static void Main(string[] args)
{
    var diagram = LWC2012DomainModel.CreateNewInstance().Root as IPIDiagram;
    diagram.Name = "P&I Network";
    var factory = diagram.DomainModel.DomainFactory;

    var systemEnd = factory.Create<ISystemEnd>();
    systemEnd.Name = "Cold Water";
    diagram.Elements.Add(systemEnd);

    var boiler = factory.Create<IBoiler>();
    boiler.Name = "Boiler1";
    diagram.Elements.Add(boiler);

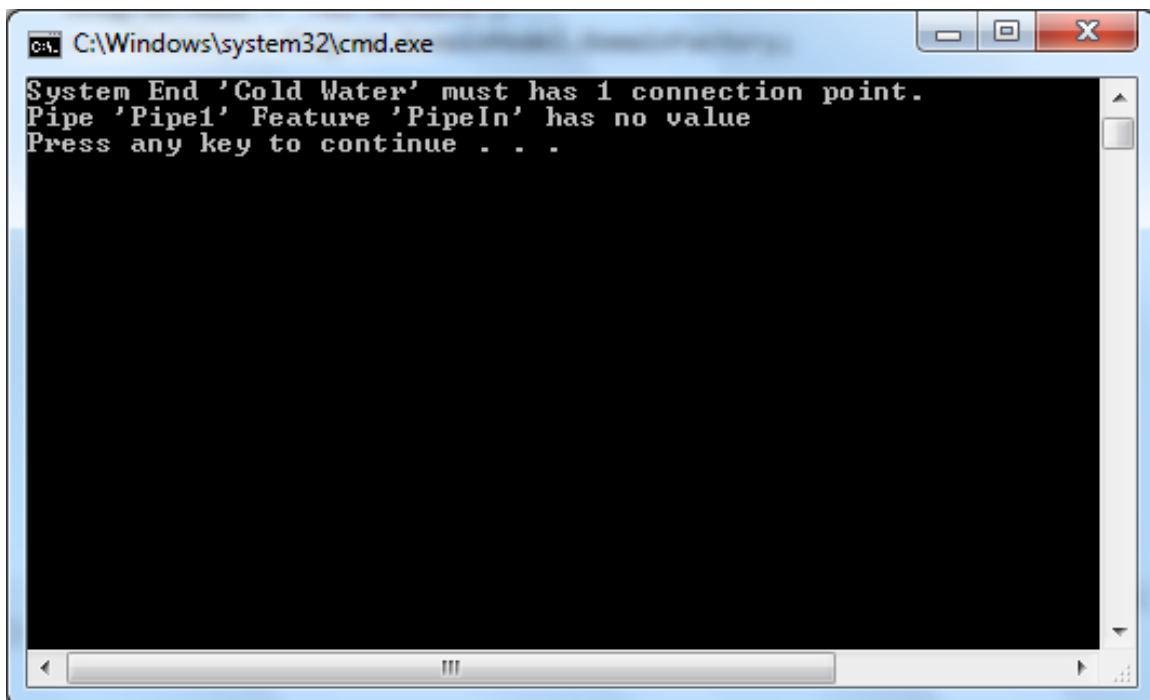
    var pipe = factory.Create<IPipe>();
    pipe.Name = "Pipe1";
    diagram.Pipes.Add(pipe);

    pipe.PipesOut.Add(boiler);

