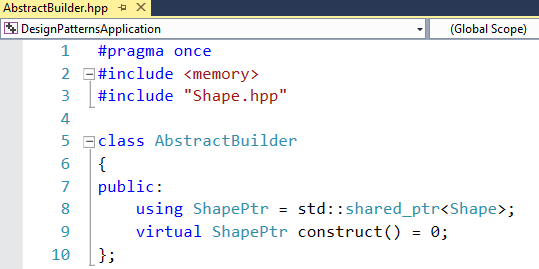
Builder Pattern

The code for creating the sample scene from the previous example is nice, but becomes boring quickly. In this example we want to separate the creation of the scene from the rest of application. We do this by introducing the Builder pattern. Using the Builder pattern, we will provide two different approaches to constructing the scene. The first will be the static builder, taking the existing source for the snowman / house scene and encapsulating it in its own class. The second approach will be an interactive Commandline builder that accepts input at startup describing the scene that will be displayed after construction has finished.

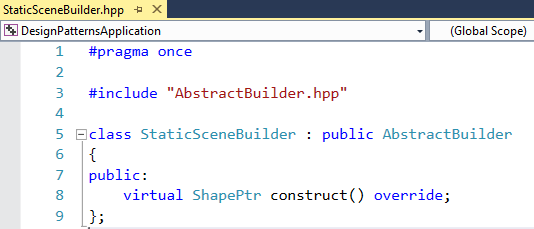
# Add the AbstractBuilder class

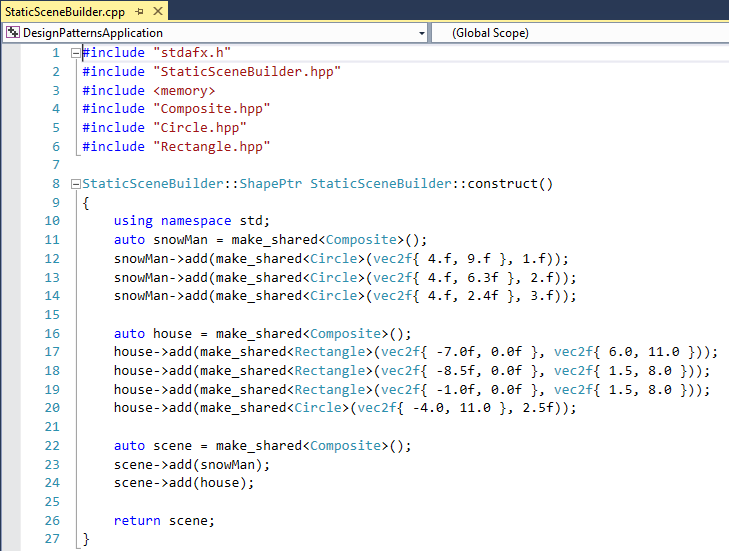
Our first step is to add a new abstract base class -- ‘AbstractBuilder’ -- that encapsulates the construction of a new Shape:



# Refactor the Snowman Scene into StaticSceneBuilder

We will now add a new class – ‘StaticSceneBuilder’ – and add the code from the application startup into the construct method of the StaticSceneBuilder.

