**Requirement**: to determine the triangle type based on 3 sides as inputs.

**Assumption**: application could be extended to offer service to determine other shape’s type, e.g. given a four-side rectangle, it can determine if it is square or rhombus etc.

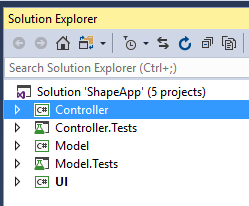
**Overall architecture**: a typical MVC architecture is adopted, see following screenshot.

1. UI layer is command line in this case, but a web user interface can be easily added without changing core service and back-end implementation.
2. Controller layer contains two major components:

a) Service that retrieves request and responds;

b) Factory that responsible for creating triangle types.

3) Model layer stores all shape/types system supports



\* Note : I didn’t take out interfaces into different project/assembly to achieve Stair-way pattern to truly separate implementation and interface , as I see it is really a overkill for this program.

**Design Pattern**: factory pattern, dependency injection, command pattern and null object pattern are used.

Next, each layer is described

**Model** is designed as following illustration: Interface IShape represents all shapes. Class Triangle is a base class that is used for sub class to inherit. Sub class such as Equilateral is the real class we want to return.

Rhombus

Square

Isosceles

Equaliteral

Triangle

Rectangle

IShape

**Controller/Service layer** is illustrated as this:

User request

IShapeService

ShapeFactory

IShapeFactory

ShapeService

UI layer

Service layer

ShapeService does 3 things :

1. receives user input
2. responds with requested information
3. most importantly, decides which shape factory to use. I will describe a bit detail here.

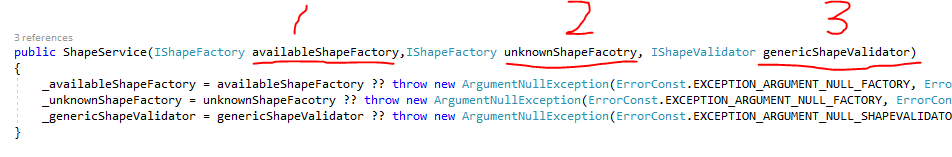
In our case, it is only TriangleFactory, but it can be easily extended to support multiple shape factories. This is achieved by injecting shape factory into the ShapeService. Below is ShapeService constructor:

#1 is the factory that is doing real work (i.e.TriangleFactory). We can easily change this to inject a list of shape factories to offer more functionalities. Inside ShapeService, there is method to find right factory to use given user input.

#2 is a special factory that is used to create null shape object when there is no shape that matches user request or when there is invalid input; this removes null reference check and leaves flexibility to handle those edge cases inside one place UnknownShapeFactory.

#3 is a validation. Validation is considered in two layers. First one is the kind that could be applied to all shapes, e.g. to test all polygon shapes side length must be a positive value, so this is done before we reach a specific shape factory. Second kind is specific for shape. For triangle, we need to check if it has only 3 sides and if it meets inequality theory of triangle. This kind of validation is done inside TriangleFactory.

The reason we inject generic shape validation instead of writing it in the ShapeService is because we can modify or enhance it if we decide to support non-polygon shapes.

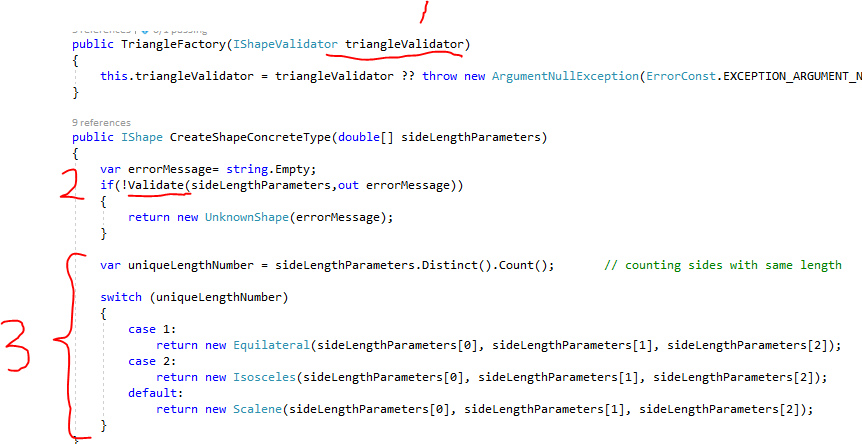


**Shape factory** is responsible for determining which type of shape based on user input. Take TriangleFactory as example, the following screenshot shows TriangleFactory constructor and public method <CreateShapeConcreteType>.

#1 shows that we inject a triangle specific validation

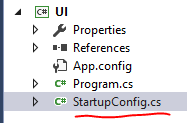
#2 shows <Validate> method delegates validation work to injected validator. And if there is any validation error, we return a UnknownShape object with validation message inside.

#3 shows the algorithm to determine which type of triangle it is and create and return it. We could inject this algorithm as well, but in this case, it is really only one line (counting how many sides have same length) , so I place here.



**UI layer** is simply a command line window. Basic input validation and parsing is done here, so any invalid input is detected and error message is returned immediately to user, without the need to send invalid input to core service. This is especially important for web application.

Notice that Program.cs is the main entrance of program, but I created another class StartupConfig.cs as a place to register dependencies and injection. This is similar to using DI container at start of program in a web application.



Unit test is organized to have same structure as production code. Notice I didn’t create a unit test project for UI. I think other tests are more interesting and enough for demonstration.