    diagram.Validate().ToList().ForEach(V => Console.WriteLine(V.Message));
}

```

Run the console to see two validation errors, the first indicates that the **Systemend** does not have any connected pipe. The second error indicates that the **PipeIn** has no value.

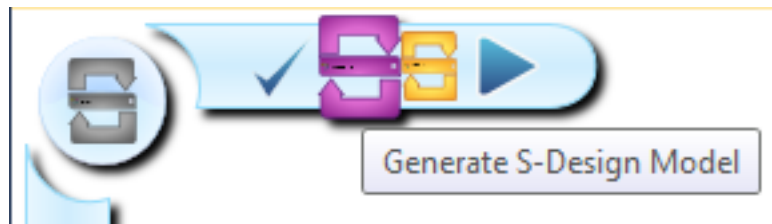


Now set the 'System End' as the 'pipe in', and run the console again to ensure that the model is free of errors.

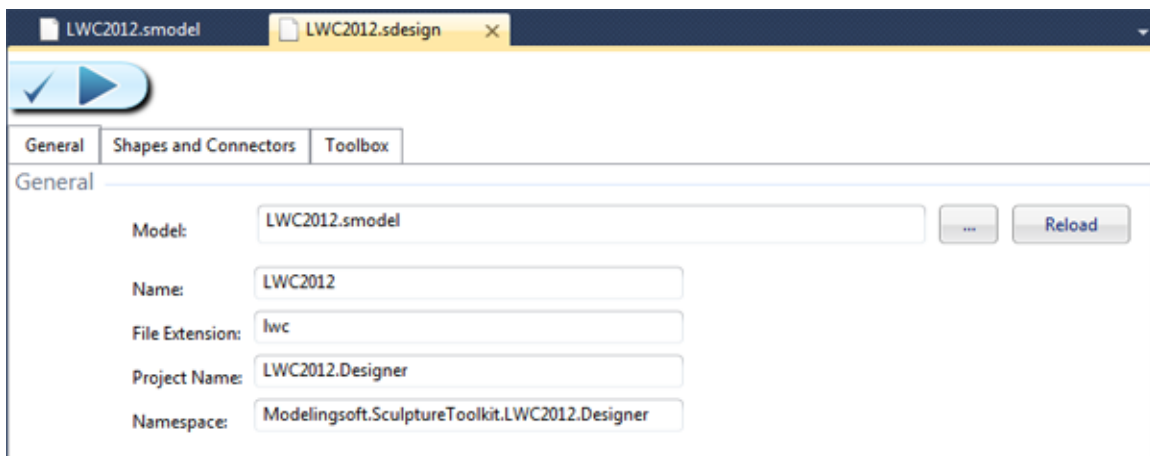
CREATE YOUR DESIGNER

Of course this way of manipulating model is not suitable for domain experts, so let's define the **PINetwork** in such a way that it is intuitive to the domain experts. We will use S-Design to create the **PINetwork** through diagrams.

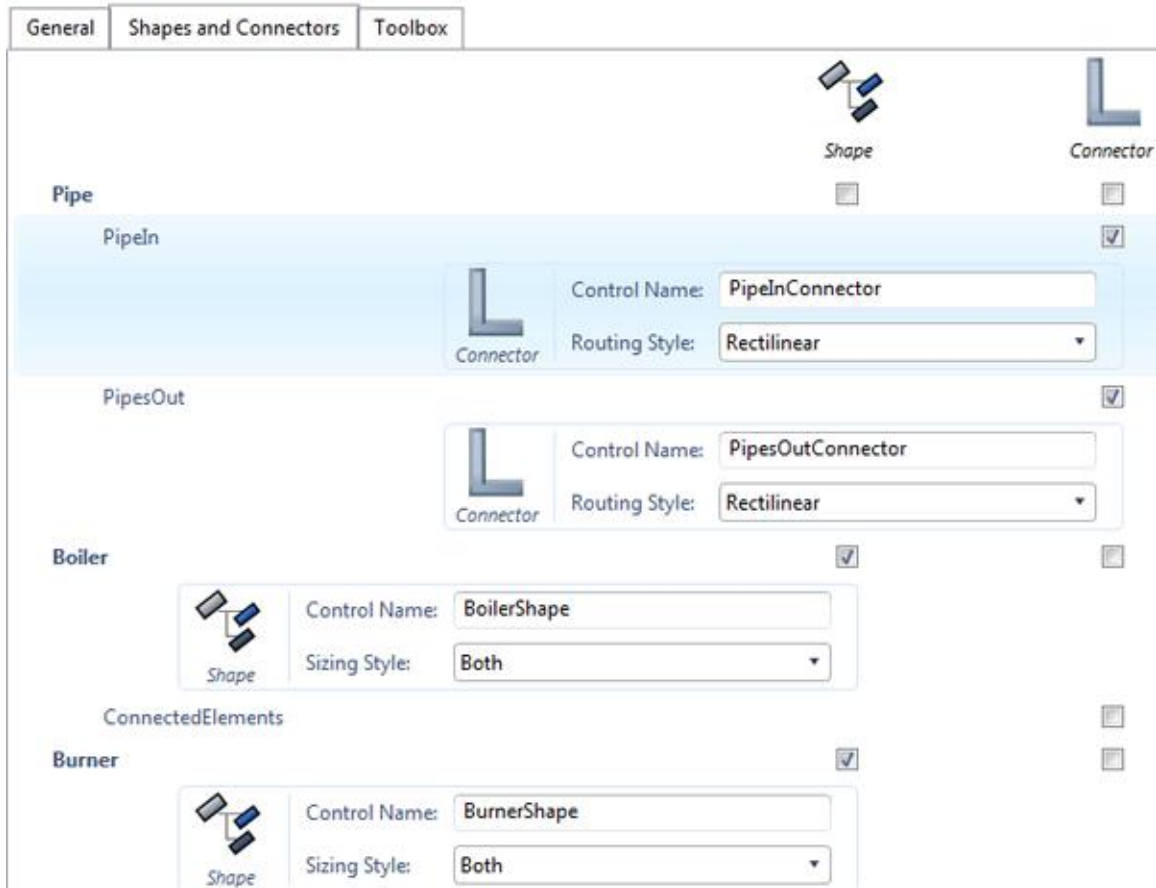
From the S-Model toolbar press **Generate S-Design Model**.



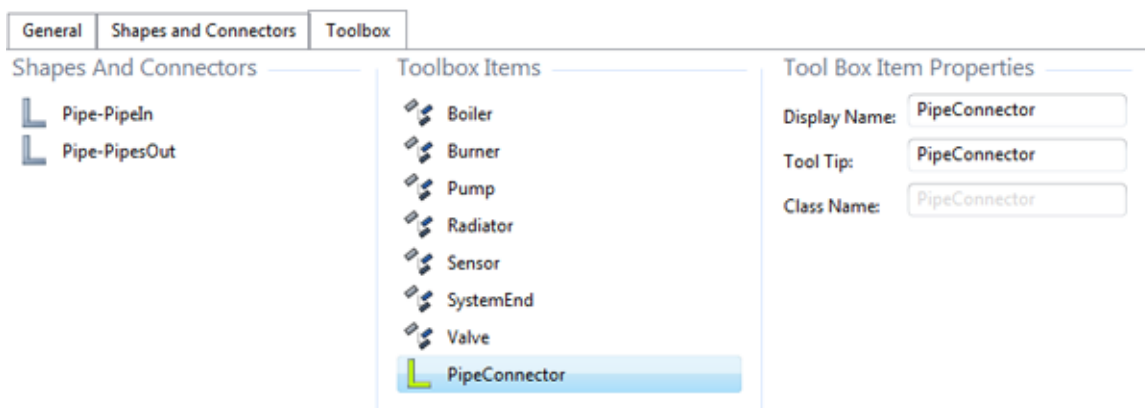
S-Design editor enables you to visually configure your diagram elements. For our particular example, Change the file extension and the project namespace.



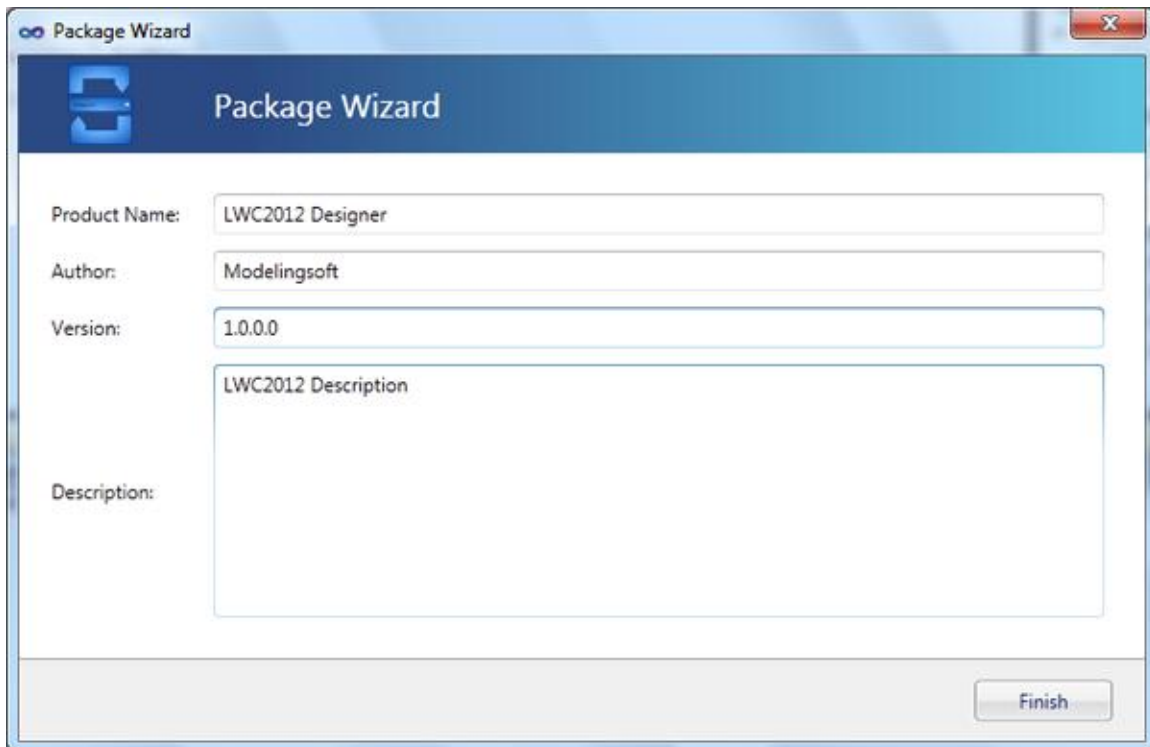
From the **Shapes and Connectors** tab you can choose your diagram shapes and connectors. Choose **Pipe**, **Boiler**, **Burner**, **Pump**, **Radiator**, **Sensor**, **System End**, and **Valve** as shapes. Choose **PipeIn** and **PipesOut** as connectors.



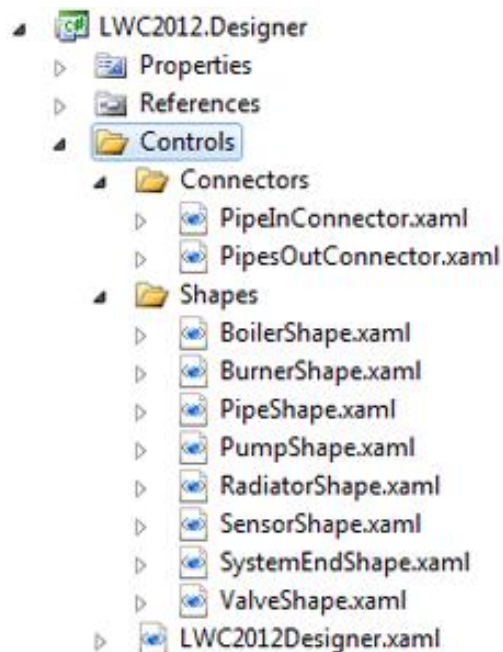
From the **Toolbox** Tab you can determine the diagram's Toolbox items. Choose all items except the pipes related items, where we will create a custom connector which will allow us to automatically create a pipe with its relationships when the user connect any two diagram elements.



The model is ready, validate the model and generate your diagram designer. Fill the Visual Studio Package Wizard.

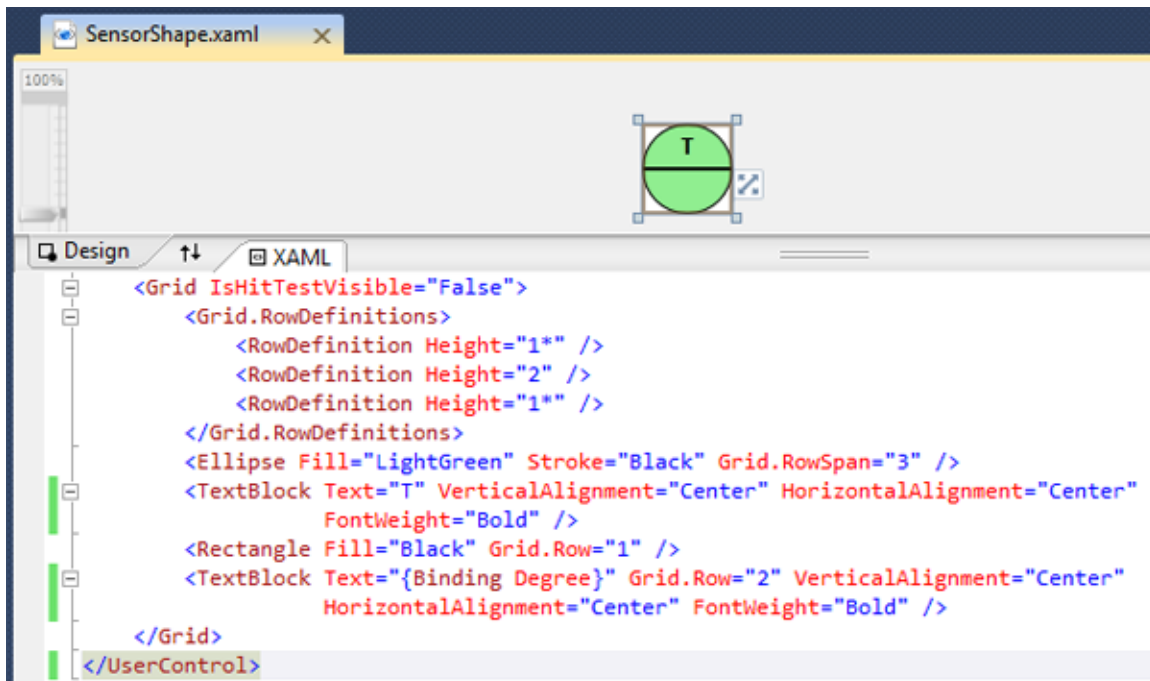


In the generated project 'LWC2012.Designer', you will find the 'Controls' folder which contains all diagram elements in pure WPF (Windows Presentation Foundation).these controls can be edited easily in XAML code.



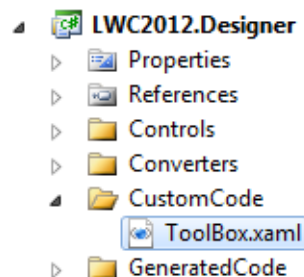
Let's show you how we could author the **SensorShape** control. In this control we will create a grid with 3 rows. In this grid we will add an ellipse that span over all rows. Then

add a text block for example with the letter **T**. And finally add another text block that binds it's text to the sensor **Degree**. By this simple binding statement, we could bind any graphical element property to properties in our model. For example as we will see later, we could bind the visibility of any control (or sometimes called widget) to a Boolean property in our model.



We already prepared some controls that fit in domain expert view point, let's remove our controls folder and add the pre created one.

In addition, let's overwrite the toolbox control with meaningful toolbox icons, first we need to drag it from the generated code folder to the custom code, and then overwrite its contents.



Now we need to implement our custom connector. Add new **PipeConnector** partial class. First override **CanAcceptSource** method to ensure that the source element is **ConnectableElement**. Second override **CanAcceptSourceAndTarget** method to ensure

that the target element also is **ConnectableElement**. Finally override **Connect** Method, in this method create a new pipe and give it a proper name by count all the pipes in the model. Assign the **PipeIn** to the source and add the target to the **PipesOut**.

```
public partial class PipeConnector
{
    public override bool CanAcceptSource(IDomainObject source)
    {
        return source is IConnectableElement;
    }

    public override bool CanAcceptSourceAndTarget(IDomainObject source, IDomainObject target)
    {
        return target is IConnectableElement;
    }

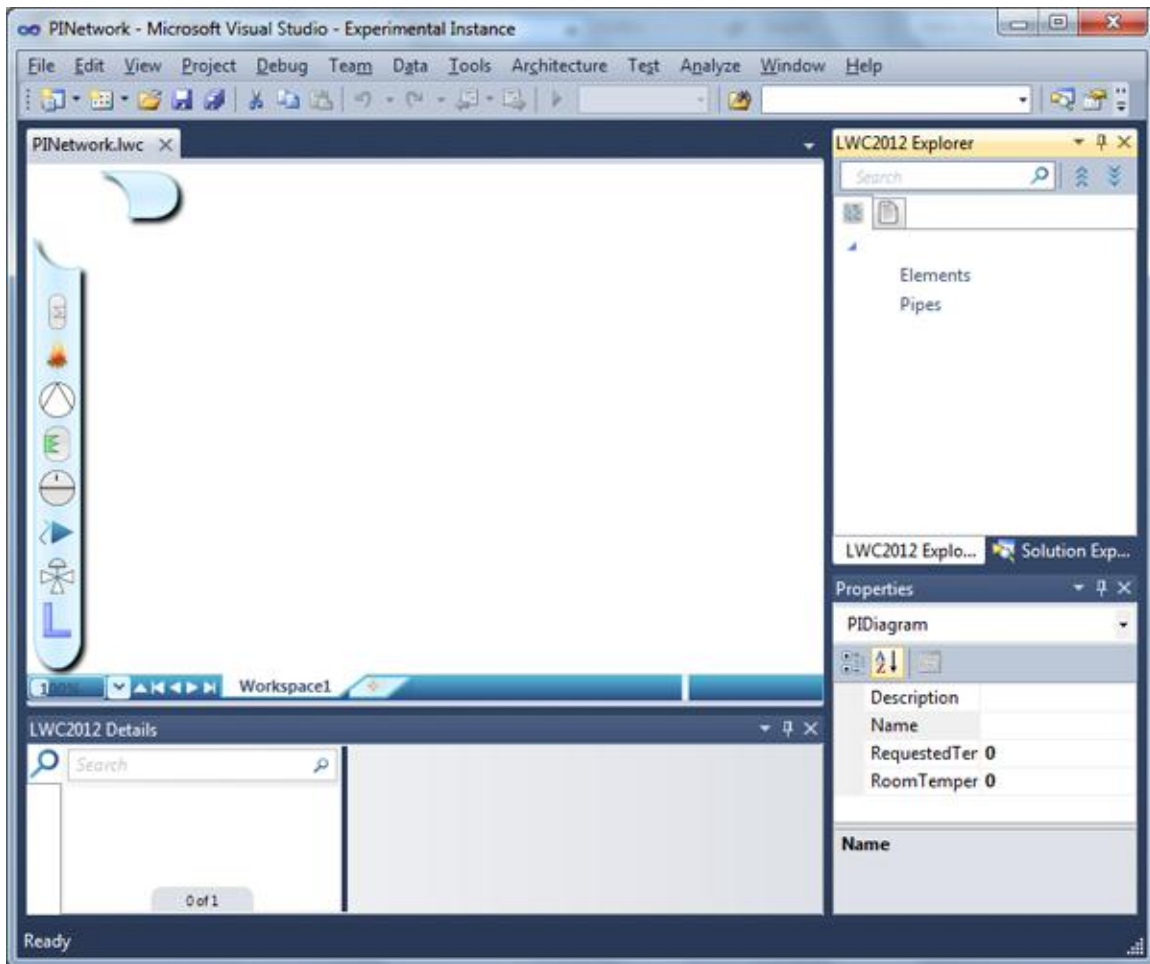
    public override IDomainObject Connect(IDomainObject source, IDomainObject target)
    {
        var pipe = source.DomainModel.DomainFactory.Create<IPipe>();
        pipe.Name = "Pipe" + source.DomainModel.Locate<IPipe>().Count();
        ((IPIDiagram)source.DomainModel.Root).Pipes.Add(pipe);

        pipe.PipeIn = source as IConnectableElement;
        pipe.PipesOut.Add((IConnectableElement)target);

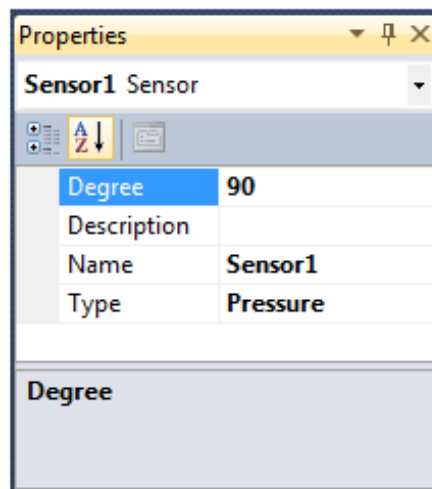
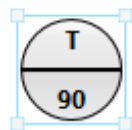
        return pipe;
    }
}
```

Let's build and run to see our designer in action. The designer will run in a new instance of Visual Studio called **Experimental Instance**. Create a new blank solution, then add new file with **LWC** extension. This file will automatically open with our new diagram designer.

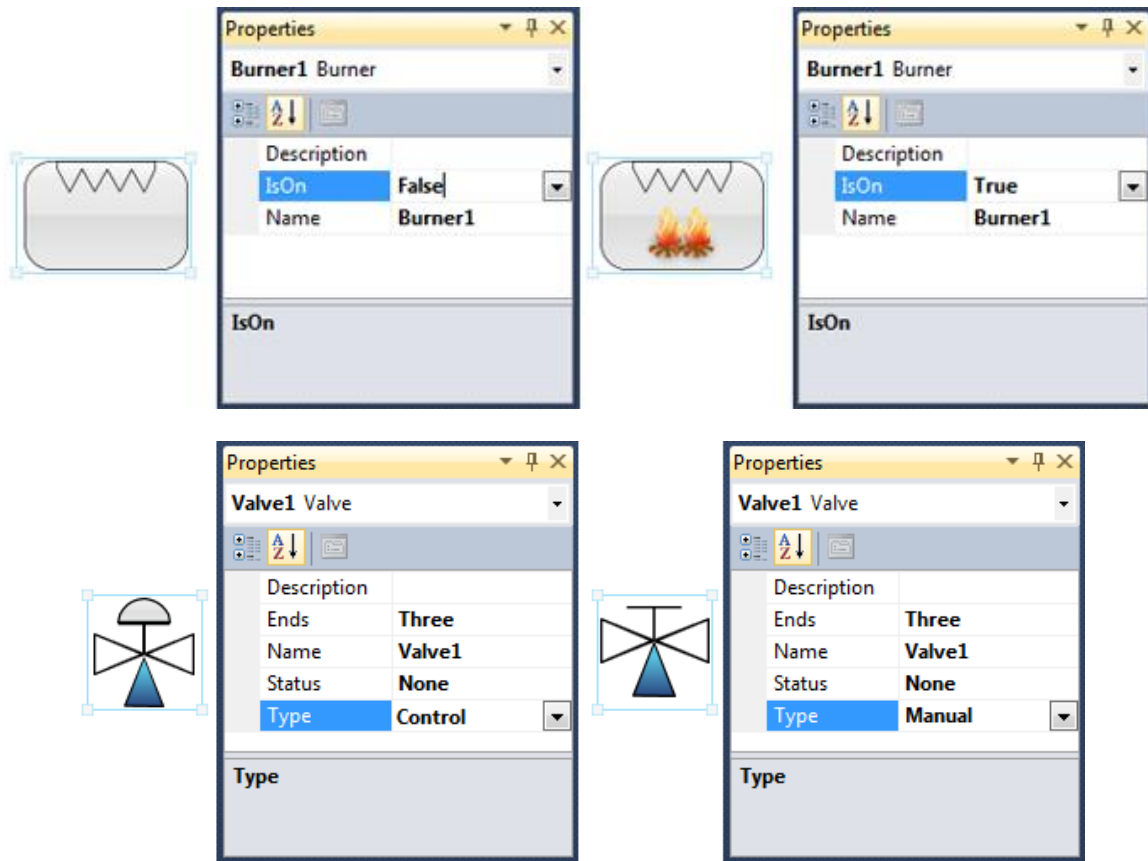
Our diagram designer is composed from diagram surface with multiple workspaces, Toolbox, Empty Toolbar, Model Explorer, and details window.



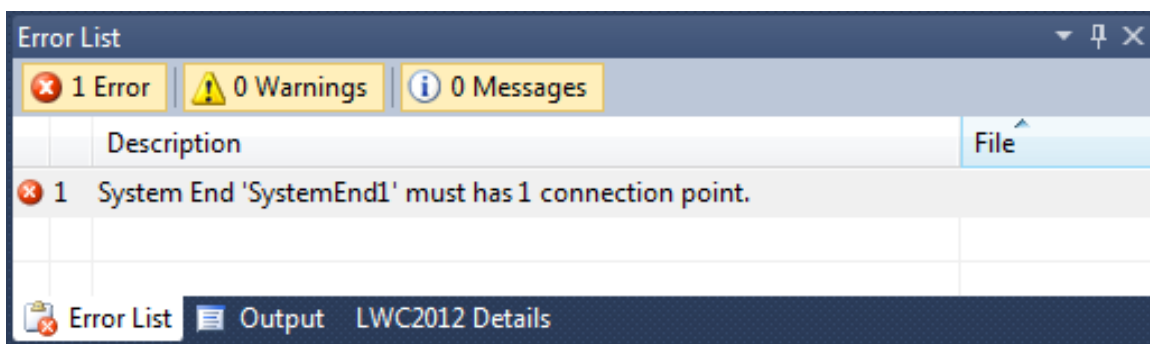
Let's set the diagram name from the properties window, create some elements by dragging it from the toolbox to the designer surface. From the properties window, change the **Sensor** degree to see it automatically reflected in its shape.



The same behavior occurred when we turn the **Burner** on or off, or when we change the **Valve** type or status.



Drag new **SystemEnd** without connecting it to any other element and validate the model to see your custom validation rule.



Now try to connect any two elements using the **Pipe Connector**. You will notice that new pipe is added to the model, but not presented in the diagram surface, where we need to write some code to add the newly created **Pipe** into the current workspace. Manually drag the pipe from the model explorer into the surface to see the connections.

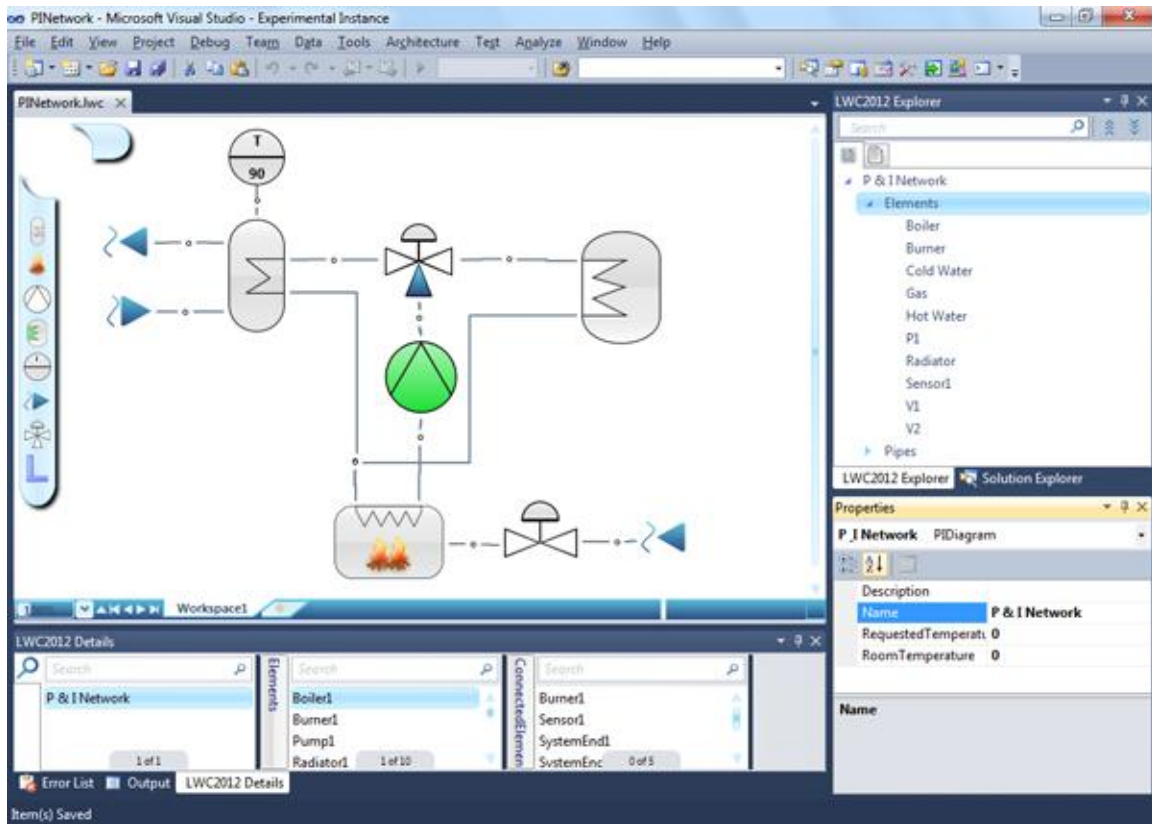


The other thing we need to do is adding some behavior to our model, for example when the **Burner** turned on, all connected **Pumps** must be turned on automatically. Let's return to the designer project and add a snippet of code to add the pipe into the current workspace on the correct position. Second let's return to the model project and create a **Burner** partial class, we will override **OnIsOnChanged** method, in this method we will iterate over all connected Pumps using our calculated property and change its status according to our new status.

```
namespace LWC2012.Model
{
    using System.Linq;

    internal partial class Burner
    {
        protected override void OnIsOnChanged()
        {
            base.OnIsOnChanged();
            foreach (var pump in this.ConnectedElements.OfType<IPump>())
            {
                pump.IsOn = this.IsOn;
            }
        }
    }
}
```

Run the project again; let's start by designing a small **P & I Network** that looks like Language Workbench Challenge reference implementation. You will notice that the pipes are automatically added to the current workspace. And if we try to turn on the burner, the connected pump will be turned on automatically.

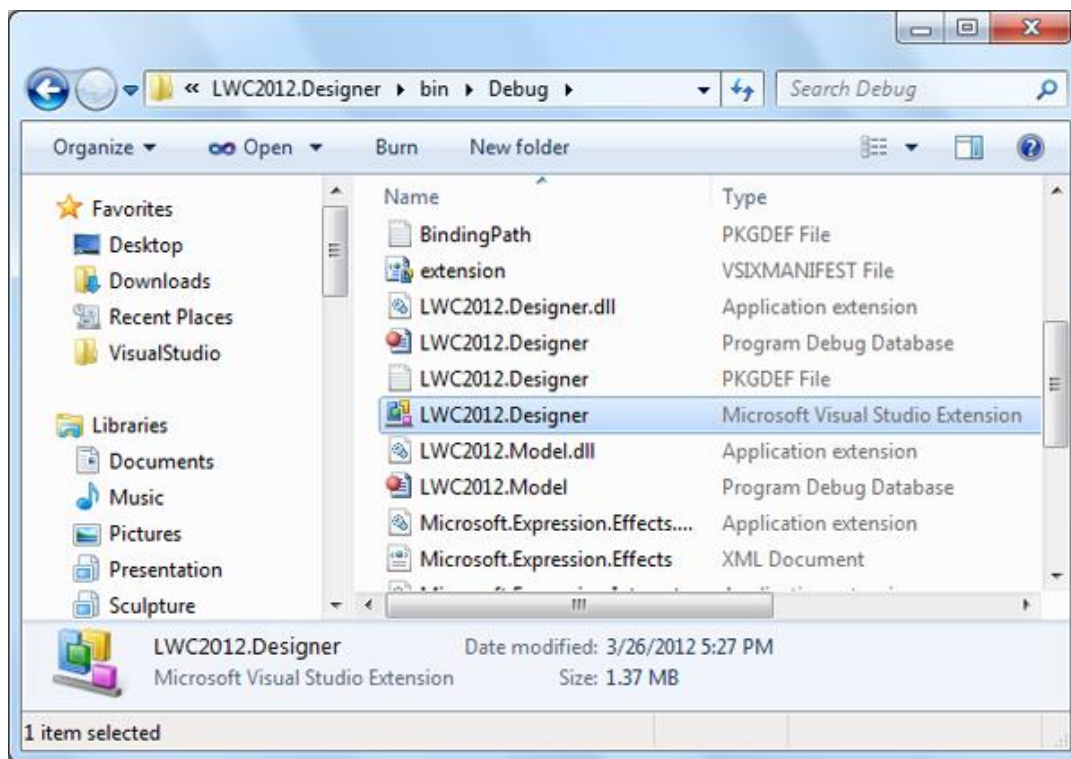


GENERATE ARTIFACTS FROM YOUR MODEL

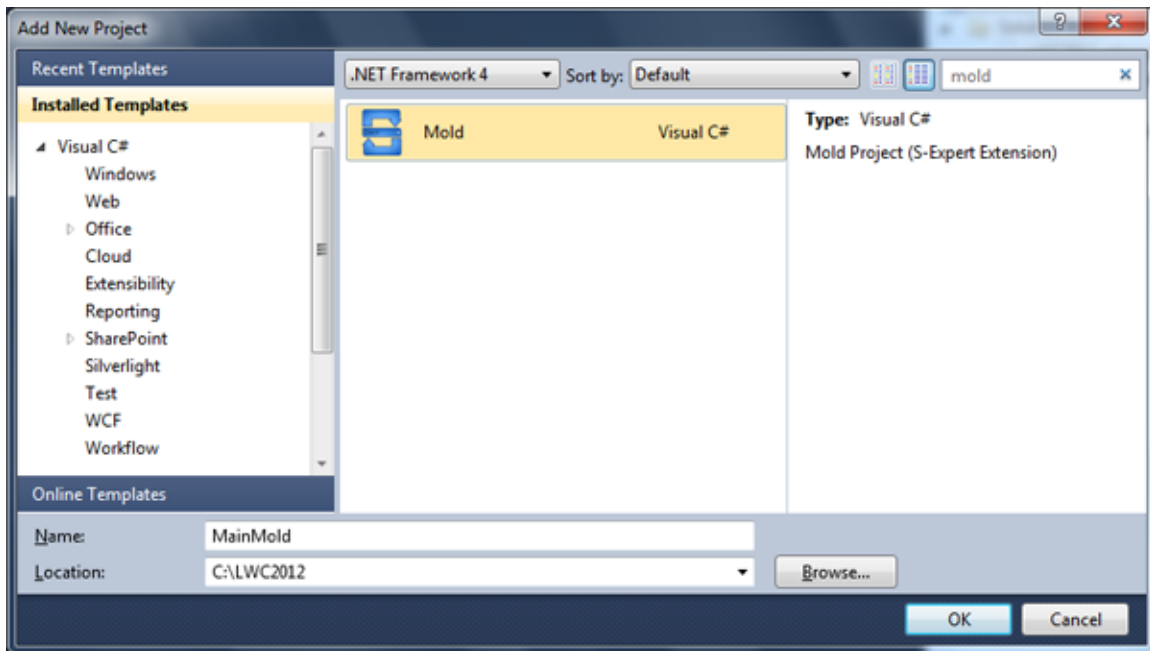
After creating our interactive designer, let's try to generate some artifacts from this model. With Sculpture Toolkit, you can use S-Generate to create templates that can generate any type of artifacts based on your model.

Sculpture Toolkit is a modular application and accepts the templates as plugins which called **Mold**. Mold is the primary plugin for any model created by Sculpture Toolkit. Through the Mold you can create loosely integrated plugins, which can extend your model elements, editor behavior, or even extending your generator.

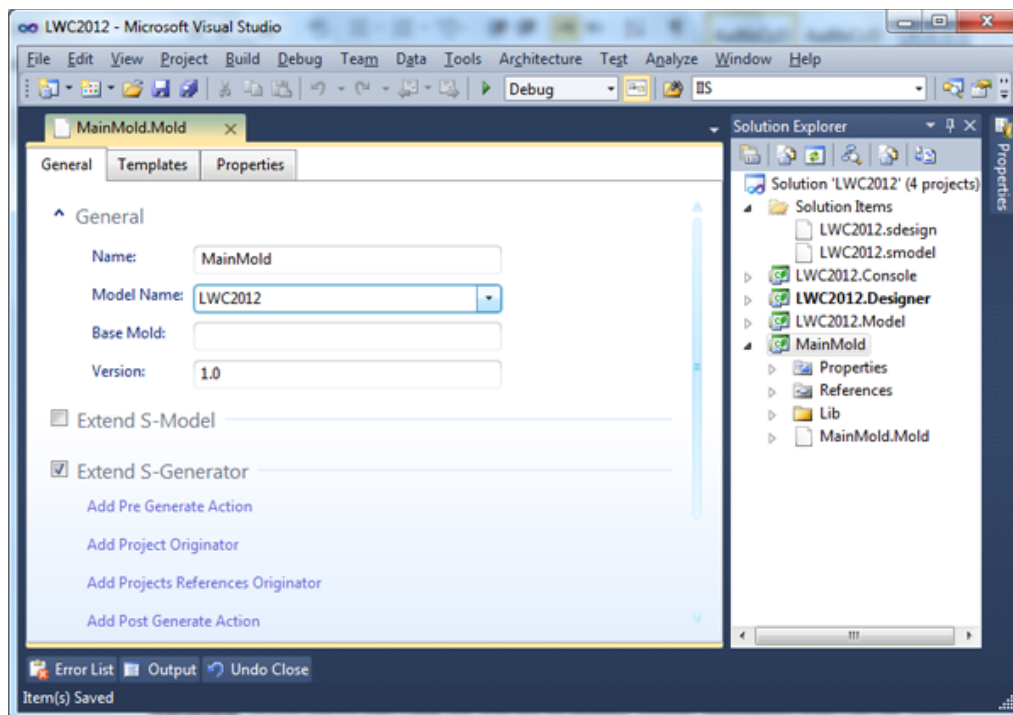
First we need to deploy our model designer, so we can create our first Mold. From the **Debug** folder run the designer **VSIX** file to deploy your designer into the Visual Studio.



Now add new Mold project to your solution called **MainMold**.

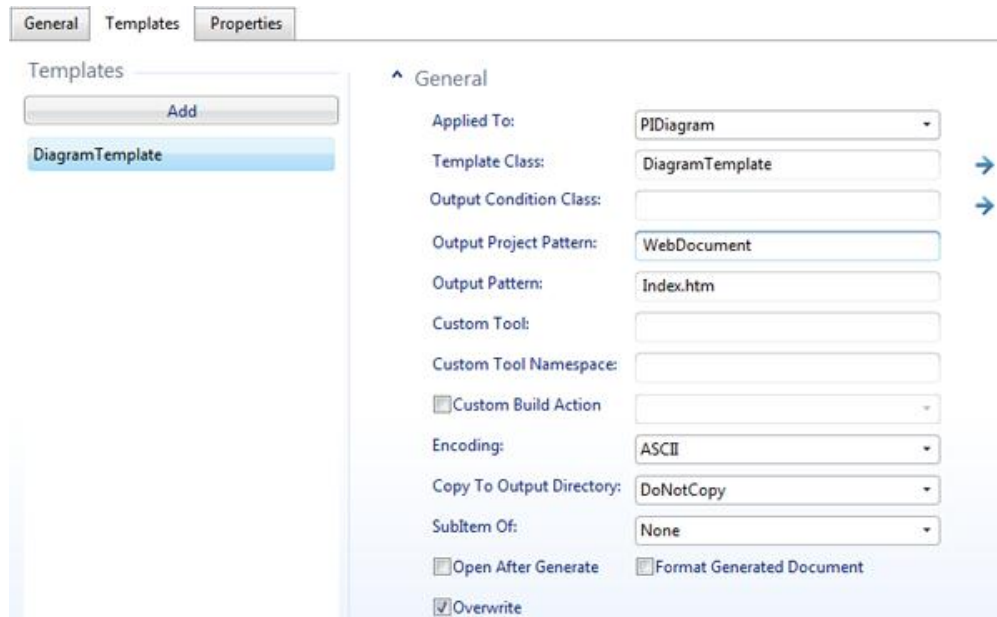


From the Mold Editor, Choose **LWC2012** for the model name, and mark **Extend S-Generate**.



In our particular case we will try to author simple templates that can be used to generate web documentation for our **PINetwork**.

From the **Templates** tab, add new template, choose the root element **PIDiagram** in the **Applied To** field. In the template class type **DiagramTemplate**. In the **Output Project Pattern** type **WebDocument**, and in the **Output Pattern** type **Index.htm** where the generated text will be saved in file called **Index.htm** in a project called **WebDocument**.



Press create template to create the skeleton of your template. Let's add a simple HTML content. In this content let's add a linked list with all elements in the model. Each element refers to another HTML page inside the 'Elements' folder.

```

MainMold.Mold  DiagramTemplate.tt
<#@ template language="C#" inherits="Modelingsoft.Sculpture.SGenerate.Common.SGenerateTemplate" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="Modelingsoft.Sculpture.SModel.Common" #>
<#@ import namespace="LWC2012.Model" #>
<#
    IPIDiagram diagram = this.DomainObject as IPIDiagram;
    #>
<html>
  <head>
    <title><#= diagram.Name #></title>
  </head>
  <body>
    <h1><#= diagram.Name #></h1>
    <h2><i>Elements:</i></h2>
    <ul>
      <#
        foreach (var element in diagram.Elements)
        {
          <li><a href="Elements/<#= element.Name #>.htm"><#= element.Name #></a></li>
          <#
        }
      <#
    </ul>
  </body>
</html>

```

Return to the Mold Editor, and create a template for all our model elements. Starting from the **Boiler** with **BoilerTemplate** with **Output Pattern** inside the **Elements** folder, at this point you will notice that we used the property **Name** of the **Boiler**, where the **Output Pattern** is not a static, you can use any property to name the generated output. Let's add templates for the other elements with the same specification.

Templates		General	
<input type="button" value="Add"/>		Applied To:	Boiler
DiagramTemplate		Template Class:	BoilerTemplate
BoilerTemplate		Output Condition Class:	
BurnerTemplate		Output Project Pattern:	WebDocument
PumpTemplate		Output Pattern:	Elements\{Name}.htm
RadiatorTemplate		Custom Tool:	
SensorTemplate		Custom Tool Namespace:	
SystemEndTemplate		<input type="checkbox"/> Custom Build Action	
ValveTemplate			

Of course creating an individual template for all these elements does not make sense, let's take this chance to illustrate the meta-model capabilities of Sculpture Toolkit. Get a copy of the **DiagramTemplate** called **ConnectableElementTemplate**. In this template we will treat all the passed domain objects as **ConnectableElement**. In this template we will loop over all the properties of the element using its meta-data property called **DomainClass**. It somehow looks like the reflection in the traditional programming languages. For each property we will print the property name and the value of this property in the current element.

```

<#@ import namespace="System.Linq" #>
<#@ import namespace="Modelingsoft.Sculpture.SModel.Common" #>
<#@ import namespace="LWC2012.Model" #>
<#
    IConnectableElement element = this.DomainObject as IConnectableElement;
#>
<html>
  <head>
    <title><#= element.Name #></title>
  </head>
  <body>
    <a href=" ../index.htm">HOME</a>
    <h1><#= element.Name #></h1>
    <h2><i>Properties:</i></h2>
    <ul>
      <#
        foreach (var property in element.DomainClass.Properties)
        {
          <li><#= property.Name #>: <#= element.Get(property).ToString() #></li>
        }
      <#
    </ul>
  </body>
</html>

```

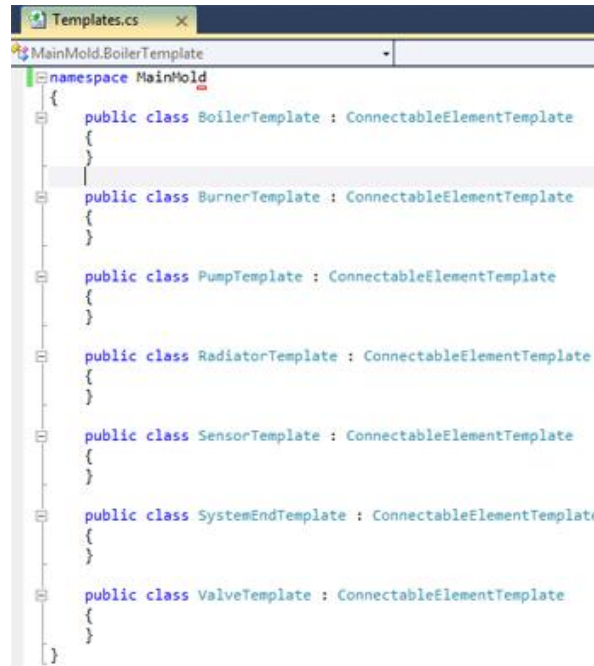
In addition, let's use our calculated property **ConnectedElements** to print a linked list with all other elements that are connected to my element.

```

    <h2><i>Connected Elements:</i></h2>
    <ul>
      <#
        foreach (var connectedElement in element.ConnectedElements)
        {
          <li><a href=" <#= connectedElement.Name #>.htm"><#= connectedElement.Name #></a></li>
        }
      <#
    </ul>
  </body>
</html>

```

After creating this abstracted template, let's add a code file to inherit all elements' templates from it. This methodology promotes authoring reusable templates.



```
Templates.cs
MainMold.BoilerTemplate
namespace MainMold
{
    public class BoilerTemplate : ConnectableElementTemplate
    {
    }

    public class BurnerTemplate : ConnectableElementTemplate
    {
    }

    public class PumpTemplate : ConnectableElementTemplate
    {
    }

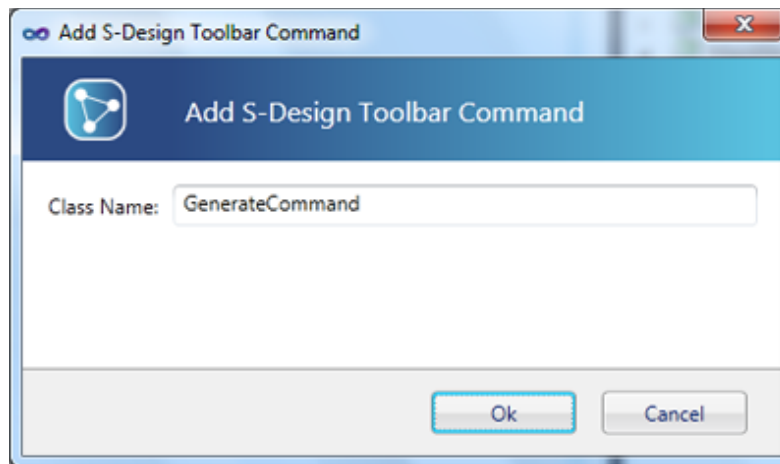
    public class RadiatorTemplate : ConnectableElementTemplate
    {
    }

    public class SensorTemplate : ConnectableElementTemplate
    {
    }

    public class SystemEndTemplate : ConnectableElementTemplate
    {
    }

    public class ValveTemplate : ConnectableElementTemplate
    {
    }
}
```

Now we need a command to trigger our generator. From the Mold Editor, check **Extend S-Design**, and press **Add Toolbar Command**. Type **GenerateCommand** for the class name.



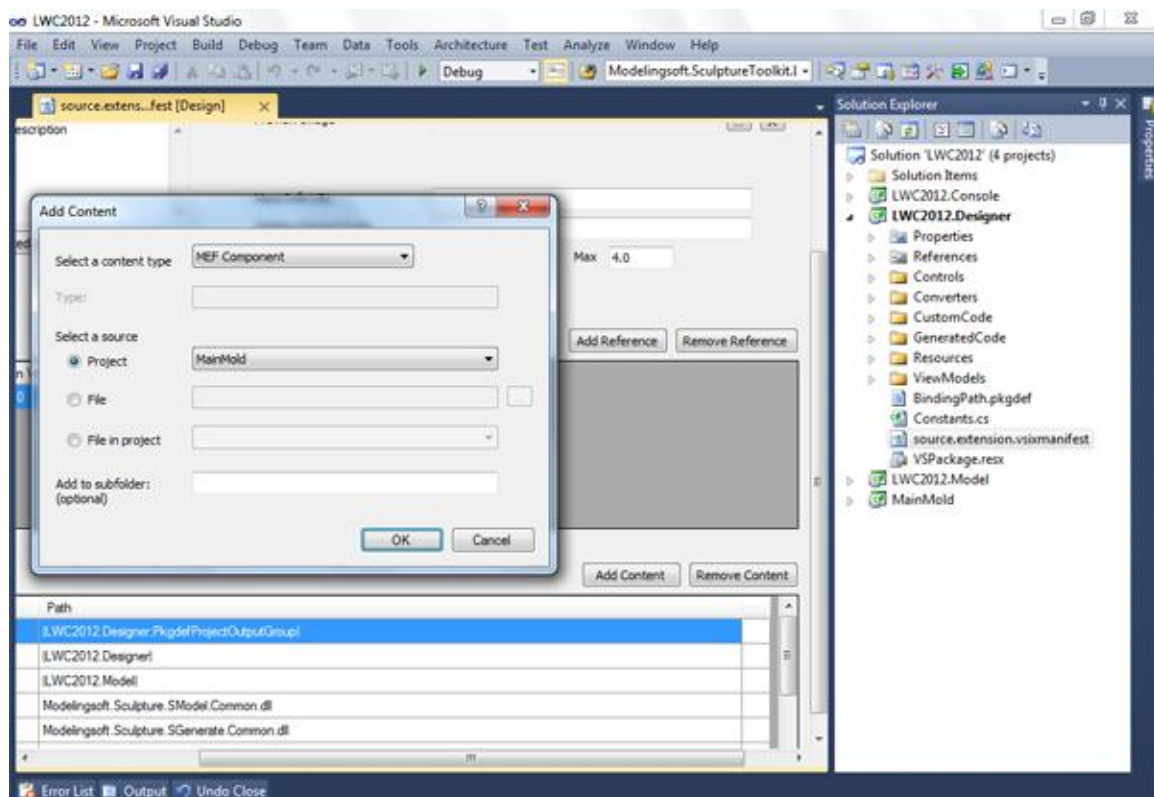
Of course we can create any appropriate control style, for now we will only change the content to the word **Generate Web Document**. And when the command is clicked we will check if the model is valid then we will trigger the generator in Asynchronous way.

```
[MoldComponent(typeof(ISDesignToolBarCommand), MainMold.MoldName, MainMold.ModelName)]
public class GenerateCommand : SDesignToolBarCommand
{
    public override int Order
    {
        get { return 10; }
    }

    public override FrameworkElement GetNewVisualElement()
    {
        // return any control as the follwong example:
        Button button = new Button();
        button.Content = "Generate Web Document";
        button.Click += new RoutedEventHandler(Command_Click);
        return button;
    }

    private void Command_Click(object sender, RoutedEventArgs e)
    {
        if (this.DomainModel.IsValid)
        {
            this.SculptureGenerator.GenerateAsync(this.DomainModel, true, true);
        }
    }
}
```

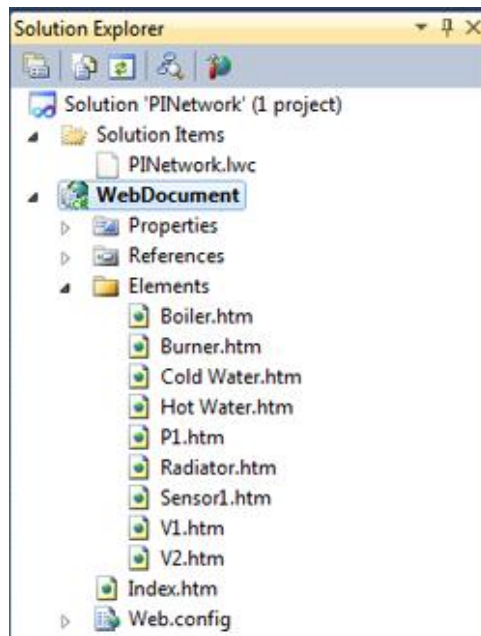
That's it, of course we can deploy the Mold independently, but for the testing purposes we will add its content directly into our designer.



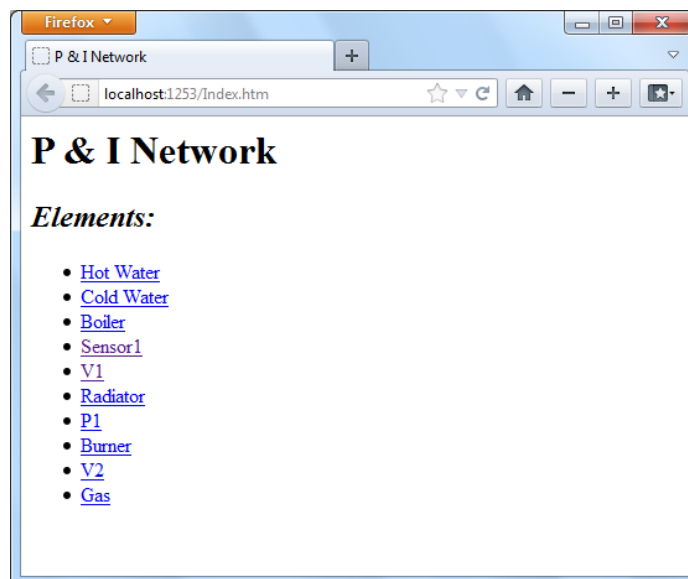
Rebuild and run the solution again. You will notice that the generate button is added to the designer toolbar.



Let's first add an empty web project called 'Web Document', press the generate button to see your generator runs in the output window.



Now let's view the generated project.



As you can see the index page contains all the elements in your model. If you click any of this elements you will be redirected to its page. As you can notice the element properties are generated automatically from its meta-data. Plus a list of all connected elements, which enables you for direct navigation.