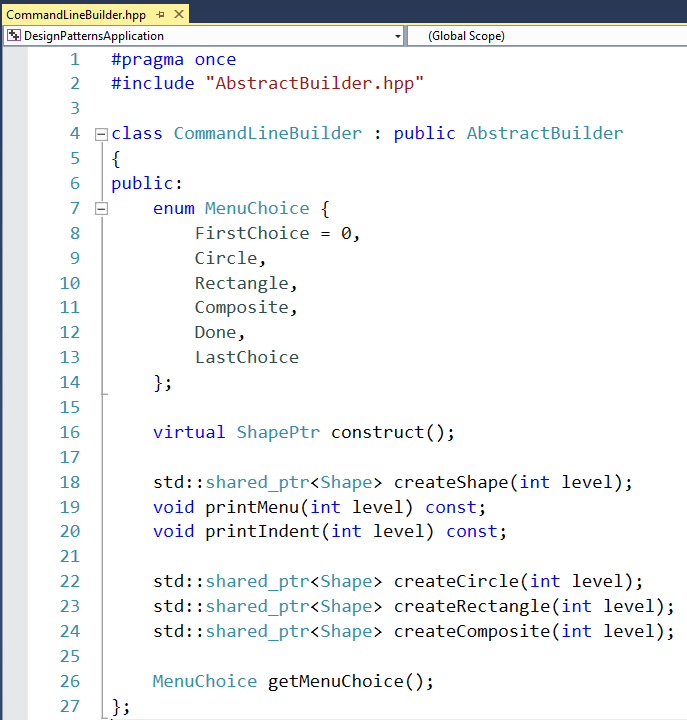




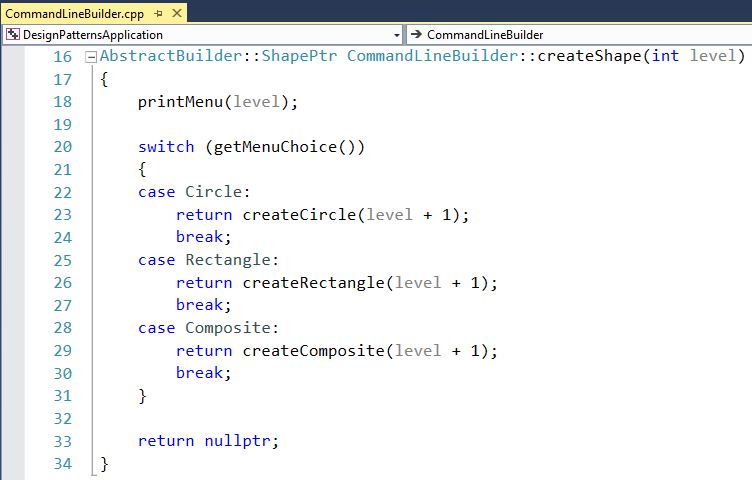
# Commandline Builder

We can create a more dynamic application by asking the user at startup of what the scene should look like. The command line builder runs a basic loop, displaying a menu from which the user can choose shapes from. Each Individual Shape then asks its properties from the user.

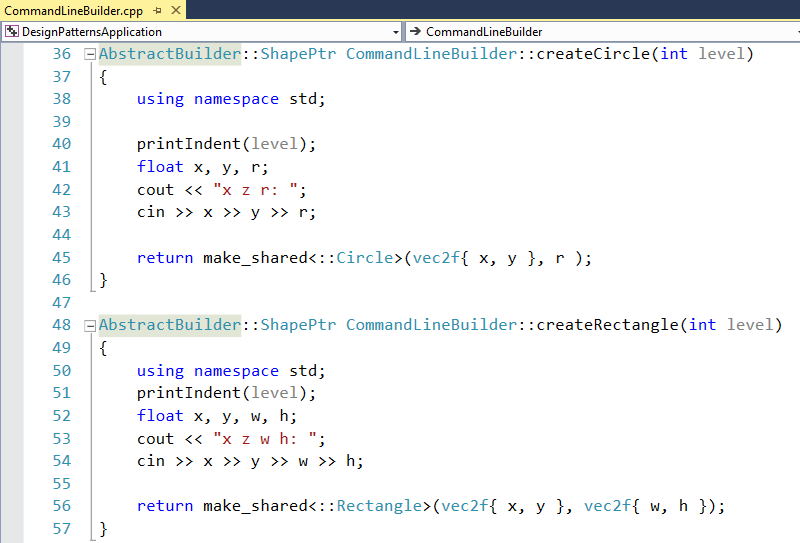
For the commandline builder we add a new class, CommandLineBuilder:



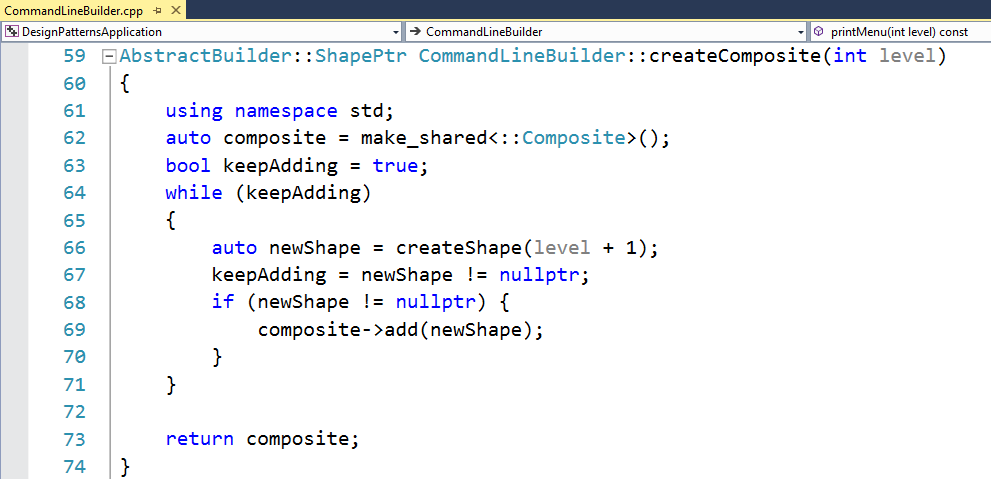
This implementation of an AbstractBuilder adds a few extra methods that perform the user interaction. The construct call forwards to a createShape(int level) method. Since we have Composite shapes, we can construct a recursive scene. The level parameter indicates the depth at which the objects are created. The createShape method runs a loop, presenting a menu and asking the user for an option of which shape to create:



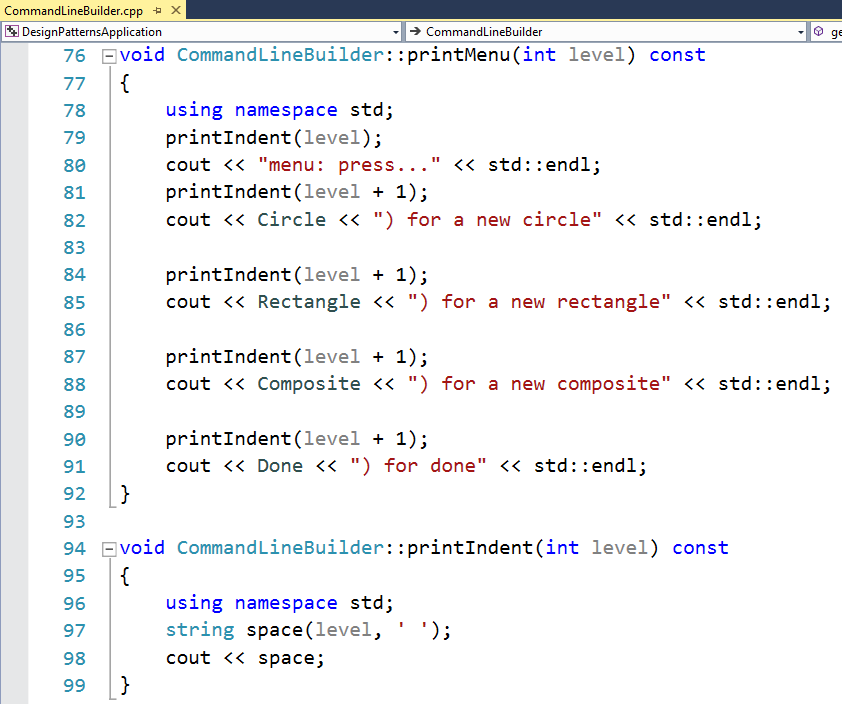
The individual create calls ask for the properties of the shape and return a newly created shape:

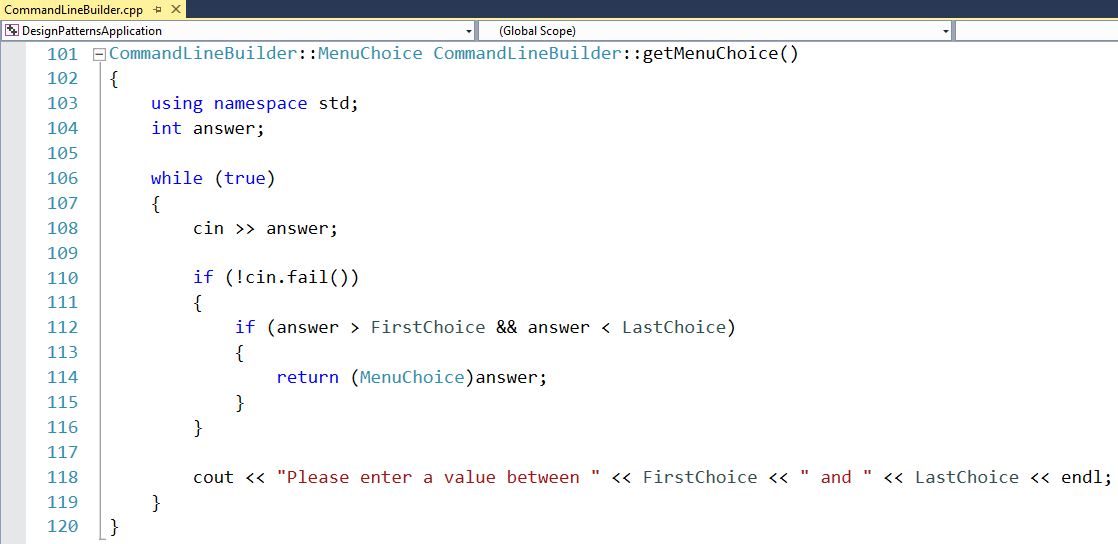


The createComposite method creates a new shape and starts a new loop, calling createShape new a higher level:



For completeness, here are the printMenu / printIndent and getMenuChoice methods:

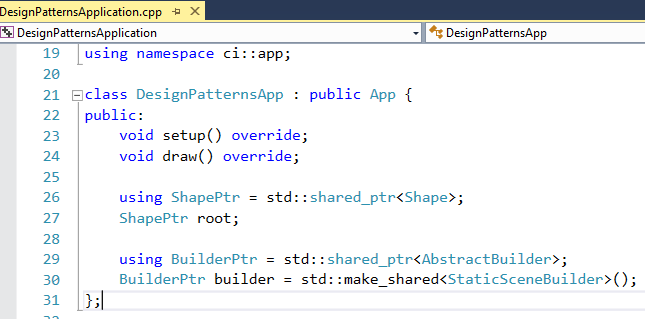


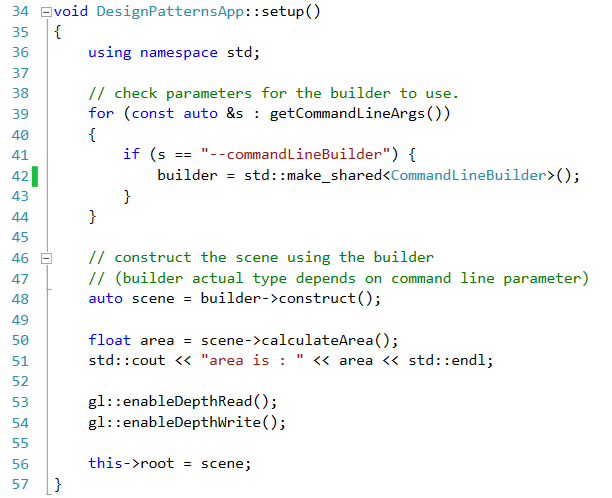


While the CommandLineBuilder is quite a complex class, using it becomes quite simple. In the next step, we will refactor the main application to use command line parameters to choose the type of builder to use.

# Refactor the Main Application

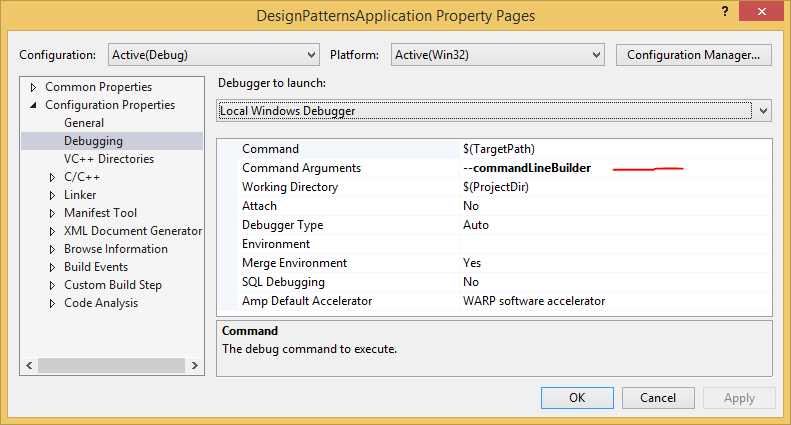
We now change the main application to get the command line arguments and check for a specific string to decide which builder to instantiate. We will add a new member variable – builder – that holds a reference to the builder in use:





# Edit command line arguments at startup

We can set commandline arguments that are passed to the application on startup by opening the properties page and setting the “Command Arguments”:



When starting with this commandline option, the CommandLineBuilder will prompt a menu for creating the scene:

