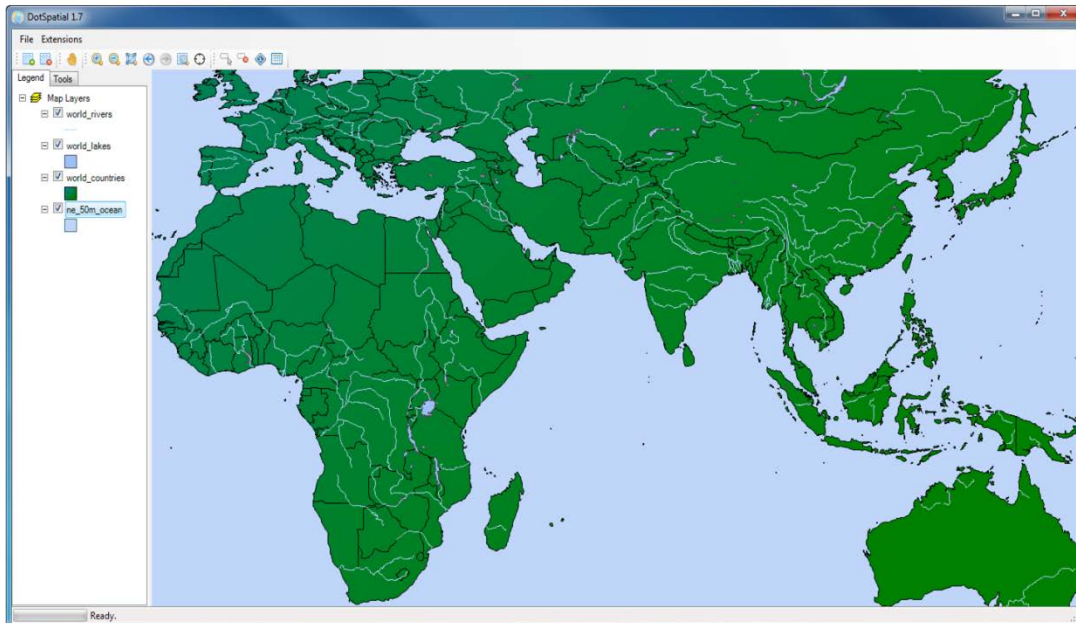


DOT SPATIAL

Developer's Corner



May 2015



Table of Contents

1. Developer's Corner	3
1.1 Geometry Cheat Sheet.....	4
1.1.1 Overlay Operations:	8
1.2 Exercise 1: Assemble a Map Project	10
1.2.1 Step 1: Start a New C# Application	10
1.2.2 Step 2: Add DotSpatial Components.....	11
1.2.3 Step 3: Set up interface for plugin Menu.....	14
1.2.4 Step 4: Adding the Map and other Controls.....	20
1.2.5 Step 5: Add the Legend and Toolbox	22
1.2.6. Step 6: Link It All Together	24
1.3 Exercise 2: Simplify Australia Data Layers.....	26
1.3.1 Step 1: Download Data	26
1.3.2 Step 2: Calculate Polygon Areas.....	27
1.3.3. Step 3: Compute Centroids	31
1.3.4 Step 4: Sub-sample by Attributes	32
1.3.5 Step 5: Export Layer to a File	34
1.3.6 Step 6: Repeat with Highway Linear Referencing System Routes	34
1.3.7 Step 7: Applying Labels	36
1.4. Programmatic Point Symbolology	38
1.4.1 Add a Point Layer	38
1.4.2 Simple Symbols	41
1.4.3 Character Symbols	43
1.4.4 Image Symbols	46
1.4.5 Point Categories.....	51
1.4.6 Compound Symbols	55
1.5 Programmatic Line Symbolology	57
1.5.1 Adding Line Layers	57
1.5.2. Simple Line symbols	57
1.5.3. Outlined Symbols	58
1.5.4 Unique Values	59
1.5.5 Custom Categories.....	60
1.5.6 Compound Lines	61

1.5.7. Line Decorations	64
1.6 Programmatic Polygon Symbology	65
1.6.1 Add Polygon Layers	65
1.6.2. Simple Patterns	66
1.6.3. Gradients.....	66
1.6.4. Individual Gradients	67
1.6.5 Multi-Colored Gradient.....	68
1.6.6. Custom Polygon Categories	69
1.6.7 Compound Patterns	70
1.7 Programmatic Labels	72
1.7.1 Field Name Expressions	72
1.7.2 Multi-Line Labels.....	73
1.7.3. Translucent Labels	74
1.8. Programmatic Raster Symbology	75
1.8.1 Download Data	75
1.8.2. Add a Raster Layer	76
1.8.3 Control Category Range	78
1.8.4. Shaded Relief	79
1.8.5. Predefined Schemes	80
1.8.6 Edit Raster Values	81
1.8.7. Quantile Breaks.....	82
1.9. Extension Methods	84
1.10 Adding Additional Plugins and Extensions.....	86
1.10.1 GDAL.....	86
1.10.2. New Ribbon.....	95
2. Acknowledgements.....	98

1. Developer's Corner

This section gives a bit of an overview of geometric relationships, which are important for understanding some of the vector analysis options, and then gives some workable exercises to demonstrate the current capabilities of DotSpatial 1.7. This should be thought of as a kind of developer's preview. This part of the document covers using DotSpatial components, stitching together a working GIS by dragging the important components from the toolbox in Visual Studio, and then gives a detailed description of how to programmatically add layers and work with symbology or

complex symbolic schemes.

A huge focus for the .Net version of DotSpatial is to put much more control into the hands of developers. By providing everything in the form of .Net components, it makes it far simpler to pick and choose what sections of the framework you want to work with. If the map is all you need, you don't need to bother adding the legend, our custom status strip, or the toolstrip. However, those components are provided for you so that it you can stitch together a working GIS with minimal writing of code.

1.1 Geometry Cheat Sheet

In addition to organizing coordinates for drawing, the geometry classes provide a basic framework for testing topological relationships. These are spatial relationships that are principally concerned with testing how to shapes come together, for instance whether two shapes intersect, overlap, or simply touch. These relationships will not change even if the space is subjected to continuous deformations. Examples include stretching or warping, but not tearing or gluing.

The tests to compare two separate features look at the interior, boundary, and exterior of both features that are being compared. The various combinations form a matrix illustrated in the figure below. It should be apparent that not only are the intersections possible, but each region will have a different dimensionality. A point is represented as a 0 dimensional object, a line by 1 dimension and an area by 2. If the test is not specific to what dimension, it can represent any dimension as "True". Likewise, if it is required that the set is empty, then "False" is used.

	Interior	Boundary	Exterior
Interior			
Boundary			
Exterior			

Figure 1: Intersection Matrix

Graphically, we are illustrating the intersection matrix for two polygons. Some tests can be

represented by a single such matrix, or a single test. Others require a combination of several tests in order to fully evaluate the relationship. When the matrix is represented in string form, the values are simply listed in sequence as you would read the values from the top left row, through the top row and then repeating for the middle and bottom rows. The following are all possible values in the matrix:

- T: Value must be “true” – non empty – but supports any dimensions ≥ 0
- F: Value must be “false” – empty – dimensions < 0
- *: Don’t care what the value is
- 0: Exactly zero dimensions
- 1: Exactly 1 dimension
- 2: Exactly 2 dimensions

The following is a visual representation of the test or tests required in each case. A red X indicates that the test in those boundaries must be false. A colored value requires that the test be true, but doesn’t specify a dimension. A gray value indicates that the test doesn’t care about the value of that cell.

- **Contains:**

- Every point of the other geometry is a point of this geometry, and the interiors of the two geometries have at least one point in common.
- T*****FF*

	I	B	E
I			
B			
E			

- **Covered By:**

- Every point of this geometry is a point of the other geometry.
- T*F**F***, *TF**F***, **T*F*** or **F*TF***

	I	B	E
I			
B			
E			

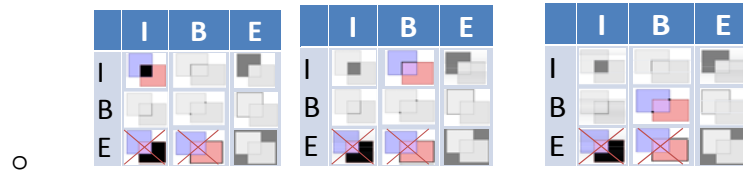
	I	B	E
I			
B			
E			

	I	B	E
I			
B			
E			

	I	B	E
I			
B			
E			

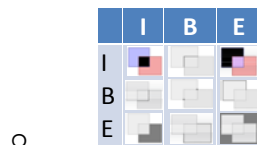
- **Covers:**

- Every point of the other geometry is a point of this geometry.
- T*****FF* or *T*****FF* or *****T*FF*

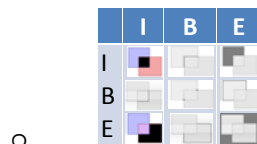


- **Crosses:**

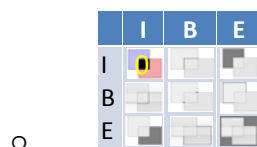
- Geometries have some but not all interior points in common.
- T*T***** (for Point/Line, Point/Area, Line/Area)



- T*****T** (for Line/Point, Line/Area, Area/Line)

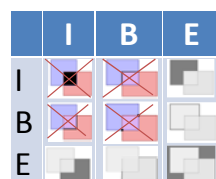


- 0***** (for Line/Line Situations)



- **Disjoint:**

- The two geometries have no point in common.
- FF*FF****



○

- **Intersects:** NOT Disjoint

- The two geometries have at least one point in common.

- **Overlaps:**

- The geometries have some but not all points in common, they have the same dimension, and the intersection of the interiors of the two geometries has the same dimension as the geometries themselves.

- $T^*T^{***}T^{**}$ (for Point/Point or Area/Area)



○

- $1^*T^{***}T^{**}$ (for Line/Line)

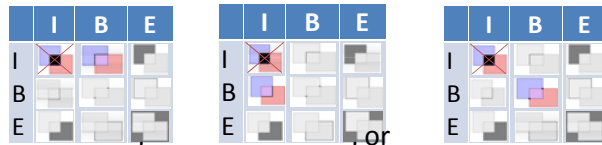


○

- **Touches:**

- The two geometries have at least one point in common but their interiors do not intersect.

- FT^{*****} , $F^{**}T^{*****}$, or $F^{***}T^{*****}$

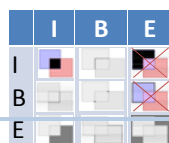


○

- **Within:**

- Every point of this geometry is a point of the other geometry and the interiors of the two geometries have at least one point in common.

- $T^*F^{**}F^{***}$



1.1.1 Overlay Operations:

Being able to test the existing relationships between geometries is extremely useful for doing analysis, but many times you need to alter the geometries themselves. Frequently you want to use other geometries to accomplish this. Consider the case of a clipping operation. In the figure below, the rivers extend beyond the boundaries of the state of Texas. Using an overlay operation is exactly the kind of operation that helps with this kind of calculation. These are not limited to specific scenarios like polygon to polygon. Instead the same terminology applies to all the geometries using the following definitions.

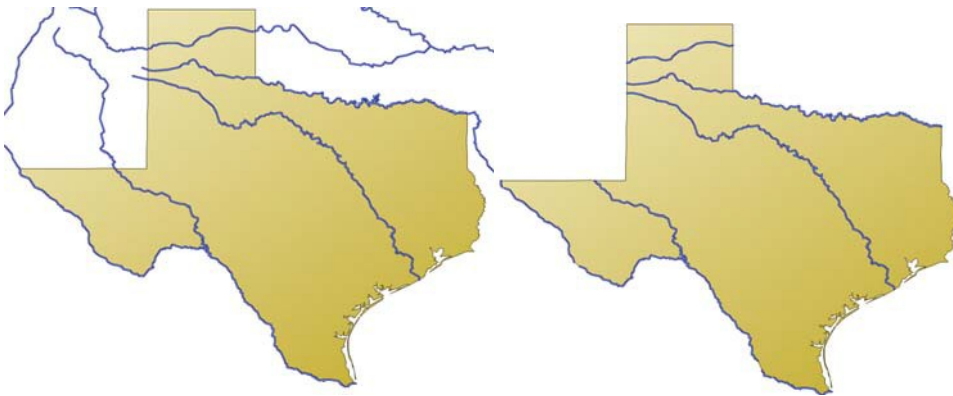
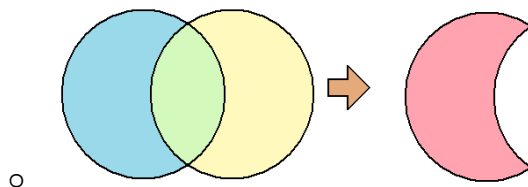


Figure 2: Before and After Clipping Rivers to Texas

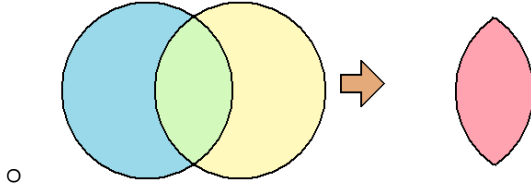
- **Difference:**

- Computes a Geometry representing the points making up this geometry that do not make up the other geometry.



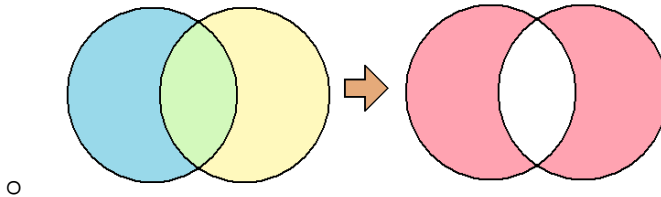
- **Intersection:**

- Computes a geometry representing the points shared by this geometry and the other geometry.



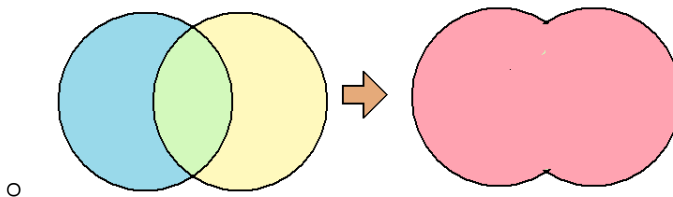
- **Symmetric Difference:**

- Computes a geometry representing the points in this geometry that are not present in the other geometry, and the points in the other geometry that are not in this geometry.



- **Union:**

- Computes a geometry representing all the points in this geometry and the other geometry.



1.2 Exercise 1: Assemble a Map Project

Putting together a GIS project has never been easier. Even a novice developer can take advantage of our ready-built mapping controls, dragging and dropping them onto a solution. Because the components are largely independent, they can be re-arranged in different layouts, or used separately. Furthermore, the extensive use of interfaces for the controls allows a control like the map control to be used interchangeably with alternate controls that act like a legend.

In this first exercise, we will take the assembly step by step. If you have never worked with anything besides the built in .Net controls, this exercise will be useful because it will demonstrate how to add the DotSpatial components to your Visual Studio developer toolbox. It will also show the basic way that the most fundamental map controls can be added to the map.

1.2.1 Step 1: Start a New C# Application

The first step of the exercise is to create a brand new application. Rather than working with an existing application, the goal here is to show that getting from a blank application to a fully operational GIS takes only a few minutes in order to add the components and link them together. To get to new project dialog in Visual Studio, simply navigate to File, New, and choose Project from the context menu. This will display the New Project dialog displayed in figure 3.

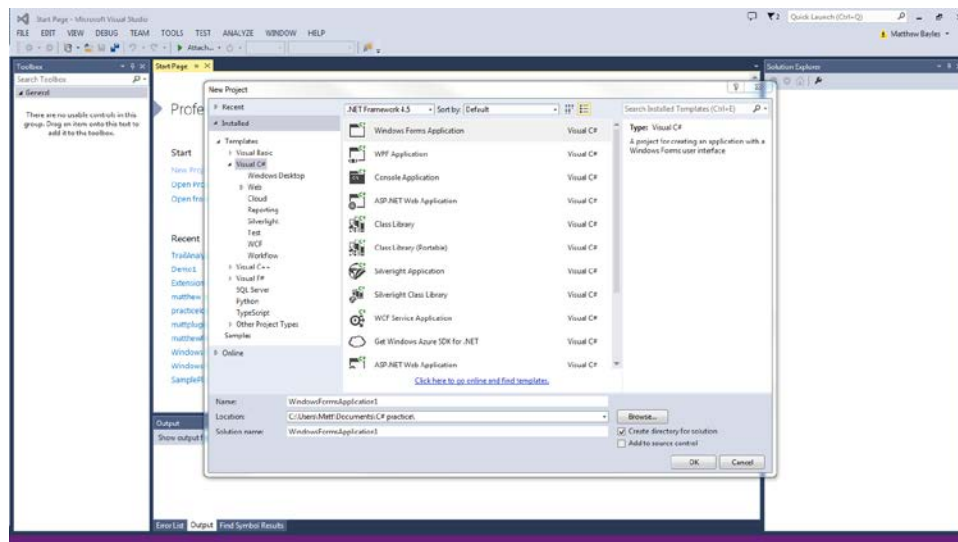


Figure 3: New Project Dial

Change the name and path to something appropriate. In this case, we chose “Build_A_Map” and C:\Users\Matt\documents\DotSpatialDev\Tutorial\Components\Ex1 is the file path. Make sure that the Project type is Visual C# and Windows. Ensure that the Template is set to “Windows Forms Application.” Then click OK.

1.2.2 Step 2: Add DotSpatial Components

The first step in any project is to ensure that you have loaded all of the designer controls and components into the DotSpatial toolbar. First, we want to create a new tab to store the DotSpatial tools. You can do this by right clicking on the toolbox and selecting the Add Tab option from the context menu.

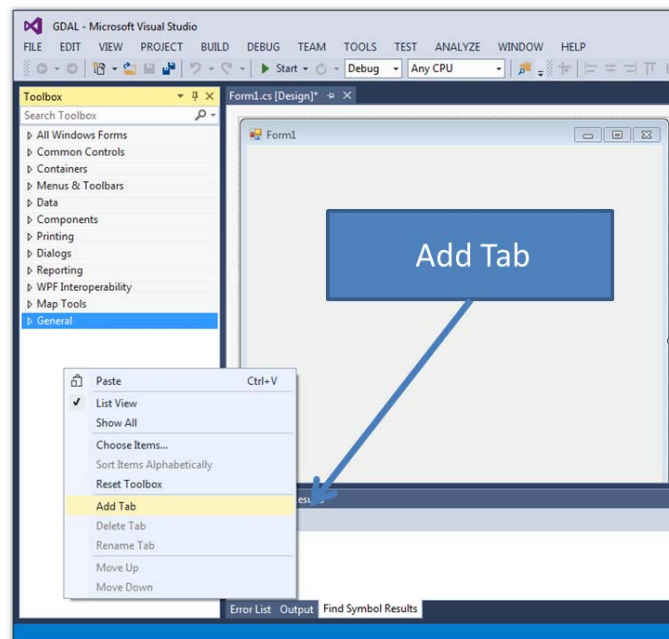


Figure 4: Add DotSpatial Tab

This will allow you to edit the name of the tab. We will name the tab “DotSpatial” so that we can easily keep the controls from the DotSpatial library together. Once you have added a DotSpatial tab, you will want to right click in the blank space below that tab and select “Choose Items” from the context menu.

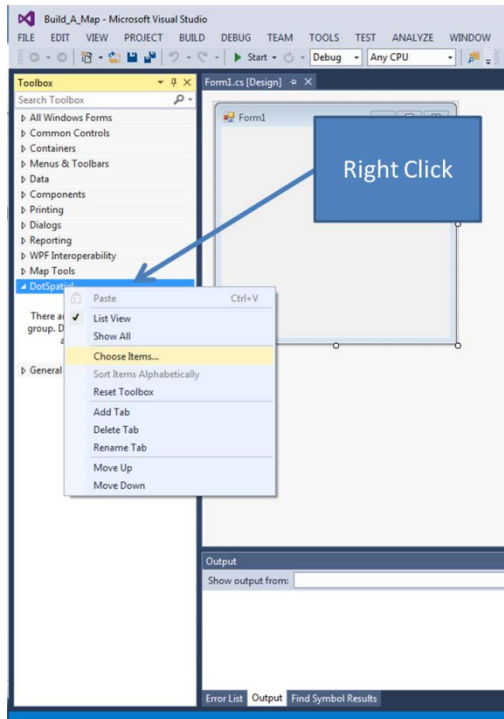


Figure 5: Choose Items

The choose items option launches a new dialog which will allow you to select from various pre- loaded .Net controls as well as some COM controls. However, we are going to use a third option, and browse for the DotSpatial.Controls.dll file.

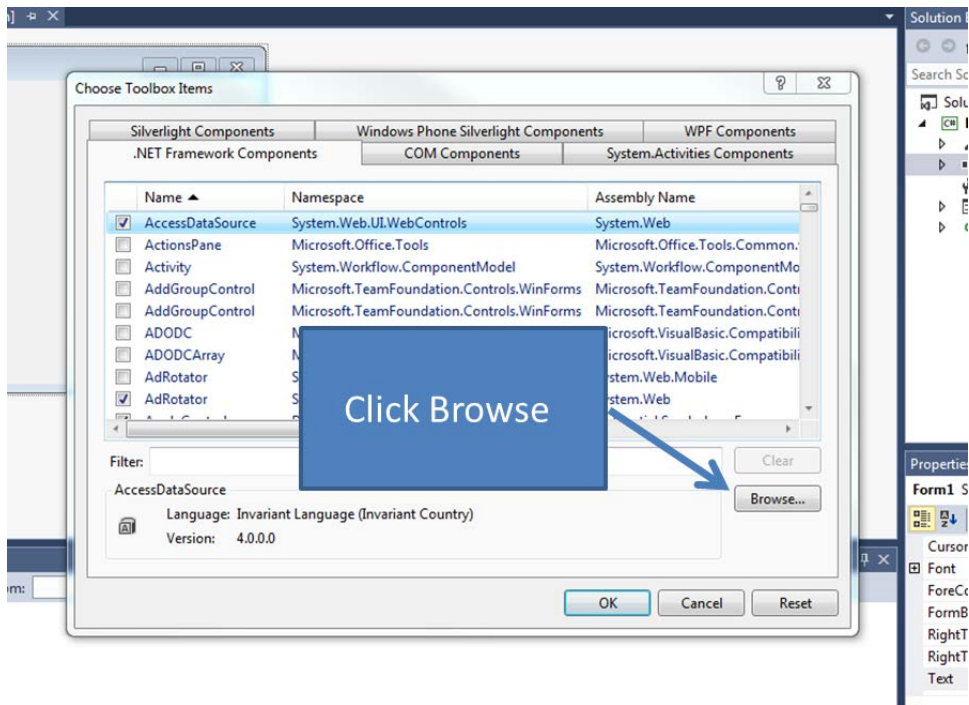


Figure 6: Browse

This, in turn, launches a file browser, and you will have to navigate to where the DotSpatial.Controls.dll file is found on your local machine. On this machine it was in the C:\Users\Matt\Documents\DotSpatialDev\DotSpatial.

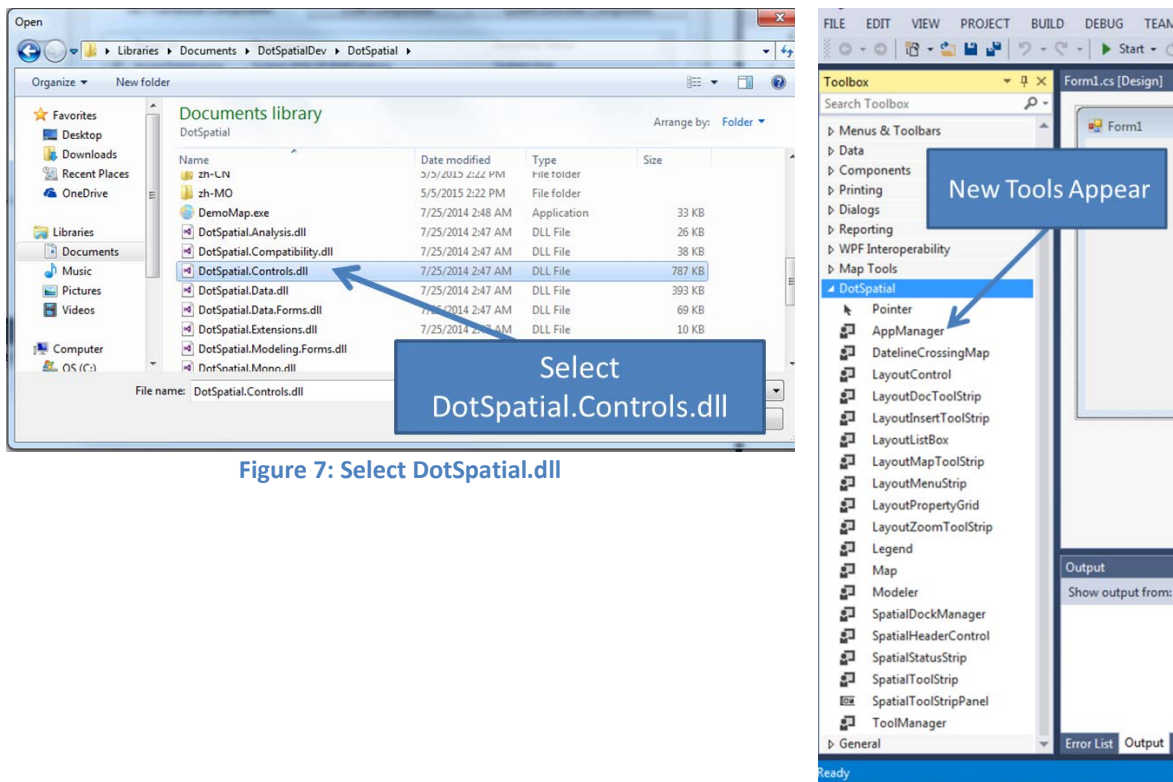


Figure 7: Select DotSpatial.dll

Figure 8: DotSpatial Tools

Once you have chosen this document, it will cause a large number of tools to be added to your toolbox, as is illustrated in the figure to the right.

1.2.3 Step 3: Set up interface for plugin Menu

Next we will add a plugin for a menu, which includes several basic GIS functions like adding data layers, switching between zoom and pan mode, and zooming to the full extent. This is not the same as the “SpatialToolStrip.” This will likely be removed in future versions of DotSpatial. We first need to reference more of the DotSpatial Library as well as an additional .net framework in order to support the plugin. The plugin comes with the DotSpatial download so we will not need to download anything extra but we will need to add some code in order to access it.

In the solution explorer under the project name is a menu option called “References.” If you click to open the menu you should see “DotSpatial.Controls” has already been added from our work in the previous steps. We now need to add more of the DotSpatial library. Right click on “References” and select “Add References.”

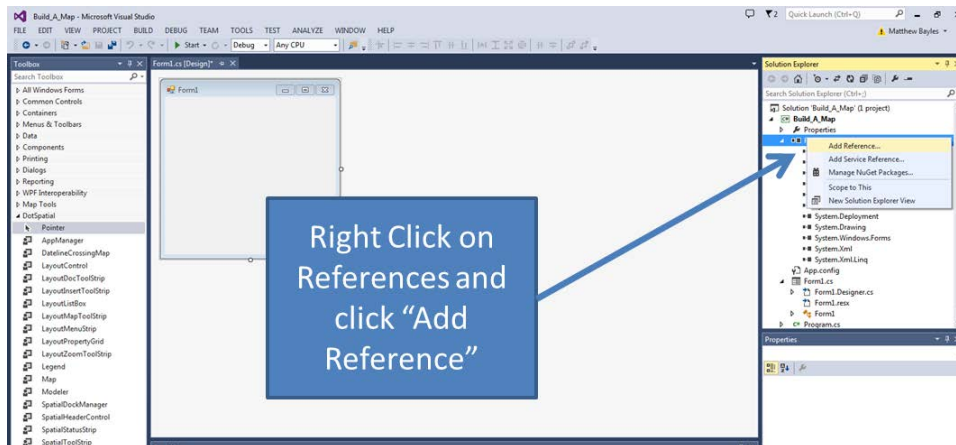


Figure 9: Adding a Reference

A new window will appear. Scroll down to “System.ComponentModel.Composition” and check the box next to it.

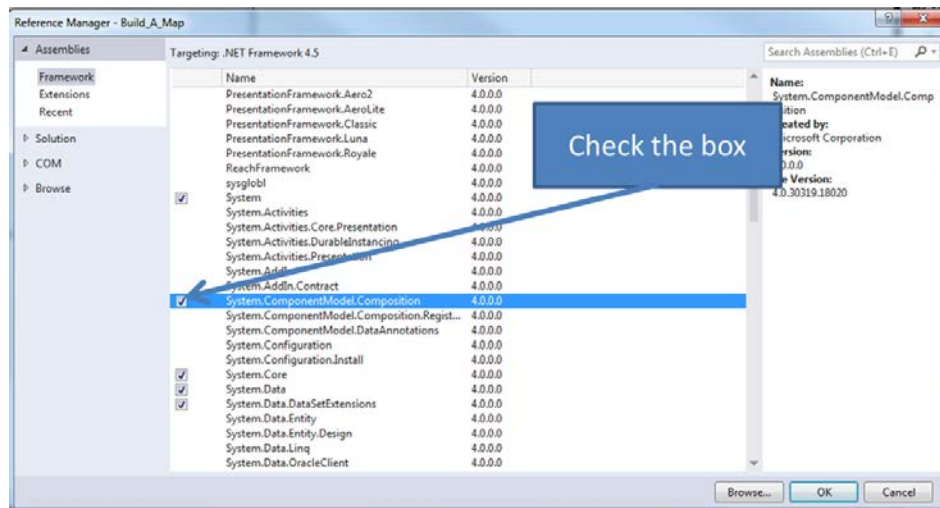


Figure 10: Adding an Assembly

Next go to the “Browse” menu option and click on the “Browse Button.”

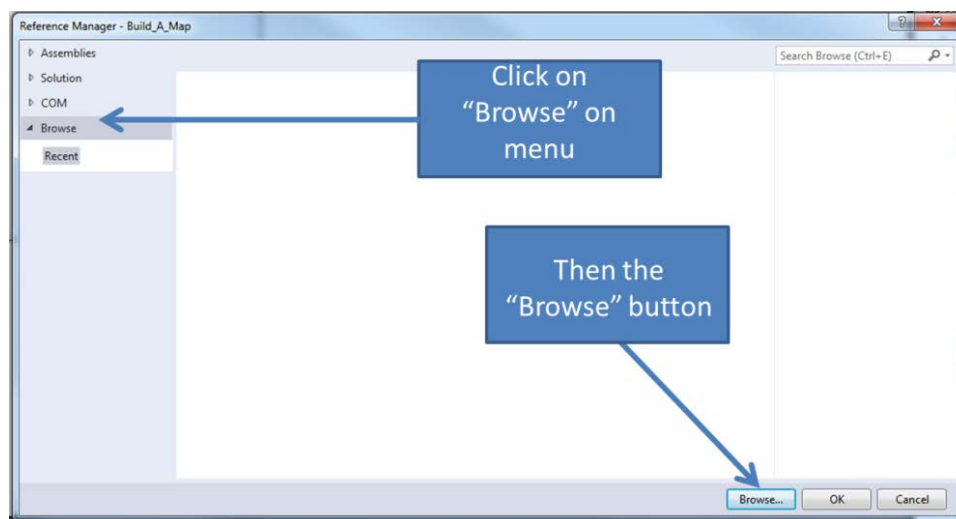


Figure 11: Browsing to Add DLLs

This, in turn, launches a file browser, and you will have to navigate to wherever the DotSpatial dlls are located on your local machine. On this machine it was in the C:\Users\Matt\Documents\DotSpatialDev\DotSpatial. Add all the files except for the “DotSpatial.Controls.dll” since that one has already been added to the project.

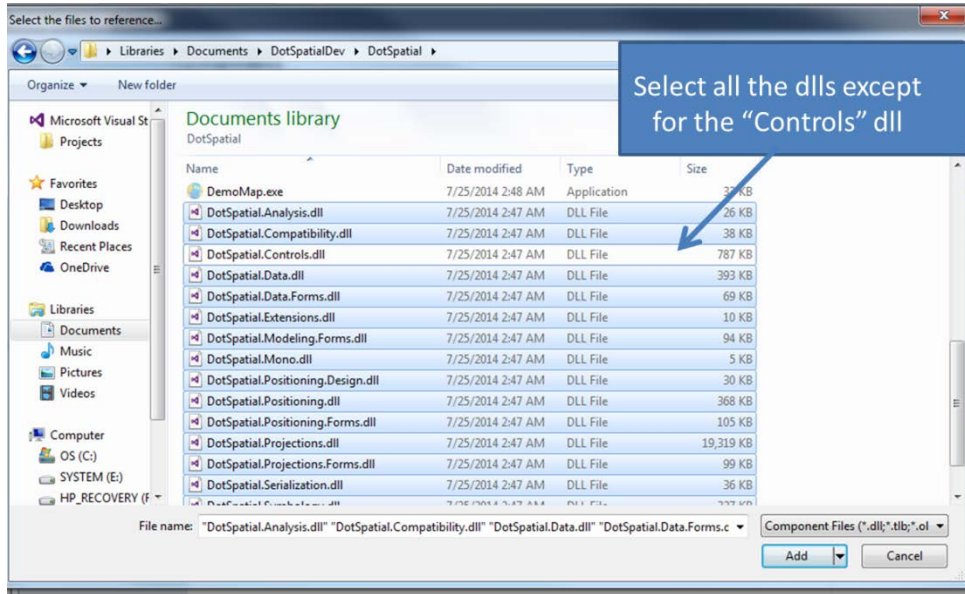


Figure 12: Adding additional DotSpatial dlls

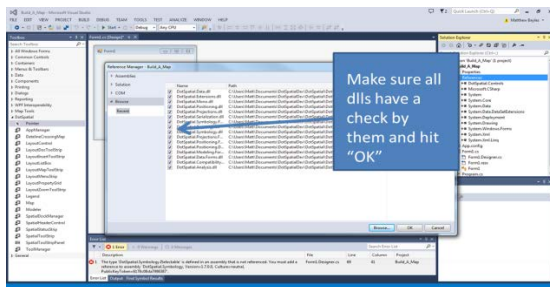


Figure 13: Checking dll status

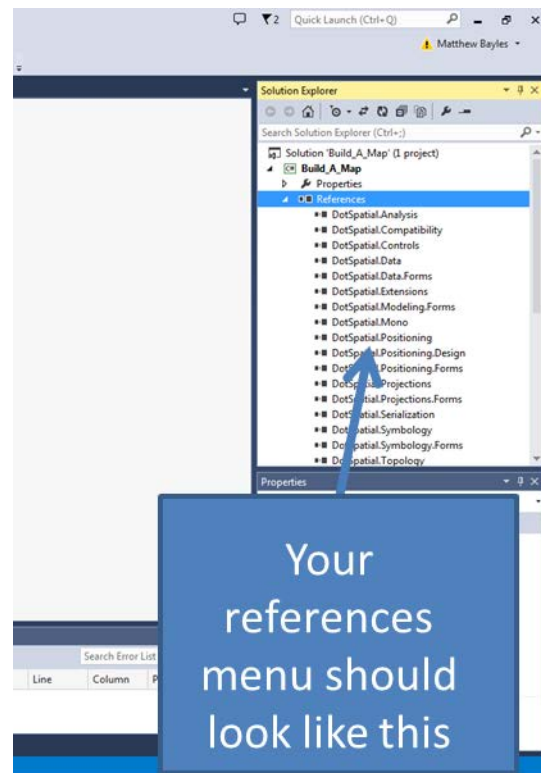


Figure 14: DotSpatial dlls

Your references menu should now look like Figure 14. The next step is to add the AppManger from the DotSpatial tools. This will allow the project to manage the various tools we are going to add to it. Drag and drop the AppManger onto the form and it will appear below the form.

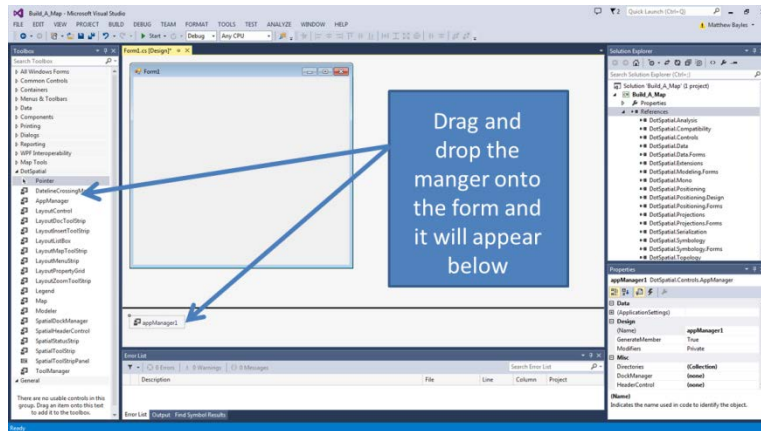


Figure 15: Adding the AppManger

For simplicity we will keep the default names for each of the tools but they should be renamed to avoid confusion and provide meaning in your applications. The next step involves adding code so that our project can add plugins and creating a shell so we can add in the tool strip.

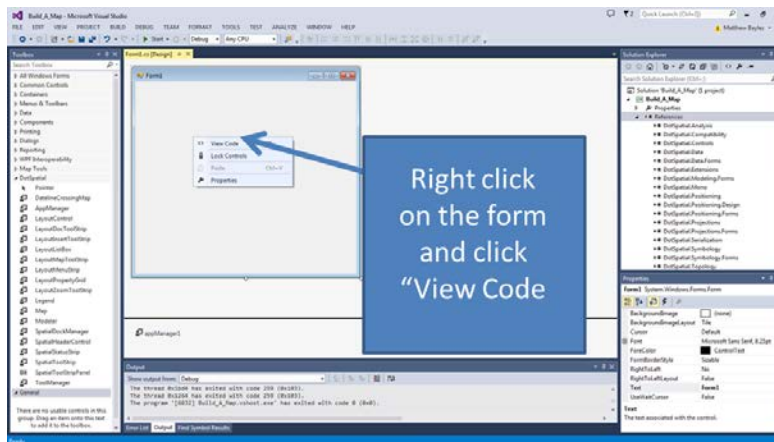


Figure 16: Accessing the Code

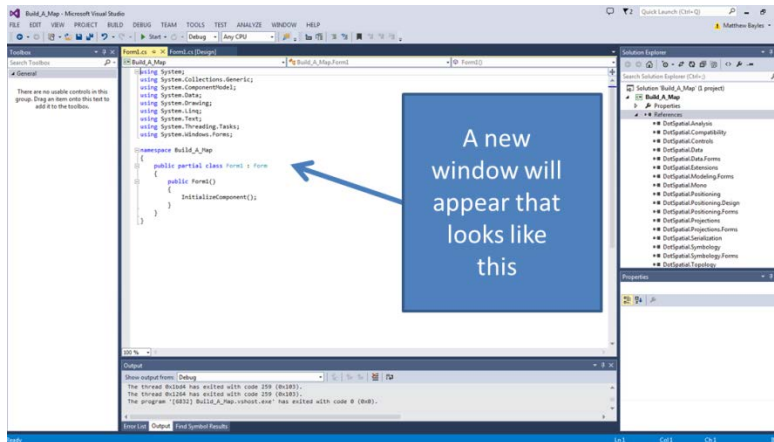


Figure 17: Code Main Page

Your screen should look similar to Figure 17. We will now add code to this page in order to have access to the tool bar plugin. At the top of the page are various “using” statements. These are files outside the main project that the program has access to. We will need to add some code to let the program use the assembly that we added earlier.

After the last “using” statement, type:

```
using System.ComponentModel.Composition;
```

Figure 18: Code Main Page

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.ComponentModel.Composition;

namespace Build_A_Map
{
    public partial class Form1 : Form
    {
        [Export("Shell", typeof(ContainerControl))]
        private static ContainerControl Shell;
        public Form1()
        {
            InitializeComponent();
        }
    }
}

```

Figure 19: Code Main Page

After the “public class Form1 : From” and the “{” type:

```

[Export("Shell",typeof(ContainerControl))]
private static ContainerControl Shell;

```

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.ComponentModel.Composition;

namespace Build_A_Map
{
    public partial class Form1 : Form
    {
        [Export("Shell", typeof(ContainerControl))]
        private static ContainerControl Shell;
        public Form1()
        {
            InitializeComponent();
            if (DesignMode) return;
            Shell = this;
            appManager1.LoadExtensions();
        }
    }
}

```

Figure 20: Code Main Page

After “InitializeComponent();” type:

```

if (DesignMode) return;
Shell = this;
appManager1.LoadExtensions();

```

The menu bar will not appear until the project has been compiled, but we can still add features to our map because the program will automatically put the menu bar at the very top of the form.

1.2.4 Step 4: Adding the Map and other Controls

Now that the menu plugin has been added, we can start adding more DotSpatial controls. In order to cleanly divide up the screen area with a minimum of custom programming, we will take advantage of the DotSpatial “SpatialDockManager” control. This will divide the content into two separate panels that are sizeable by the user.

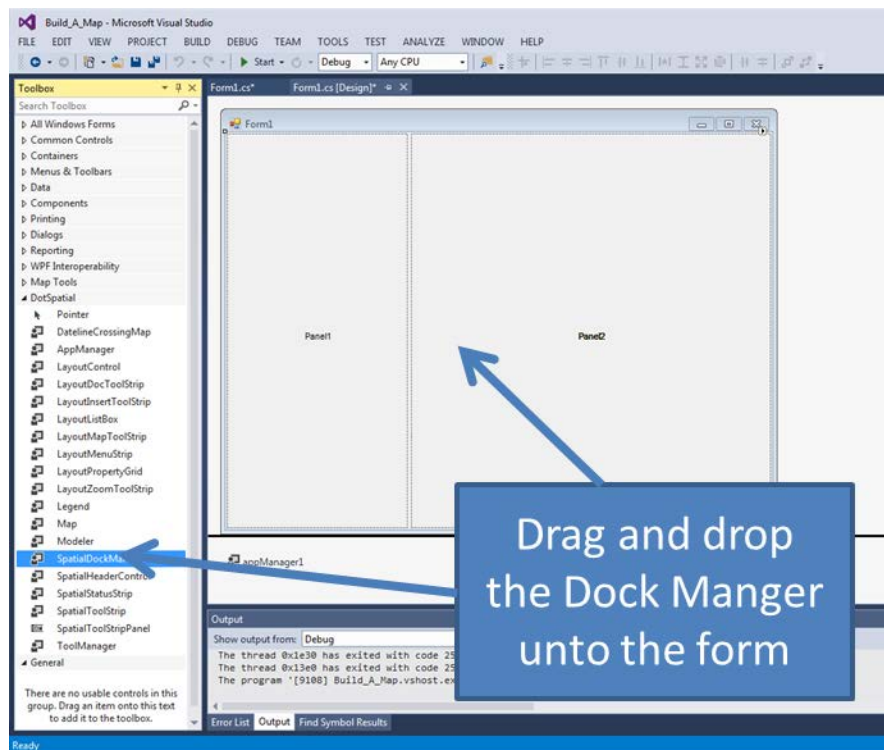


Figure 21: SpatialDockManager

On the left side on “Panel 1” we are going to add a Tab Control which will help us manage the legend and DotSpatial Tools. The tab control is under “All Windows Forms” on the Toolbox. Set the dock to fill.

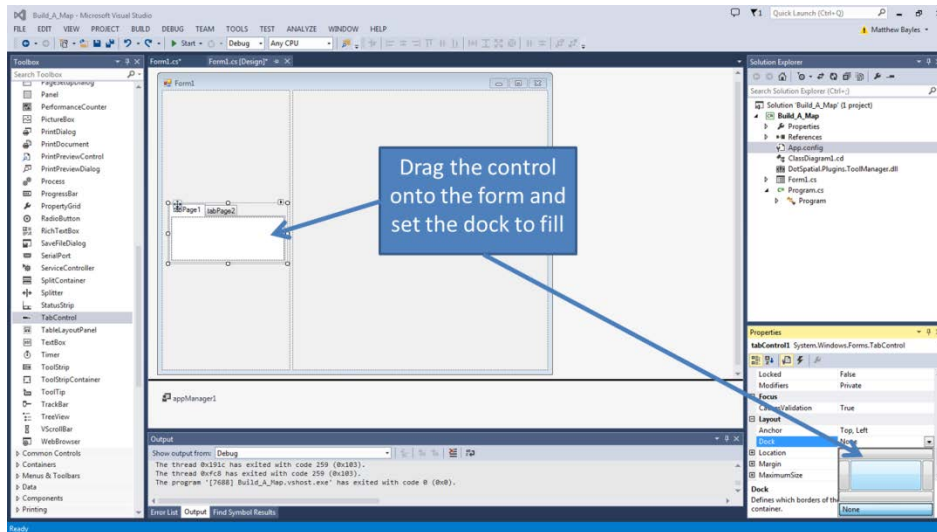


Figure 22: TabControl

Once the separate panels exist, we can add the map to the project. When adding the map, it will likely be the wrong size for the panel that we created. In order to allow the user to resize the map so that it always is the right size for the panel, change the “Dock” property to “Fill”. This can be done by choosing the central rectangle in the drop down editor that appears in the property grid when you click on the down arrow.

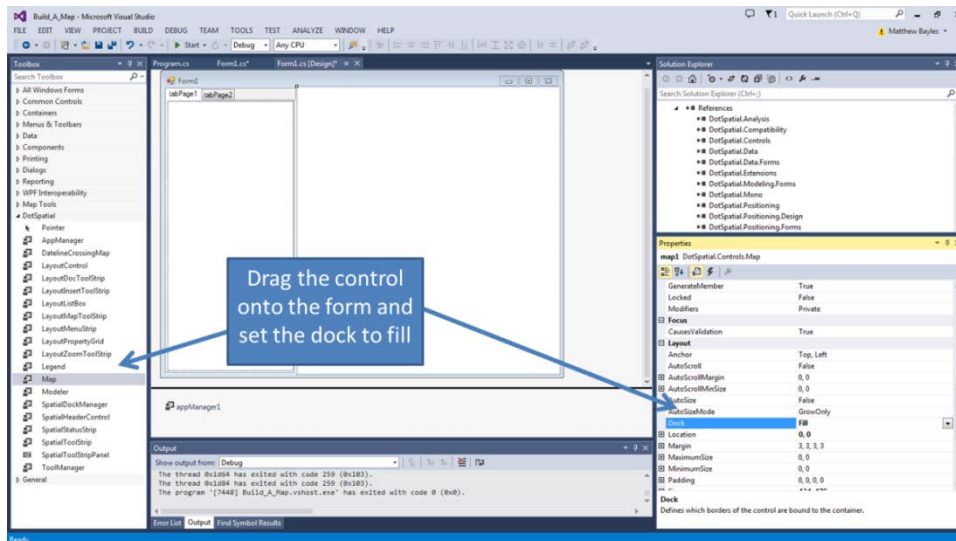


Figure 23: Add a Map

1.2.5 Step 5: Add the Legend and Toolbox

Because the Legend and Toolbox can both exist in support of the map, and it is not critical to have both of these tools visible at the same time, for this project we will take advantage of the .Net Tab control to help re-use the same space more effectively. Because these components are interchangeable, the Tab control is not necessary, and second split panel, fixed panels, or even third party docking panels are all acceptable alternatives that will not affect the proper behavior of the map, legend or toolbox.

To add a legend, drag and drop the DotSpatial Legend onto the first tab and set the dock to fill.

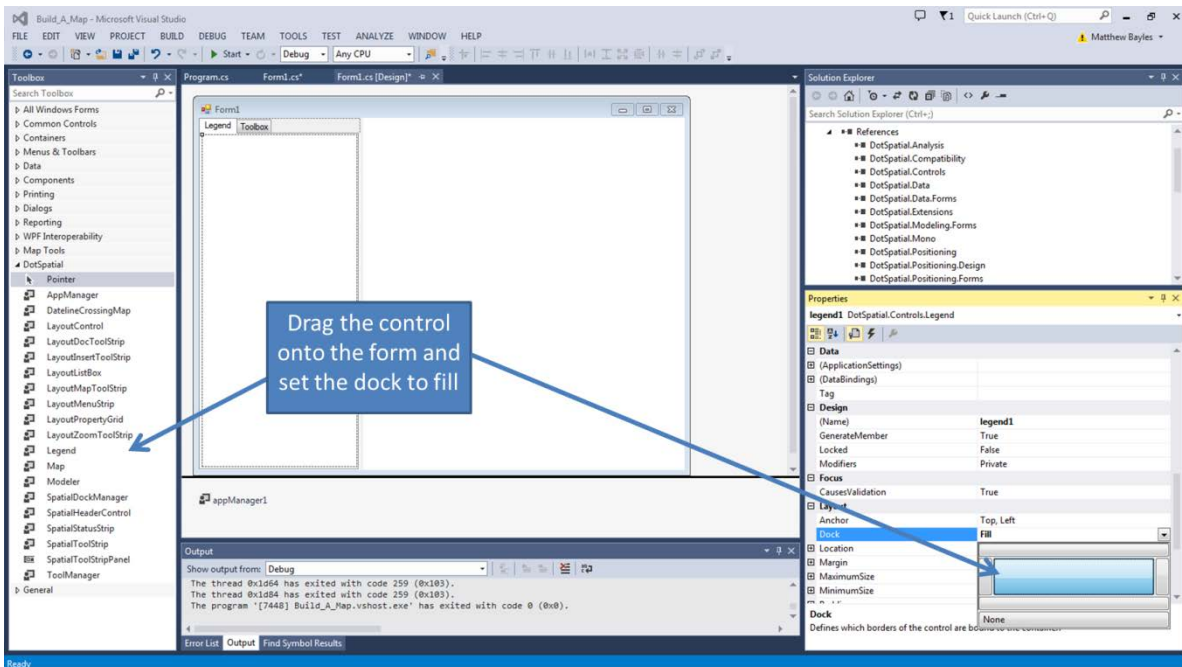


Figure 24: Add a Legend

In this instance we changed the text on the first tab to read Legend. This can be done through the property grid that appears when you activate the tab control by clicking on it after it has been added to the main form.

The toolbox has been designed with basic map analysis tools and has been designed to function as a plugin. In order to have the tools be usable on the form we are going to need to add the “Tools” folder from the DotSpatial folder to our project’s Plugin folder which we will create next.

We will need to create a folder named “Plugins” in bin/Debug folder. The file path on this computer was C:\Users\Matt\Documents\DotSpatialDev\Tutorial\Components\Ex1\Build_A_Map\Build_A_Map\bin\Debug

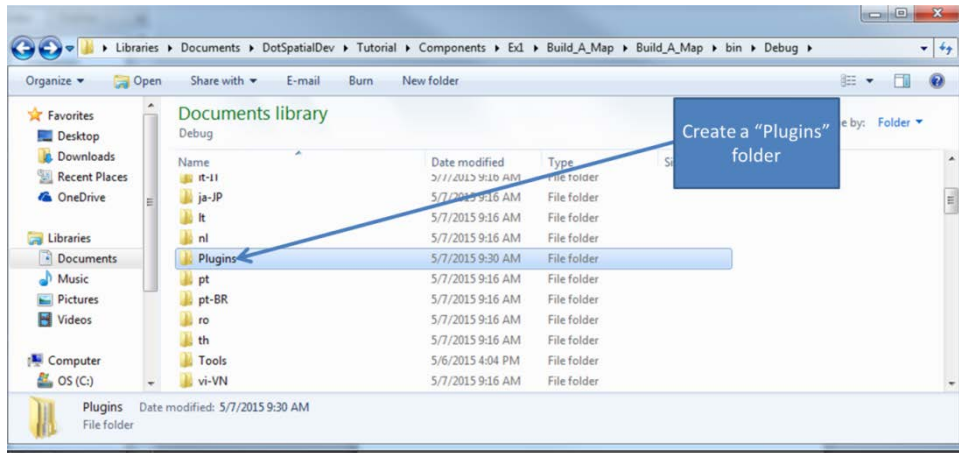


Figure 25: Create a Plugins Folder

Next locate the “Tools” folder in the DotSpatial folder that you downloaded earlier. This is the same folder that has the DotSpatial dlls. Copy this folder and place it inside the Plugins folder.

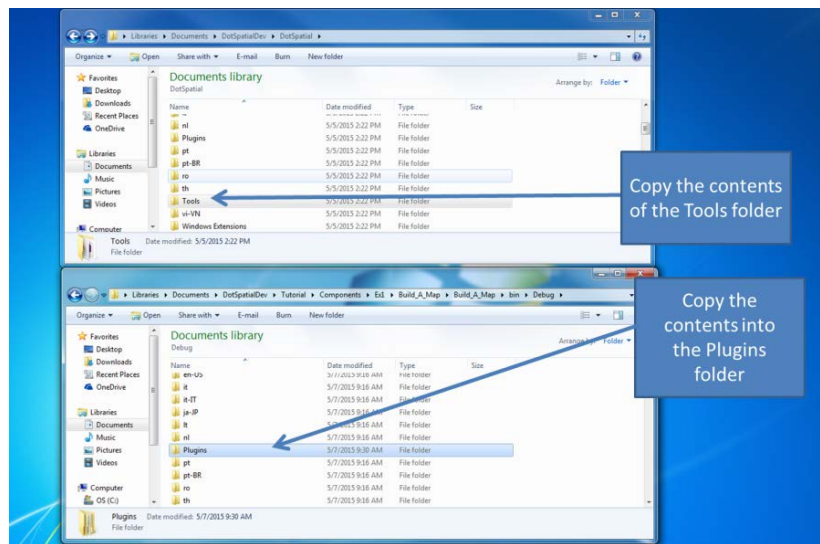


Figure 26: Create a Plugins Folder

Now that we have the “Tools” folder in the “Plugins” folder we now have the basic controls and tools for a map. Once we finish linking the controls together a “Tools” tab will be added next to the “Legend” tab. Therefore we can delete the second tab as we will not need it. The only thing now is to connect each of the controls together.

1.2.6. Step 6: Link It All Together

We could technically run the project right away, but it would not appear to do anything. The add data button, for instance might happily open a file dialog, but nothing would happen when it was finished. In order for the legend to show the layers from the map, we need to link things together. We need to change the settings on two controls: the AppManger and the Map.

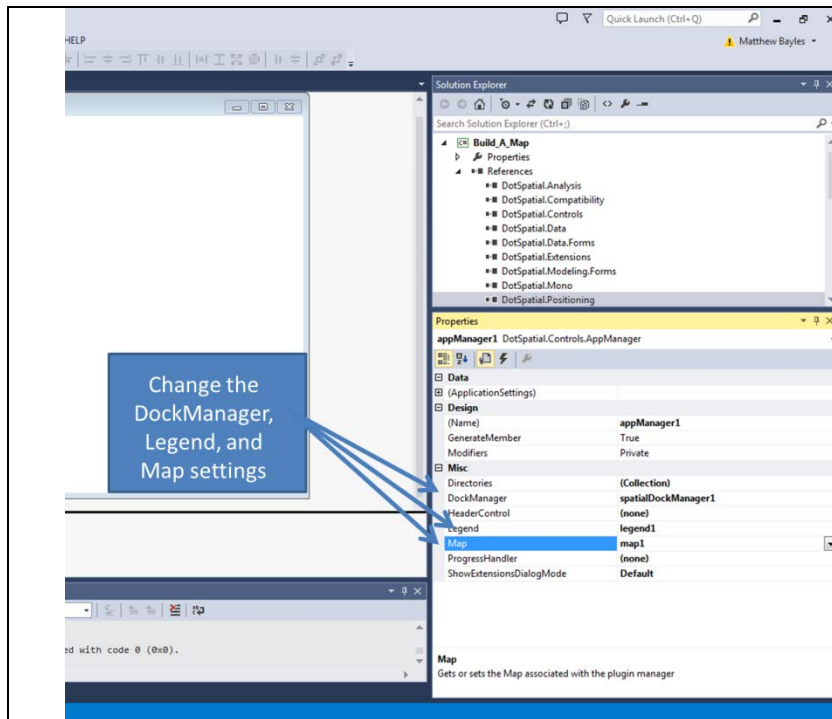


Figure 27: Linking the AppManger

In order to link the AppManger change the settings on the DockManger, Legend, and the Map (found under the Misc. menu) to the names you gave them. If you have not changed the names then the defaults are shown in Figure 27.

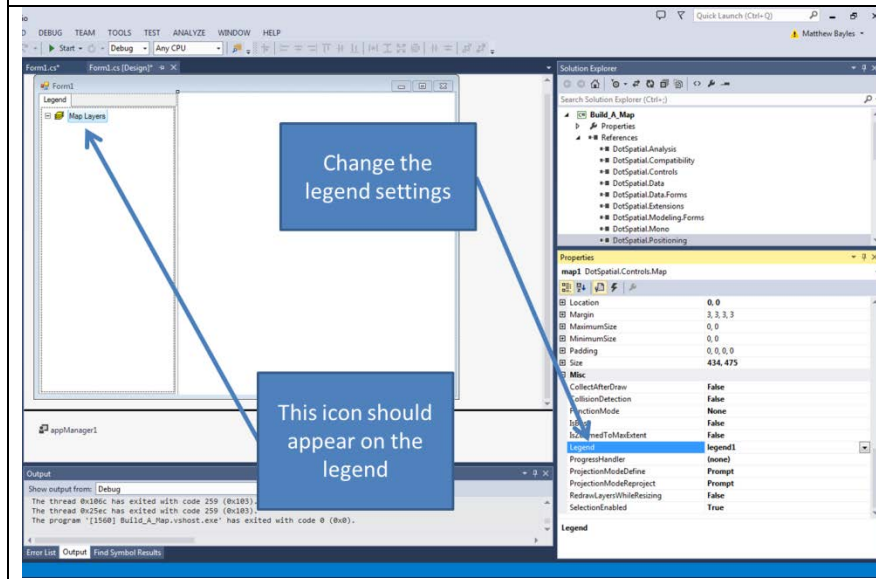


Figure 28: Linking the Map

Next we need to link the Map. There is only one setting we need to change, which is the legend. Change that setting to the name of you legend and your legend should now display an icon which reads "Map Layers."

Now that our project is connected, the first thing to do is to get some sample data. For our project, almost any data in shapefile format will do, but in the spirit of improving online data awareness, this book features many online data sources that should provide up-to-date GIS data. As example we will use a shapefile from the U.S. Census, which can be found among other files here <http://www.census.gov>



Help Tip

Sometimes links can become broken or out of date. In cases like this, it is often useful to look at the first part of the address and then search for data manually. Example: instead of https://www.census.gov/geo/maps-data/data/cbf/cbf_state.html you could use <http://www.census.gov>

The raw data in this case is stored in the form of a zip file. Most modern operating systems can unzip files automatically, but in the event that you need an unzip utility, a free, open source utility called 7-zip is available for download from <http://www.7-zip.org>. Once you have downloaded and extracted the shapefiles, you will see a .shp, a .shx and a .dbf file all with the same name before the extension. This is the basic file format known as an ESRI shapefile, and is a commonly used format for GIS analysis because it is portable and has an open standard and so is widely compatible between different GIS software vendors.

The figure above illustrates the view of the continental United States. Because the data also includes counties in Hawaii, Alaska and Puerto Rico, it will be necessary to zoom in a little to see a view like the one above. You can also use the mouse wheel to zoom in or out of the scene and click on the “hand” icon to pan. We will next look at some of the features of DotSpatial.

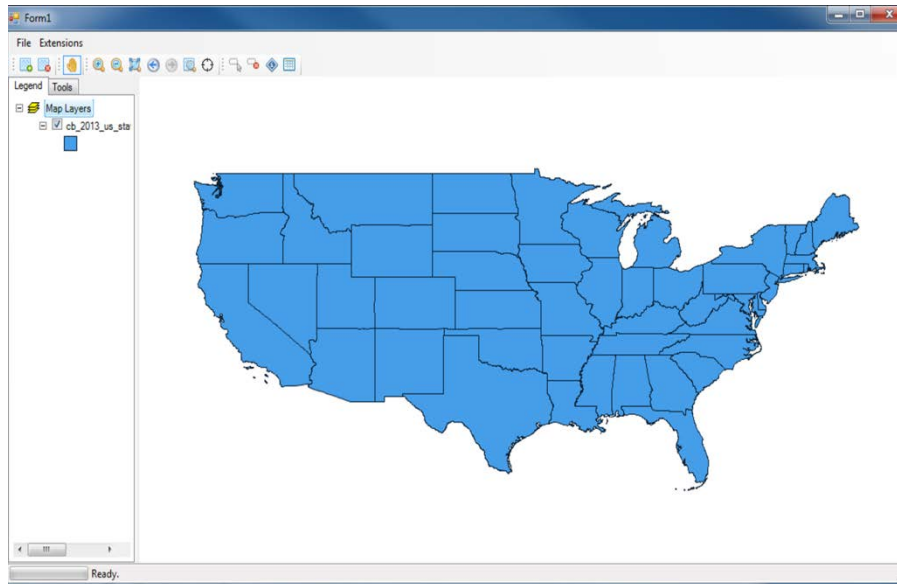


Figure 29: Map of the United States

1.3 Exercise 2: Simplify Australia Data Layers

1.3.1 Step 1: Download Data

For this exercise, we need some thematic vector layers that represent more than just polygons. For this project we will use shapfiles from the Utah Automated Geographic Reference Center. Producing high quality maps for the State of Utah, the Reference Center has been active for over 30 years. Access to these maps is free and they can be found at:

<http://gis.utah.gov/>

For this exercise, we downloaded the Municipal, County and State Boundaries; Lakes, Rivers, Streams, & Springs; and the Highway Linear Referencing System Routes shapefiles. These files are stored in zip format, so you will have to unzip them first (see exercise 1). Before we do any programmatic symbolizing, we will want to convert the polygons from the municipalities' shapefile to a set of points. Fortunately DotSpatial has the tools necessary to do this operation.

Before we begin these exercises, however, we need to be able to access the attribute table. The attribute table is a Plugin that comes in the DotSpatial download. To activate it, simply copy the "DotSpatial.Plugins.TableEditor" from the downloaded DotSpatial folder to the Plugins folder we created in the last exercise.

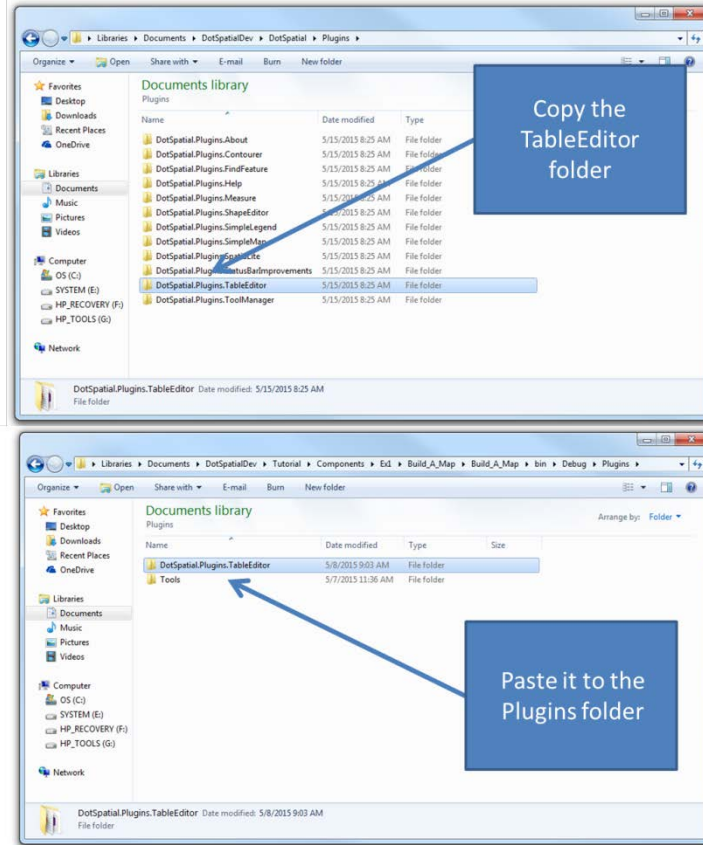


Figure 30: Adding the Attribute Table

Now that we have access to the attribute table we are ready to start modifying our shapefiles.

1.3.2 Step 2: Calculate Polygon Areas

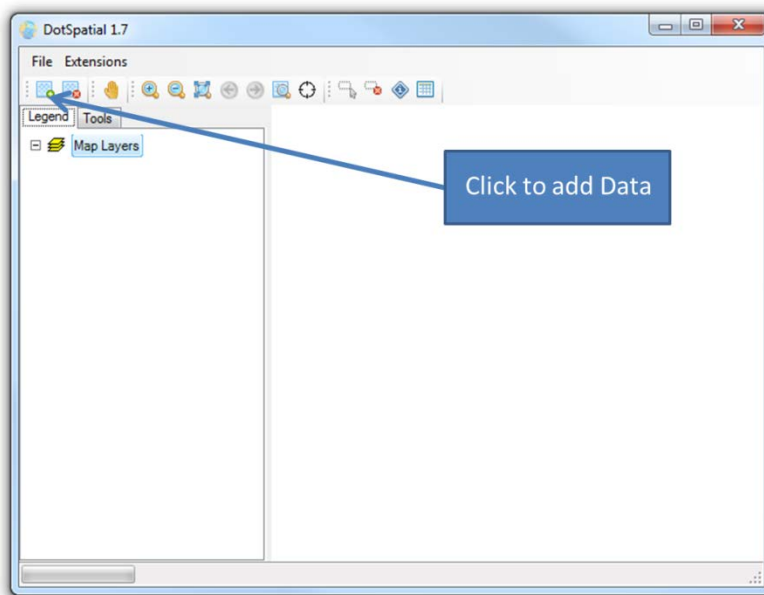


Figure 31: Adding Data

After clicking the green plus to open a file dialog, browse for the “Utah.shp” and the “Municipalities.shp” shapefiles you just downloaded and extracted. When they open, you should see polygons that represent large city areas and the state of Utah, mostly at the center of Utah. Since the polygons are small compared to the size of the entire state, we can see that the city regions are in fact polygons by zooming into the central area.

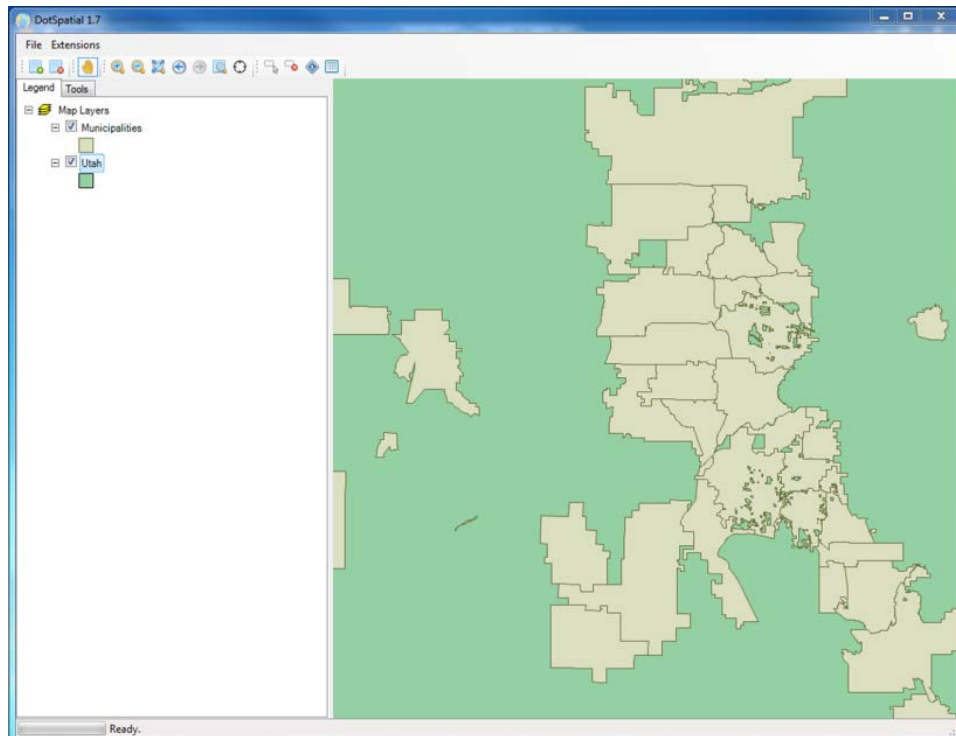


Figure 32: Municipalities Polygons

We will eventually like to be able to distinguish the largest cities. We will use the area calculation tool to calculate areas for each of the polygons. These values will be used to separate the larger cities from the small ones. To see the toolbox, simply change the tab to the Toolbox tab. While this shapefile already includes the areas of the municipalities, it is useful to go through the process as an example of how to use the DotSpatial tools.

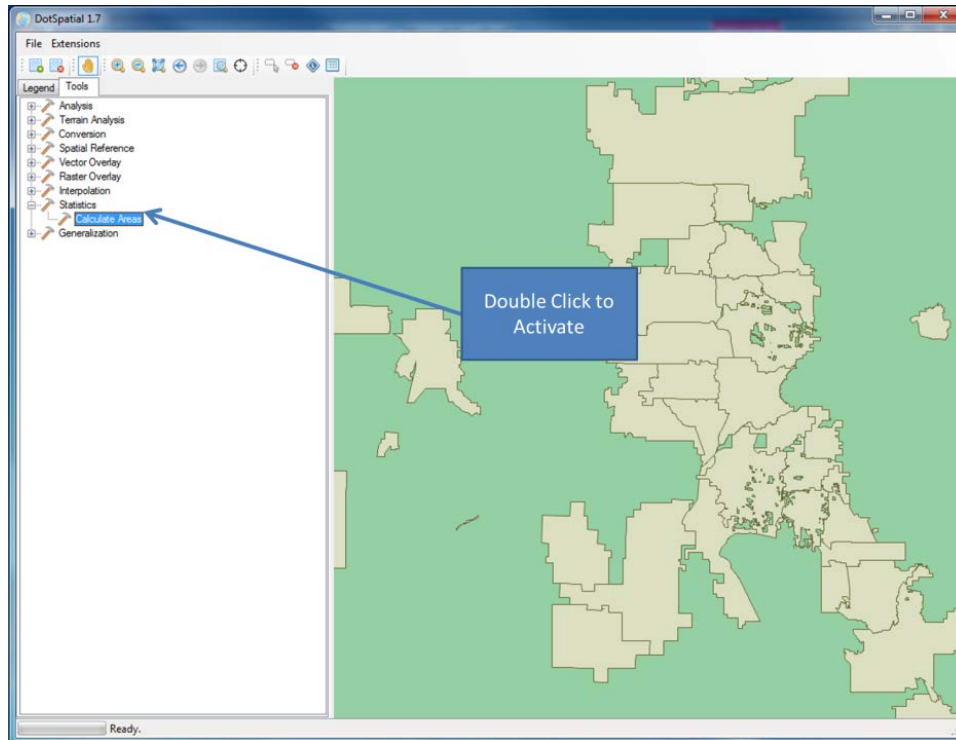


Figure 33: Calculate Area

A new window will open and have two user fields. The first field is the input feature set. In this case it is our Municipalities shapefile. The next field is the resulting feature set. When the green plus is clicked, it will open a new window allowing us to specify a storage location and a name for our new shapefile. In this example we named our shapefile "Area of Municipalities."

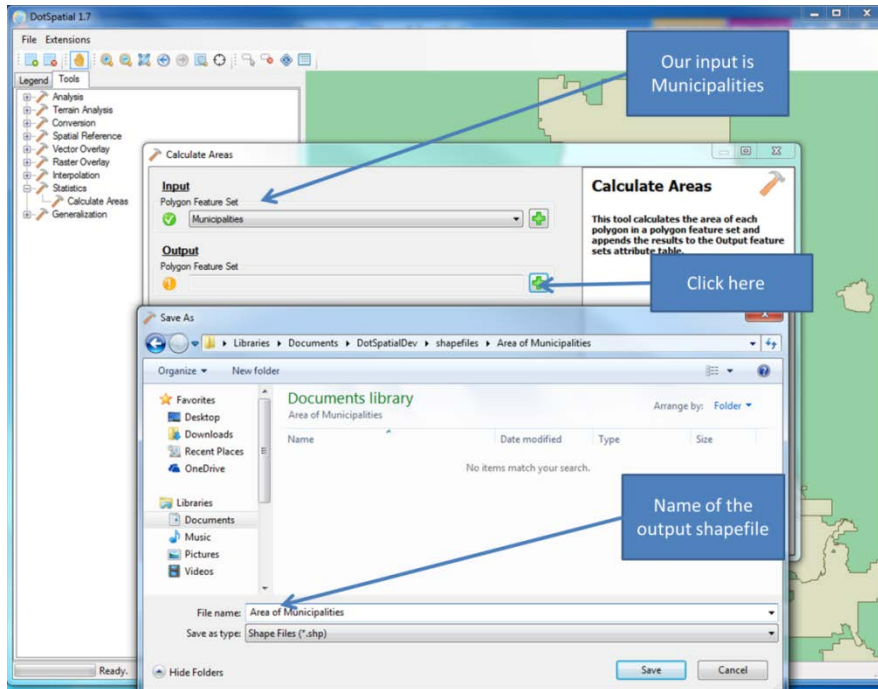


Figure 34: Making a new shapefile

Once we click save the program will calculate the areas of the municipalities and a warning window will appear. Make sure to use the map's coordinate system.

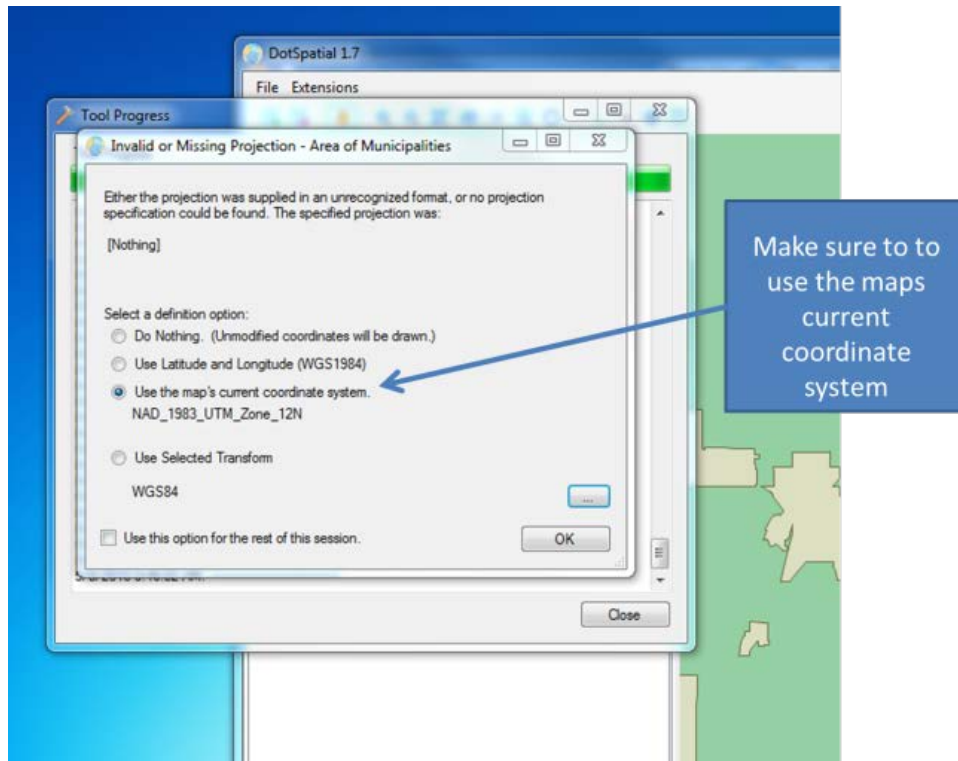


Figure 35: Coordinate System

The resulting shapefile has been added to the map as a new layer. If we open the attribute table we can see that a new field has been added to the resulting shapefile that shows the area.

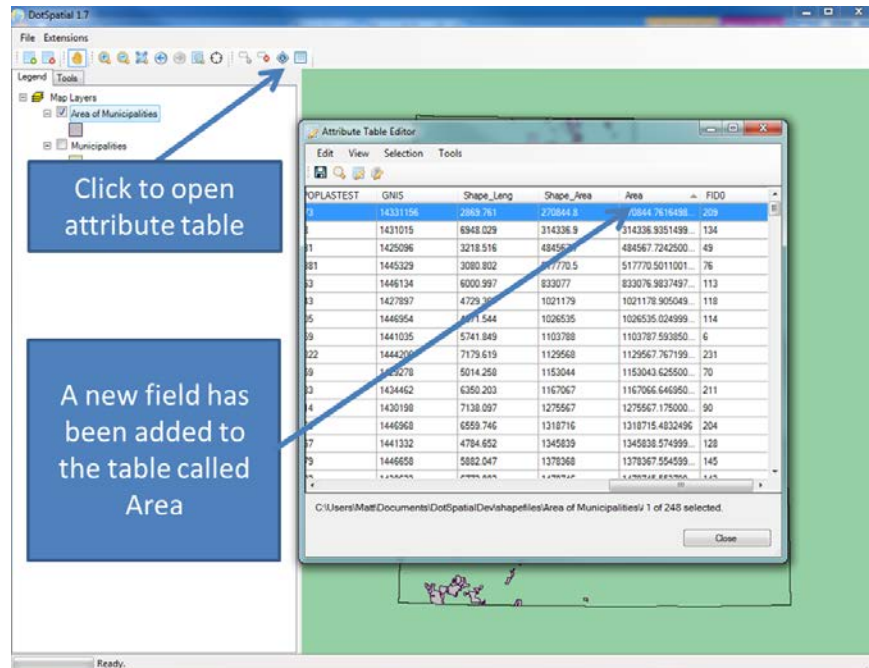


Figure 36: Newly Added Area

1.3.3. Step 3: Compute Centroids

Now that we have a way to order the cities by way of areas, we can now create a simple cities layer from the existing polygons. We can do this by using the centroid tool.

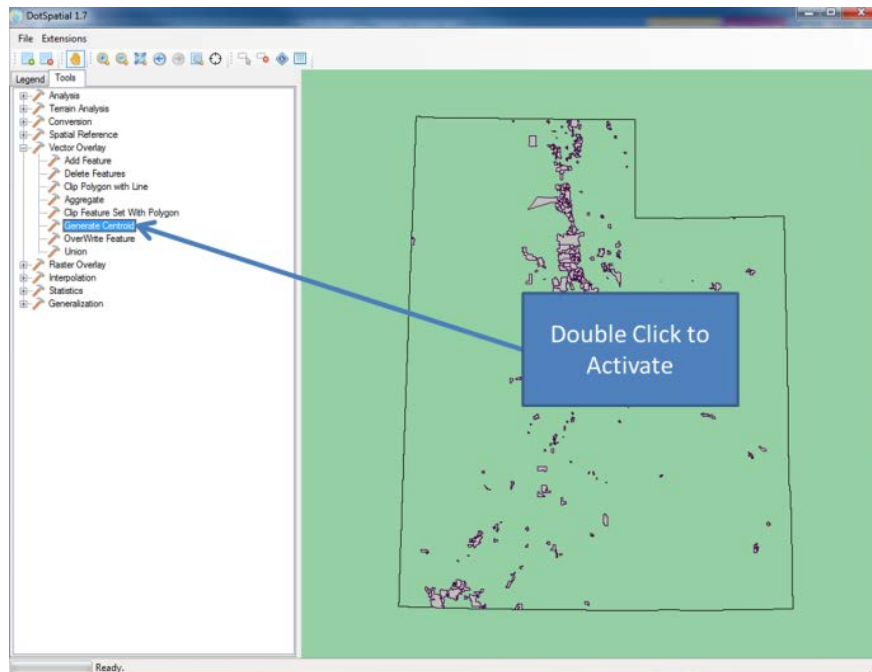


Figure 37: Calculate Centroid

The same windows will appear for calculating centroids as for when we calculated the area. Follow the same steps. The result should look something like Figure 38.

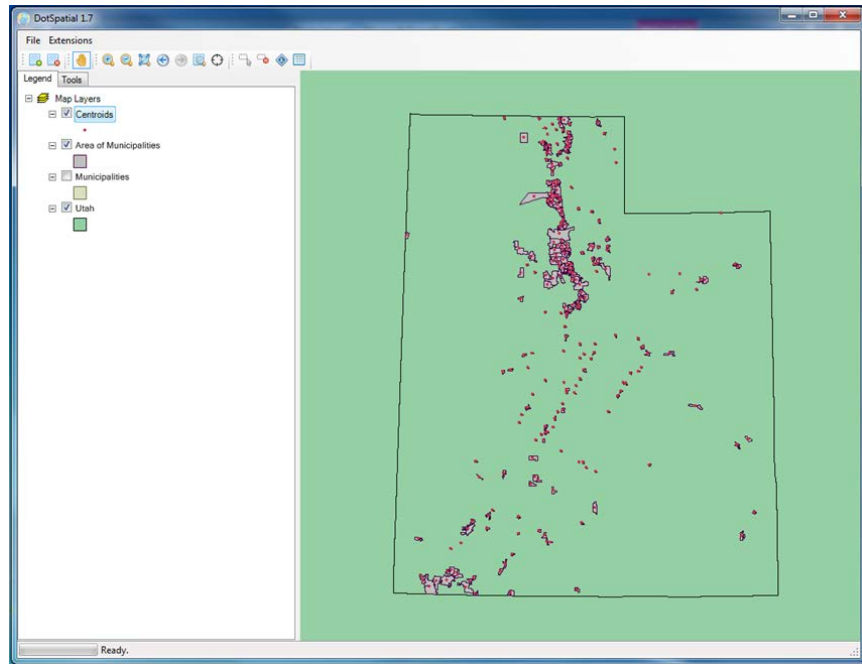


Figure 38: Too Many Cities

The shapefile created by calculating the centroids will now be a good representation of the city locations for built up areas, and in addition, it will give us an approximate measure of the size of the built up area using the area attribute.

1.3.4 Step 4: Sub-sample by Attributes

As can be seen from the figure above, there are a few too many cities visible to make a good map. We need to build a cities shapefile with just the largest cities. We can use the select by attributes ability, and in this case choose $[Area] > 1e+08$ as the criteria, in order to select Utah's seven biggest cities.

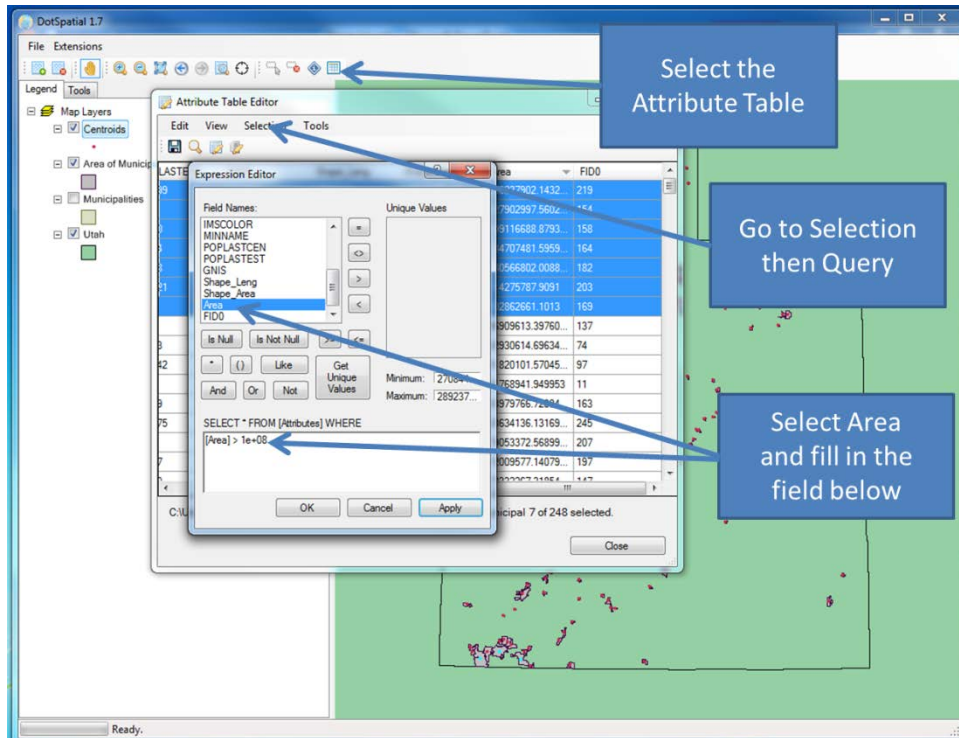


Figure 39: [Area] > 1e+08

We will rename the “Centroids” layer to “Cities.” In order to create a new layer with the features we have selected, we can right click on the cities layer in the legend. By highlighting the “Selection” tab in the context menu, we gain the ability to create a layer from the selected features. This will create a new in-memory shapefile that is not yet associated with a true data layer.

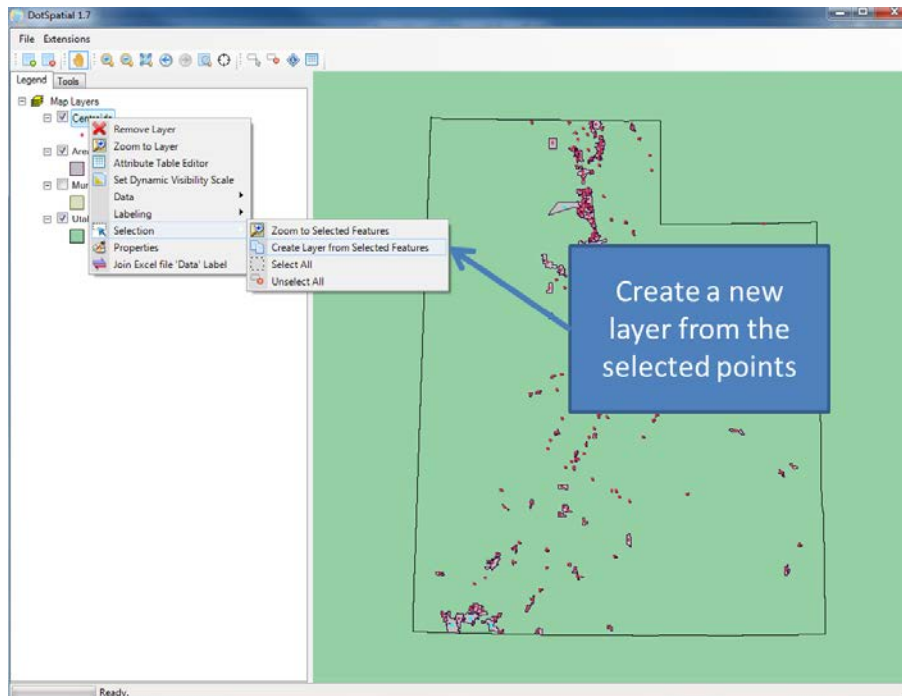


Figure 40: Create a Layer

1.3.5 Step 5: Export Layer to a File

Now that we have created this new layer we need to export it in order to save it as an actual shapefile.

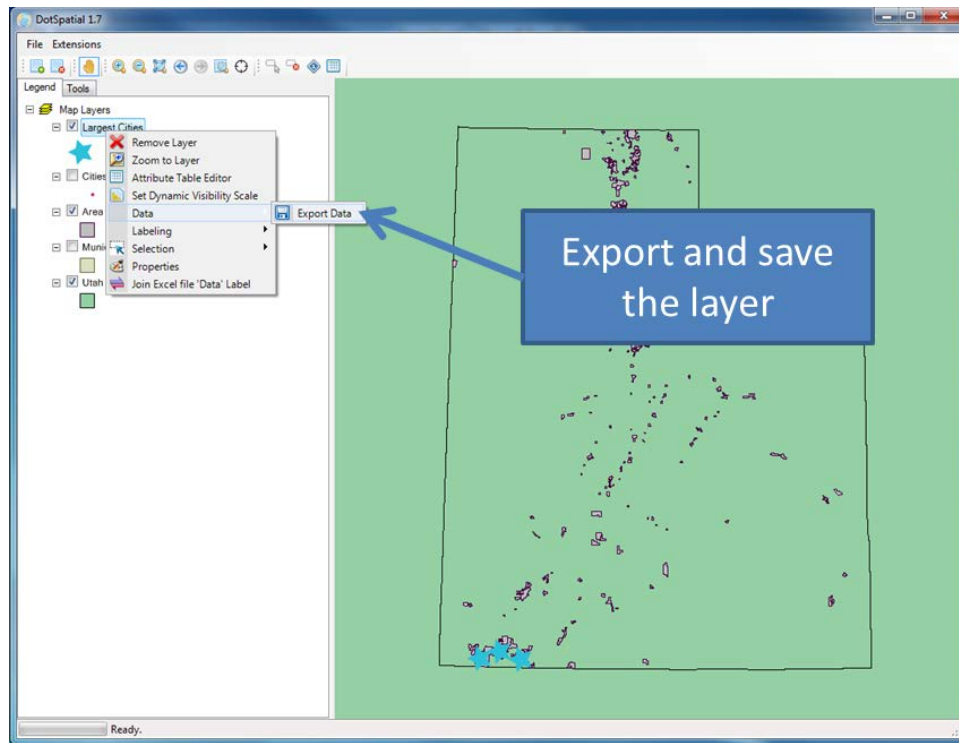


Figure 41: Export Layer

Exporting the layer will save this newly created city layer with just the cities with the largest areas.

1.3.6 Step 6: Repeat with Highway Linear Referencing System Routes

We will limit the size of this shapefile as well. There are a variety of ways to limit the shapefile's size but in this example we will use length. This shape file contains not only the highways but the exit ramps and other short roads that we do not want in our project. To simply our project we will only keep roads greater than 1000 meters.

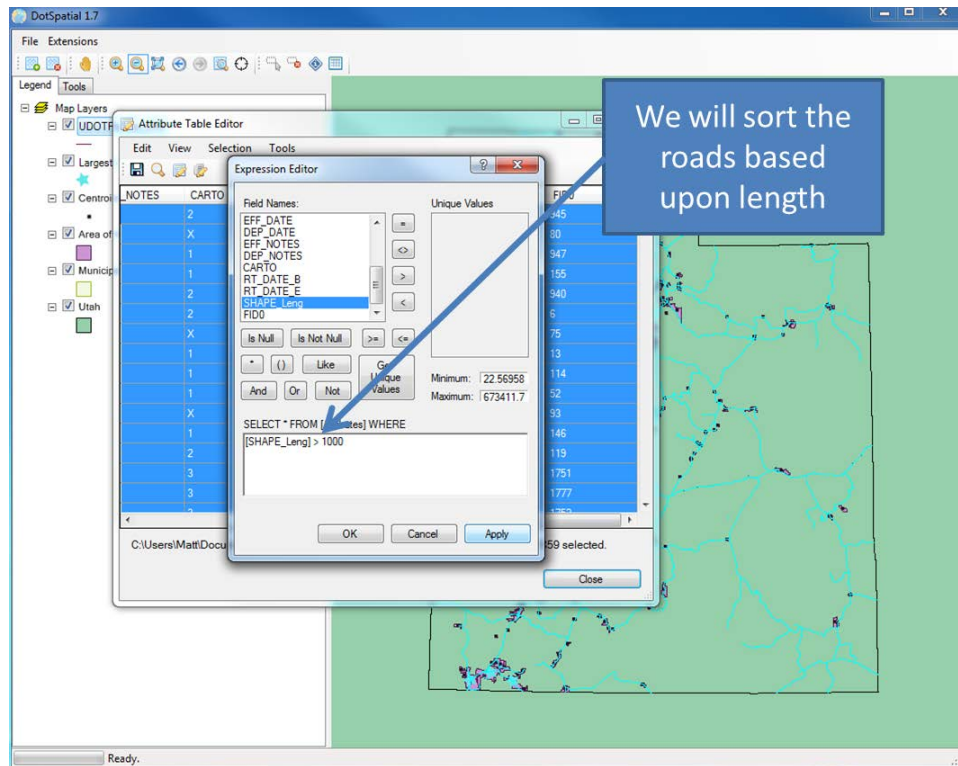


Figure 42: Primary Roads

Make a new layer using the data and export the file. Selecting by attributes is not the only way to narrow down the features that you want to use. First, make sure that the layer you want to interact with is selected in the legend. This will prevent content from other layers from being selected at the same time. Next you can activate the selection tool by using the button in the toolbar indicated in the figure below. Holding down the [Shift] or [Ctrl] keys allows you to select multiple layers at one time. Once you have selected the major polygons, create a new layer the same way by using the create layer from selection option in the legend.

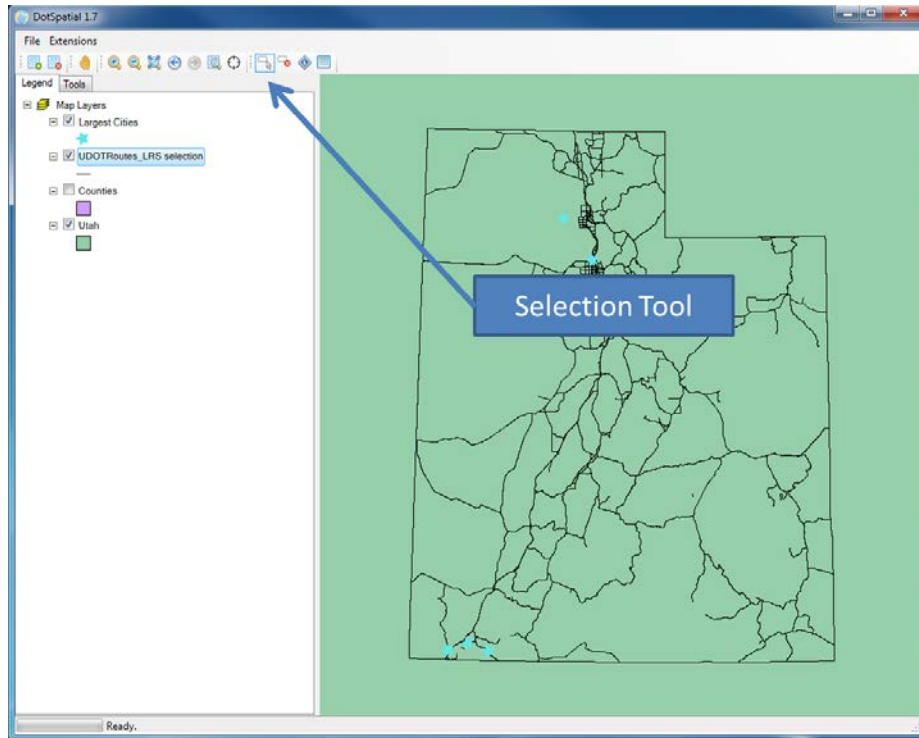


Figure 43: Selection Tool

1.3.7 Step 7: Applying Labels

Add the counties shapefile to the map. In order to finish our map exercise we will add labels to our map for each county.

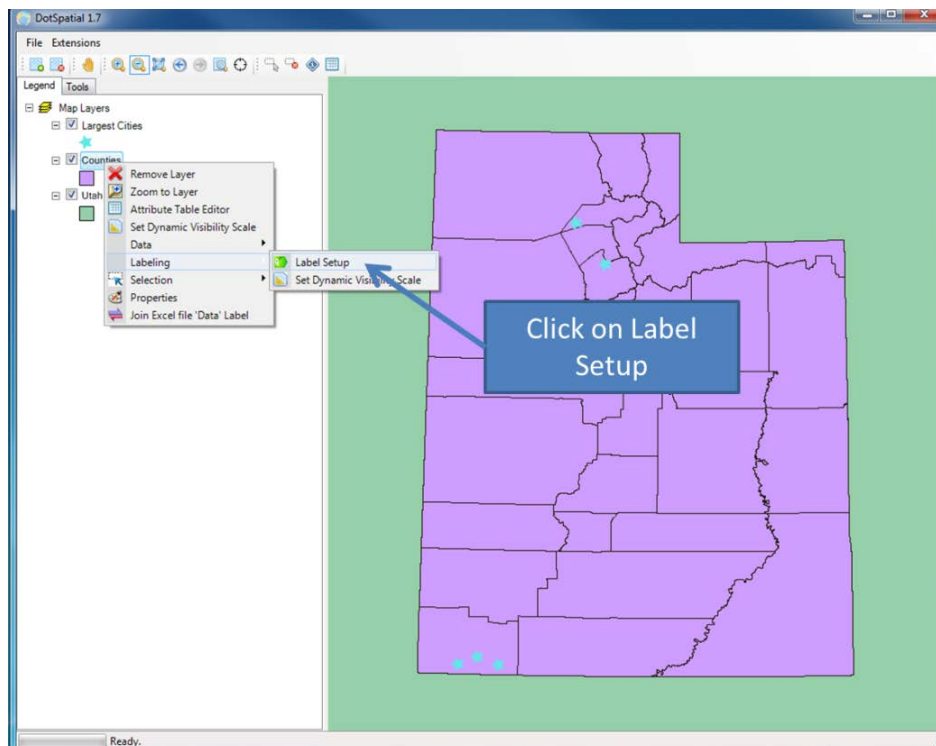


Figure 44: Label Setup

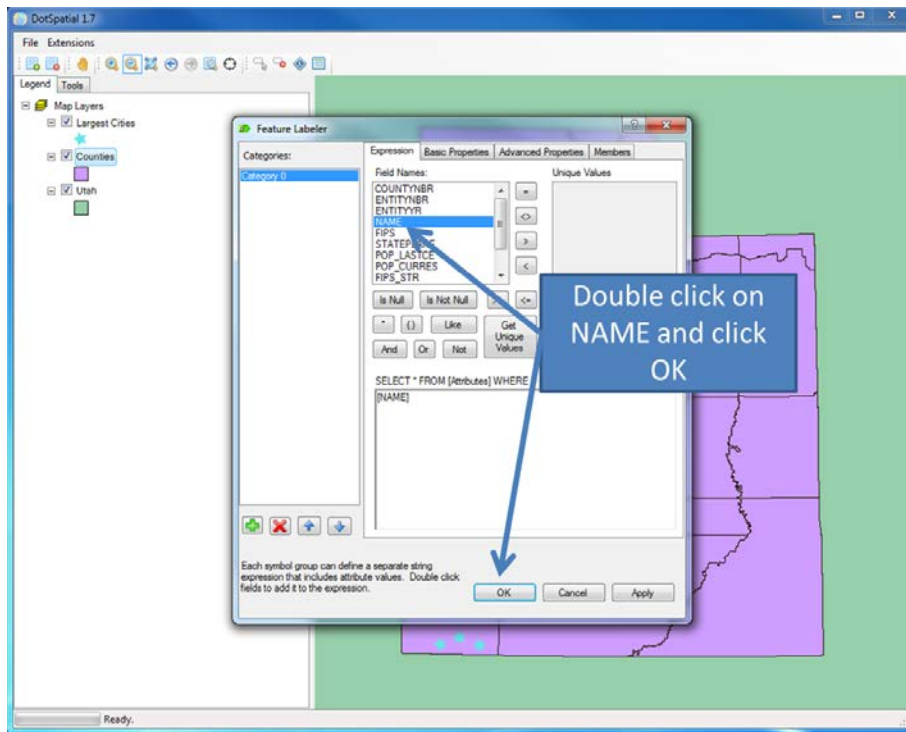


Figure 45: Feature Labeler

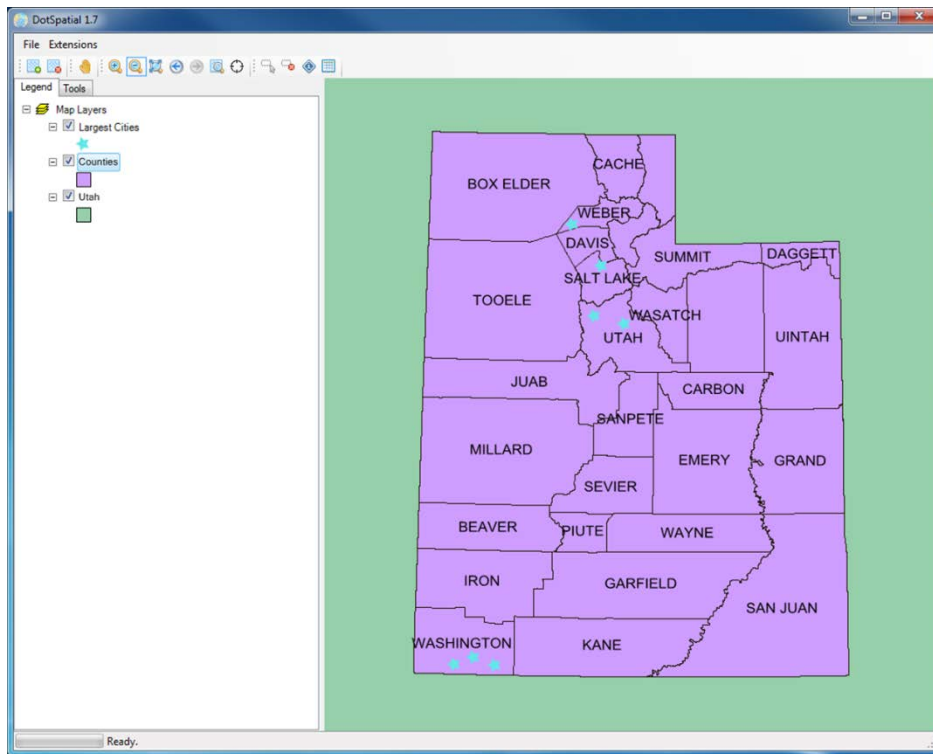


Figure 46: Applied Labels

This is the end of the basic map tutorial. If you have more questions feel free to check the “Discussions”

tab on <http://dotspatial.codeplex.com/discussions>.

1.4. Programmatic Point Symbology

Unlike the previous section, where we were assuming that the reader would perform each step, in this section, we will explore the most basic features of the extremely powerful symbology toolkit provided by DotSpatial, but listed under completely independent objectives. These are subdivided into 5 basic categories: Points, Lines, Polygons, Labels, and Rasters. A comprehensive set of cascading forms launched from the Legend provide a built in system so that users can get started editing symbology right away using the built in components. However, this section is not about mastering the buttons on the dialogs. Rather, this section makes the assumption that you are either writing a plug-in, or else are building your own GIS software using our components. In such a case, you might want to be able to automatically add certain datasets, and control the symbology automatically, behind the scenes.

In previous versions of DotSpatial, setting the color of the seventh point in the shapefile to red was extremely simple, but the trade-off was that anything more complex was not inherently supported in the base program. Instead, developers would have to write their own code to make symbolize the layer based on attributes. DotSpatial introduces thematic symbol classes that on the surface appear to be much more complicated. However, we will show that accessors have been provided to still allow easy access to the simplest steps, but also provide a basic structure that makes symbolizing by attributes much simpler than in previous versions.

To begin this exercise, we will need the datasets created as part of exercise 2. We will also be working with a copy of the Visual Studio project that we created in exercise 1, to hammer in the point that you do not need to be working with the DotSpatial executable, but rather can be working directly with the components in a new project. The first step is to explore adding data to the map programmatically.

1.4.1 Add a Point Layer

Objective:

Add A Point Layer To The Map

```
using DotSpatial.Data;
```

```
IFeatureSet fs = FeatureSet.Open(@"[YourFolder]\DotSpatialDev\shapefiles\Centroids of Municipalities\Centroids.shp");
```

```
IMapFeatureLayer mylayer = map1.Layers.Add(fs);
```

The three lines of code above are all that is needed to programmatically add the cities with centroids shapefile to the map. The first line allows us to use the DotSpatial data library which includes the IFeatureSet. The second line creates an external FeatureSet class. This is useful for opening and working with shapefiles. It also creates an external FeatureSet class and reads the content from the vector portion of the shapefile into memory. The @ symbol tells C# that the line should be read literally, and won't use the \ character as an escape sequence. You can substitute [YourFolder] with the folder containing these exercises on your computer. Finally, the last line adds the data to the map and creates a map feature layer named "mylayer" which we will use in the next section.

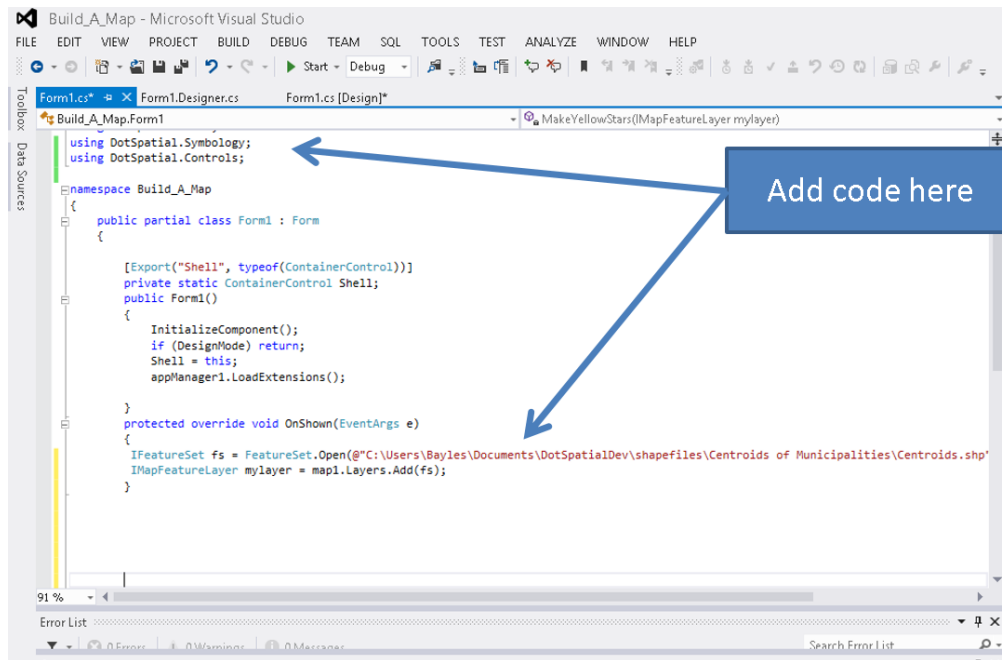


Figure 47: Adding a Shapefile

We can add the lines of code above in the main form by overriding the OnShown method. That way, when the form is shown for the first time, it launches the map. The FeatureSet variable gives us access to all the information stored directly in the shapefile, but organized into feature classes.

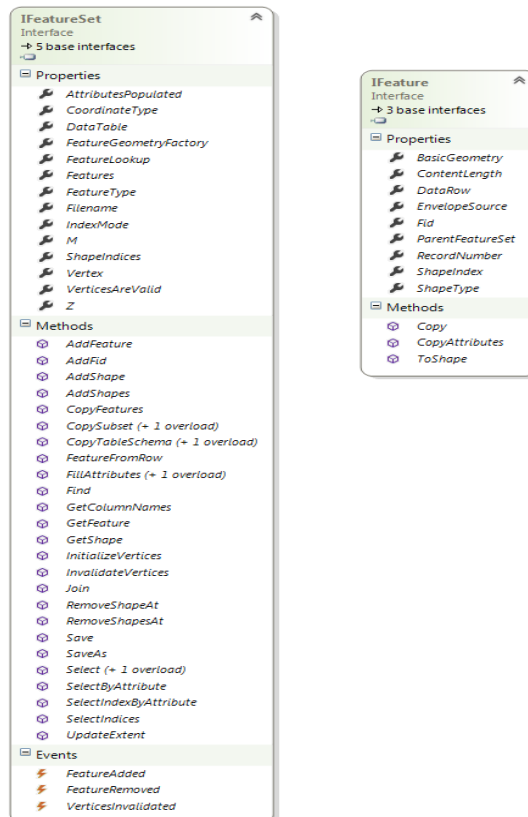


Figure 48: Feature Set and Features

The simplified class diagrams from above give an idea of what kinds of information you can find directly on the FeatureSet class, or in one of the individual Features. The DataTable property returns a standard .Net System.Data.DataTable, filled with all of the attribute information. This table is also used by the SelectByAttribute expression. The Envelope is the geographic bounding box for the entire set of features. The Features property is the list of features themselves. This is enumerable, so you can cycle through the list and inspect each of the members. The FeatureType simply tells whether or not the FeatureSet contains points, lines or polygons.

The other significant vector data class shown here is the Feature. The BasicGeometry class lists all of the vertices, organized according to OGC geometric structures, such as Points, LineStrings, Polygons, and MultiGeometries of the various types. Because we were interested in making extensible data providers, we did not require these basic geometric classes to support all mathematical overlays and related operations that can be found in the various topology suites. Instead, the role of the BasicGeometry is to provide the data only interface.

We did not overlook the possibility of wanting to perform, say, an intersect operation with another feature. Instead of building the method directly into the feature class, we have instead build extension methods so that from a programmatic viewpoint, it will look like any IFeature will be able to perform Intersection calculations.

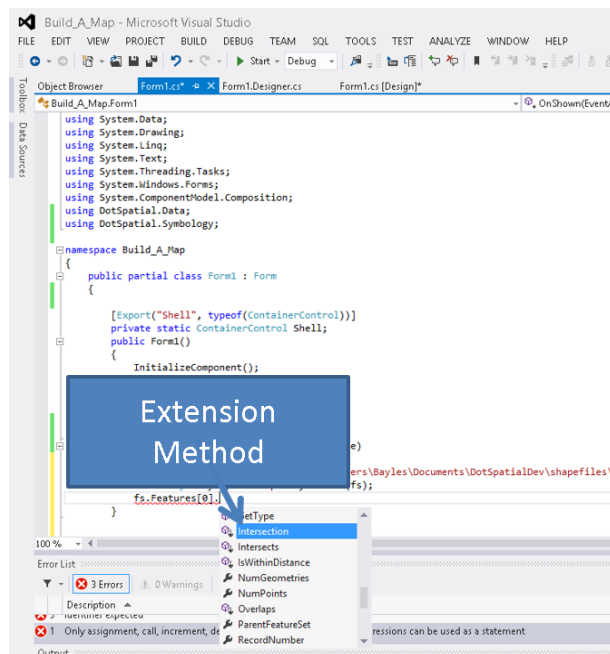


Figure 49: Extension Method

Typing a period after a class in .Net will display an automatic list of options. Continuing to type normally filters this list of options, so that you can very rapidly narrow the displayed items and reduce the chance for spelling mistakes. It also gives you an instant browse window to explore the options on a particular class. This auto-completion tool is referred to as Microsoft Intellisense. The exact

appearance of this function will depend on the version of visual studio, as well as whether or not you have any extensions like Re-Sharper loaded. If XML comments have been generated for the project, (which they have been for DotSpatial) you will not only see the methods, properties and events available, but you will also see help for each of these methods that extends to the right.

Methods are identified by having a purple box. Extension methods are represented by a purple box with a blue arrow to the right of that box. Instead of having the programming code built into the class, the code is actually separate, in an external static method. Using this technology, it becomes easy to associate a behavior like Intersects directly with the feature, but without every external data provider having to rewrite the intersection code itself.

1.4.2 Simple Symbols

Objective:

Make Yellow Stars

```
using DotSpatial.Symbology;  
using DotSpatial.Controls;
```

```
private void MakeYellowStars(IMapFeatureLayer mylayer)  
{  
    mylayer.Symbolizer = new PointSymbolizer(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 16);  
}
```

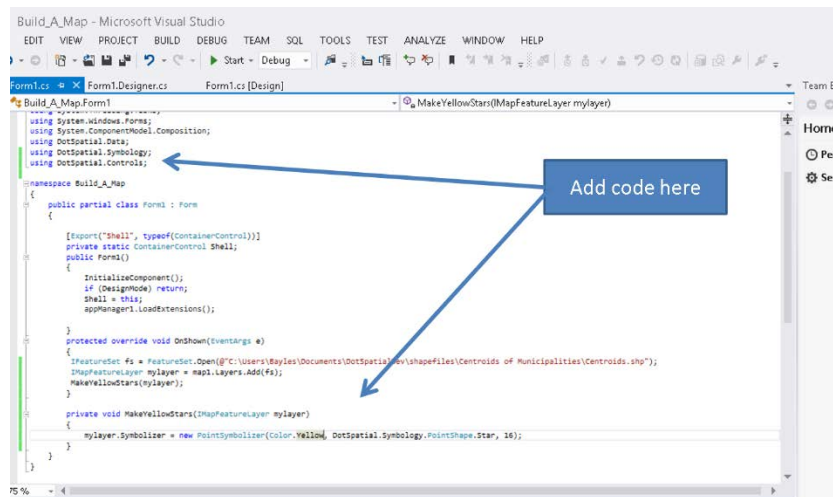


Figure 50: Yellow Stars Code

Because we know in advance that we are working with points, we don't have to work directly with the existing classes, or use casting. We can simply use the constructor. If we pass mylayer from the earlier code into the method above, we will automatically create the outpoints shown in the following image.

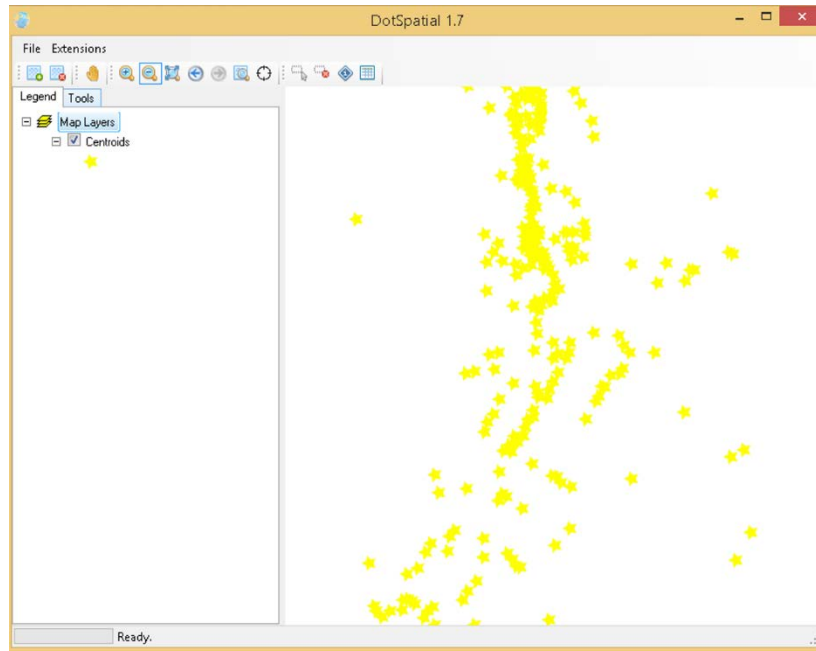


Figure 51: Yellow Stars

One thing that you might notice is that the borders of the stars are hard to see because we only specified one color, and that color was the fill color. In order to give the stars black outlines, we need to call a slightly different method.

```
mylayer.Symbolizer = new PointSymbolizer(Color.Yellow, PointShapes.Star, 16);  
mylayer.Symbolizer.SetOutline(Color.Black, 1);
```

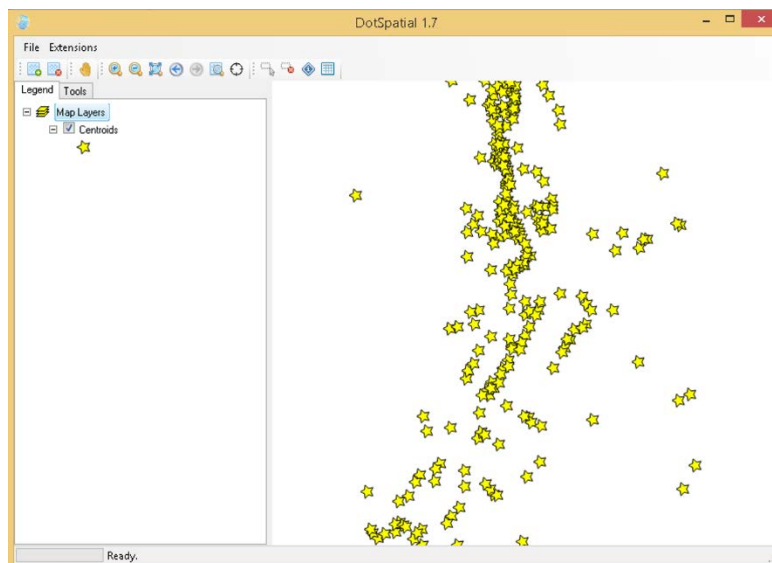


Figure 52: Yellow Stars with Outlines

You will notice that the layer does not control these characteristics directly. Instead, it uses a class called a Symbolizer. Symbolizers contain all of the descriptive characteristics necessary to draw something. They have a few simple accessors that allow us to work with the simple situations

like the one listed above. In this situation, we are not worried about a scheme, or complex symbols that have multiple layers. A method like SetOutline may or may not work as expected in every case, since some types of symbols do not even support outlines. However, if we inspect the parameters that we can control above, we already have the basic symbology options that were provided in previous versions of DotSpatial.

1.4.3 Character Symbols

Objective:

Use Character Symbols

In addition to the basic symbols, DotSpatial also provides access to using characters as a symbol. This is a very powerful system since character glyphs are vectors, and therefore scalable. They look good even when printing to a large region at high resolution. It is also incredibly versatile. Not only can you use pre-existing symbol fonts (like wingdings) that are on your computer, there are open source fonts that provide GIS symbols. One helpful site that has lots of GIS symbol fonts that can be downloaded for free is found here: <http://www.mapsymbols.com>. For this exercise, we are downloading and unzipping the military true type fonts from the site. Downloading and unzipping the file produces a file with the extension .ttf, which is a true type font. The next step is to find the Fonts option in the Control Panel.

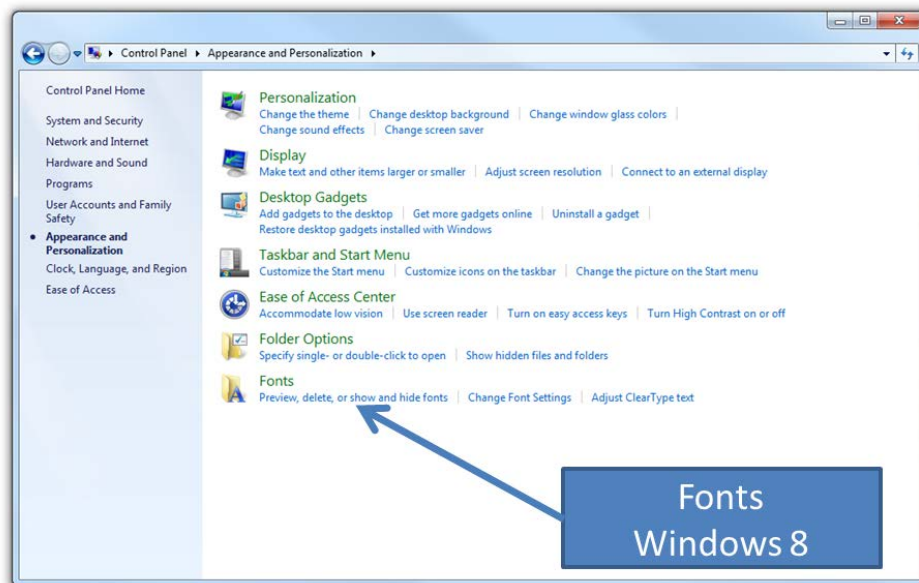


Figure 53: Fonts in the Control Panel

Clicking on this folder will open the folder showing all of the currently installed true type fonts. You will

need to copy and paste the new font into this folder and it will automatically install then new font for you. In this case we are using the Military.ttf file.

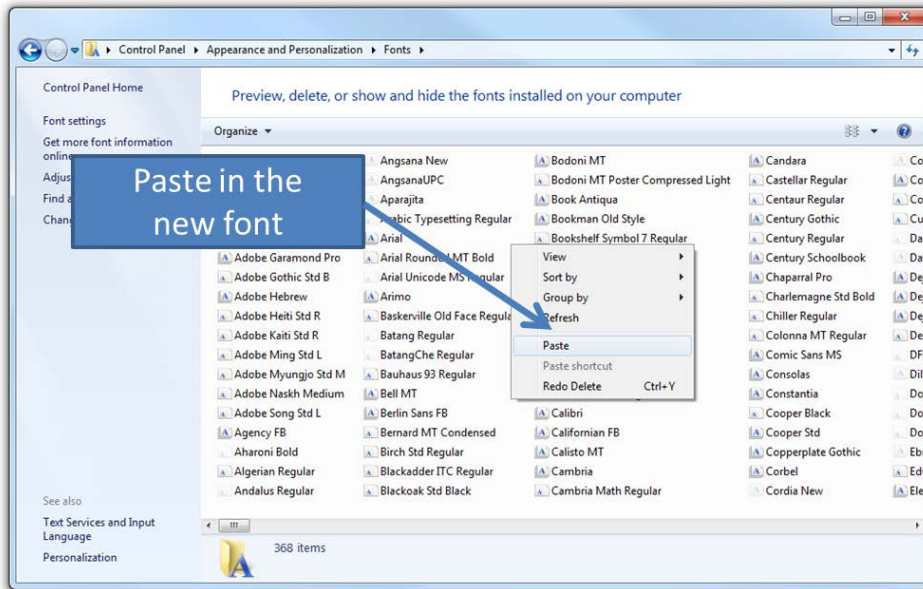


Figure 54: Right Click to Paste New Font

We can verify that the new font is available directly by running DotSpatial 1.7, or our newly created program, adding a point layer, and then symbolizing with character symbols. To use character symbols, double click on the symbol in the legend. This will open the Point Symbolizer Dialog. There is a Combo-box named "Symbol Type" which enables the user to choose a new symbol type. In this case, we want to choose Character.

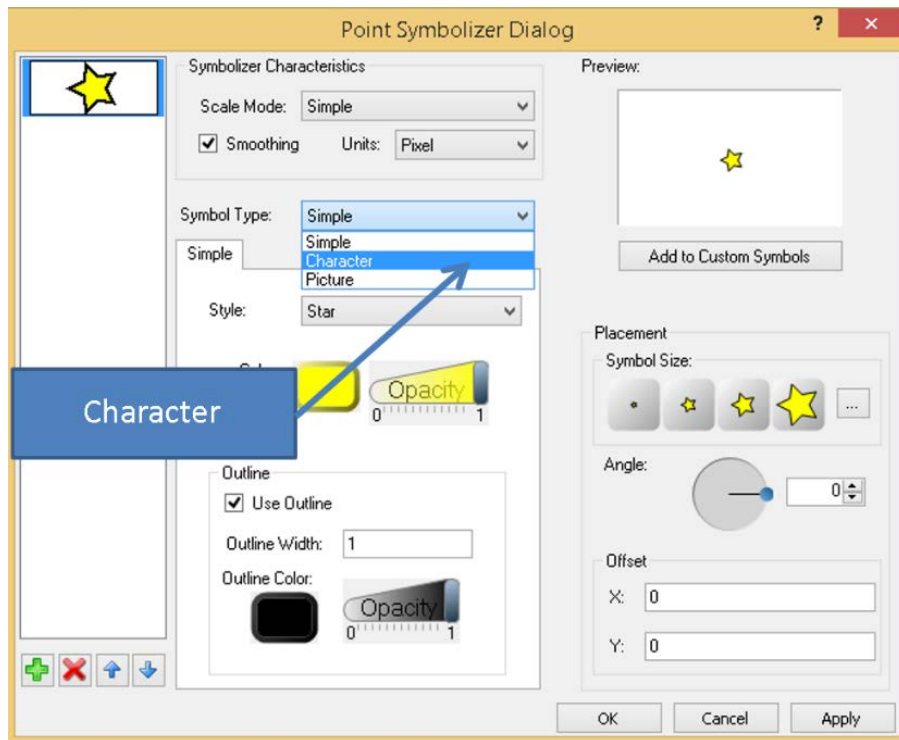


Figure 55: Switch to Characters

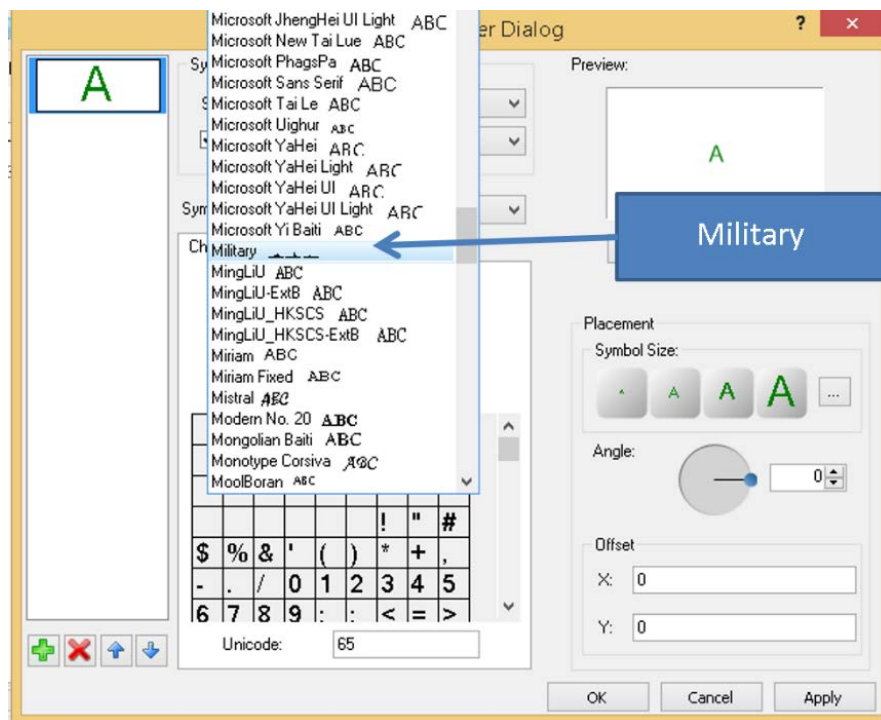


Figure 56: Choose Military

Once you select military, the icons listed below in the character selection drop-down should be replaced with the new military symbols that we have just downloaded. Because many GIS

systems use true type fonts, it is possible for DotSpatial to show font types from pre-created professional font sets. Having verified that we have successfully enabled the software to use the new military font type, we will now attempt to use one of these symbols programmatically. To draw planes, we will choose the character M, which draws a character of a plane, and specify the use of the Military font name. We can also specify that they will be blue and will have a point size of 16.

```
mylayer.Symbolizer = new PointSymbolizer('M', "Military", Color.Blue, 16);
```

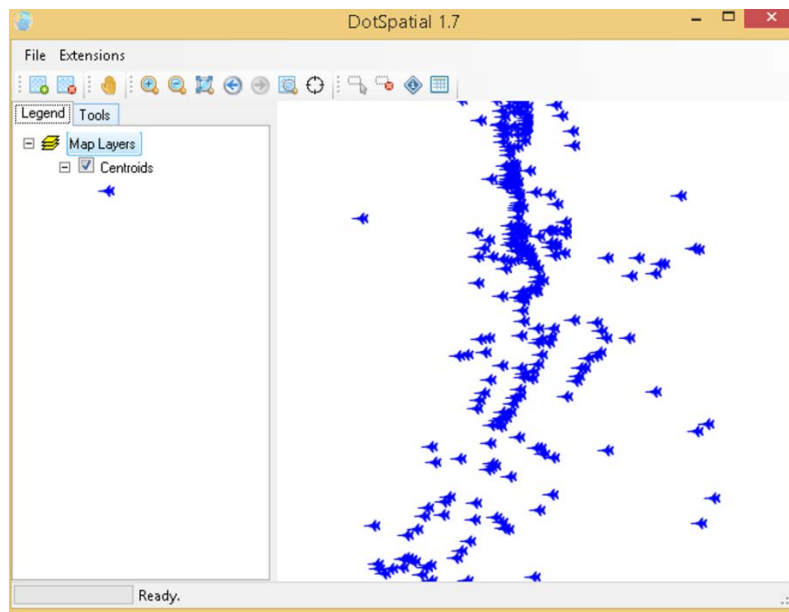


Figure 57: Military Plane Characters

As a side note, the Military font is slightly abnormal in that it has a huge amount of white space at the bottom of each glyph. Attempting to center the font vertically will cause problems because of all the whitespace. As a result, an added line of code catches this possibility and uses the width for centering instead. Since these symbols may not be exactly square, this may place the military symbols slightly off center. However, more professionally created symbols will be centered correctly since they will have a realistic height value that can be used for centering. It is also possible to create custom glyphs for use as point symbols using various font editor software packages.

1.4.4 Image Symbols

Objective:

Use Image Symbols

Sometimes, it simply isn't possible to work with fonts, however, and an image is preferable. In such a case, you can use almost any kind of image. Remember that if you have lots of points, it is better to be working with smaller images. As an example, you can download this wiki-media tiger icon to work with:

<http://commons.wikimedia.org/>

To use this symbol programmatically, first download the image to a file. Once you have saved the image file, you reference it using either standard file loading methods for images, or else you can embed the file as a resource to be used. We will look at embedding the image as a resource. First, from solution explorer, add a resource file named Images.

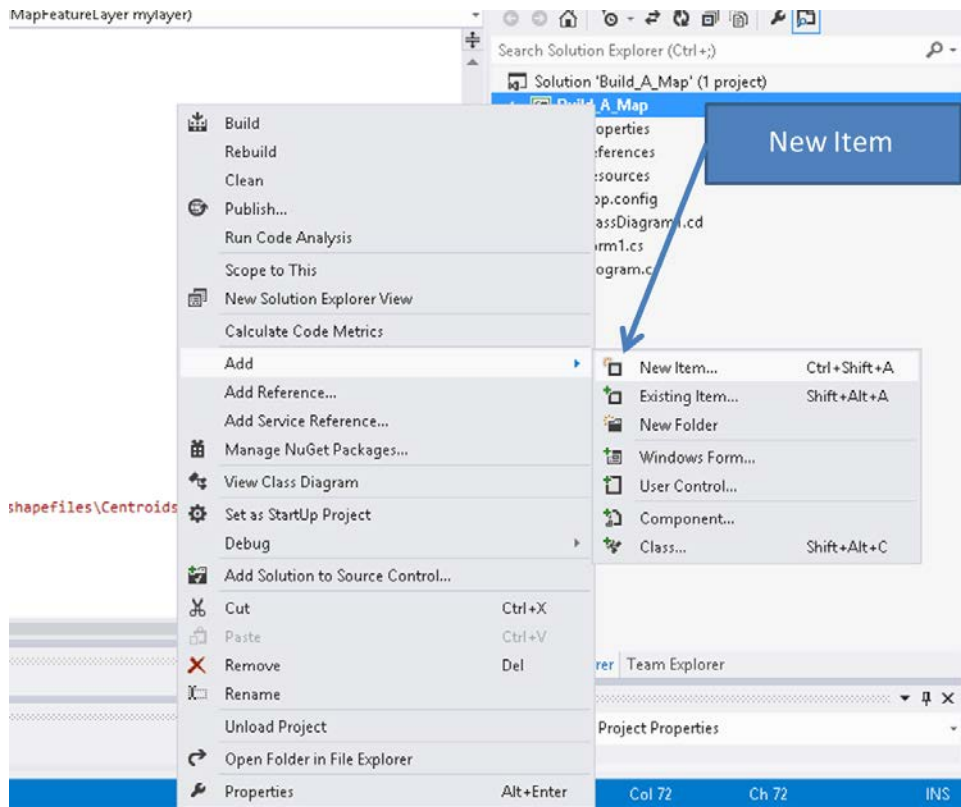


Figure 58: Add a New Item

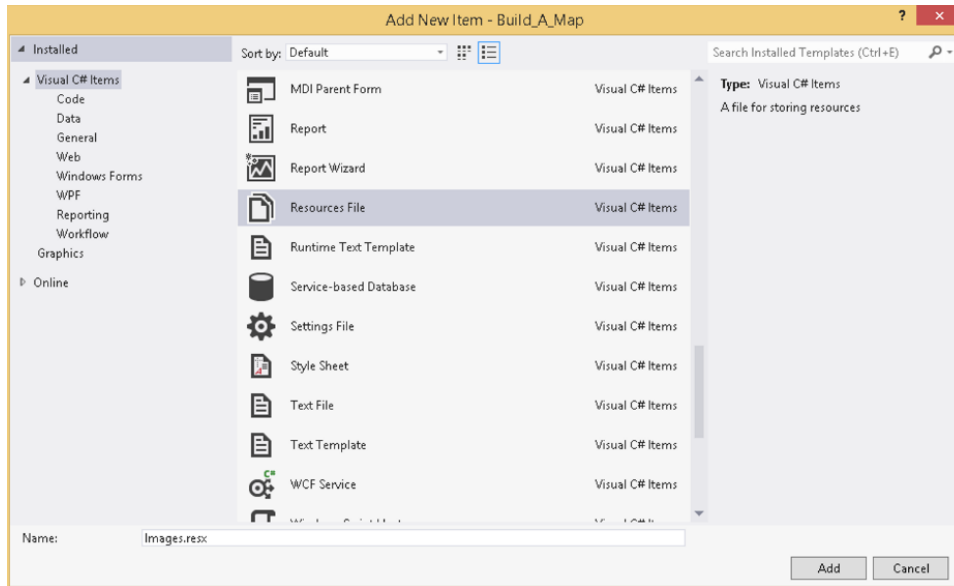


Figure 59: Images.resx Resource

Choose Resources File from the available Templates, then name the resource file Images, and click Add. From the solution explorer, simply double click the newly added Images resource file to open it. Adding it for the first time should automatically open the resource file as a tab. Under the Add Resource option in the toolbar just below the Images.resx tab, choose “Add Existing File...”

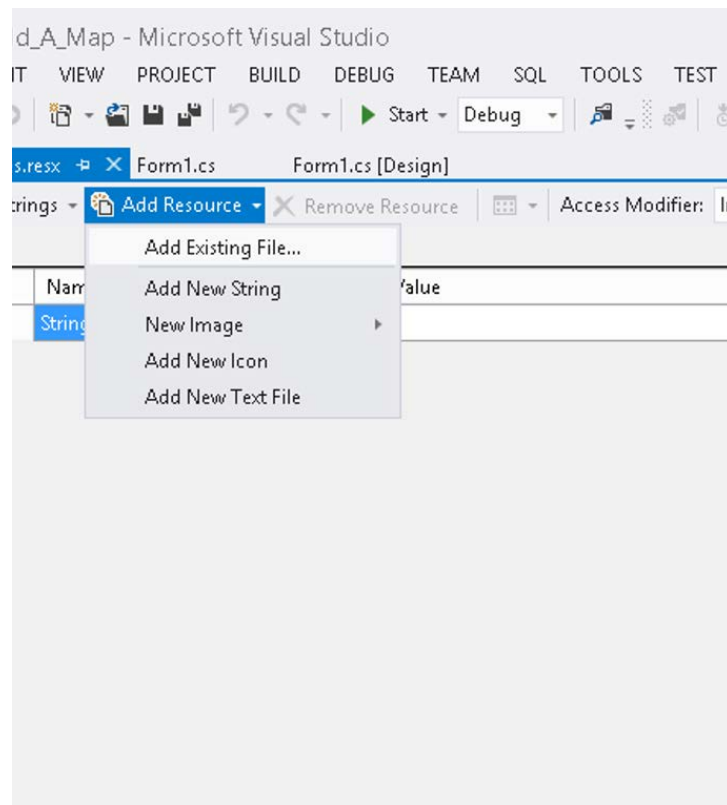


Figure 60: Add an Existing File

Then simply browse to the file we just downloaded, and add it to the resource file. To make it easier to find, we can rename the file “Tiger”.

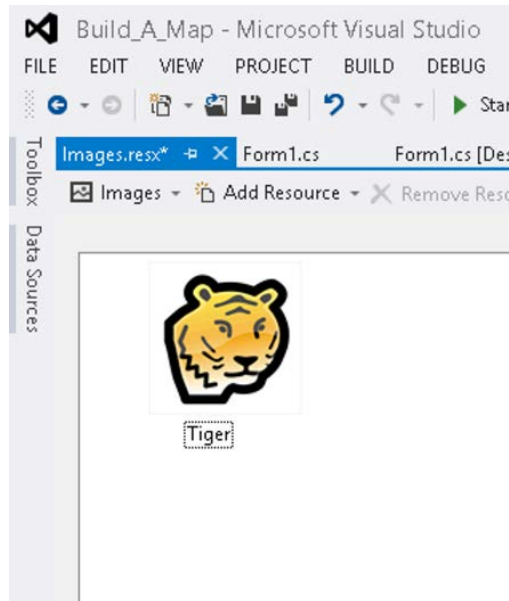


Figure 61: Rename to Tiger

Now, we can programmatically reference this image any time using the `Images.Tiger` reference. Be sure to build the project after adding the image to the resource file, or else the Intellisense will not show the Tiger image yet. Adding the image to a resource file sets up the framework for the following line of code where we will specify to use the Tiger image for the point symbology:

```
mylayer.Symbolizer = new PointSymbolizer(Images.Tiger, 48);
```

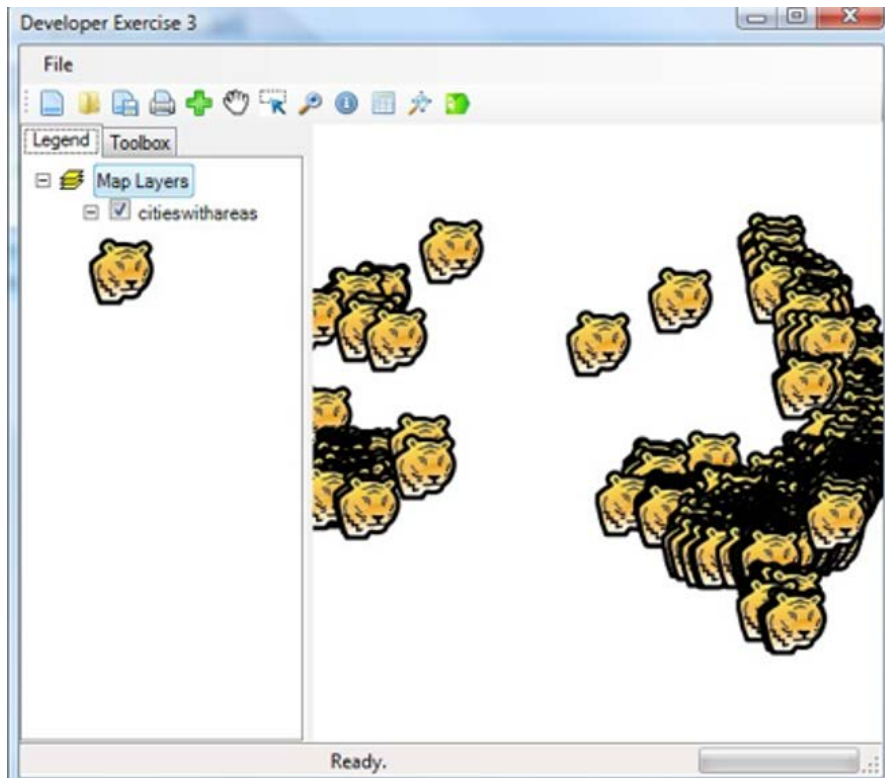


Figure 62: Tiger Images

1.4.5 Point Categories

Objective:

Use Point Symbol Categories

For the next topic, we will introduce the concept of casting. In many cases, methods or properties are shared between many different types of classes. When that is true, the interface may provide the shared base class, instead of the class type that is actually being used. For instance, if you are working with points, the Symbolizer that is being used should be a PointSymbolizer. This has a shared base class with the FeatureSymbolizer.

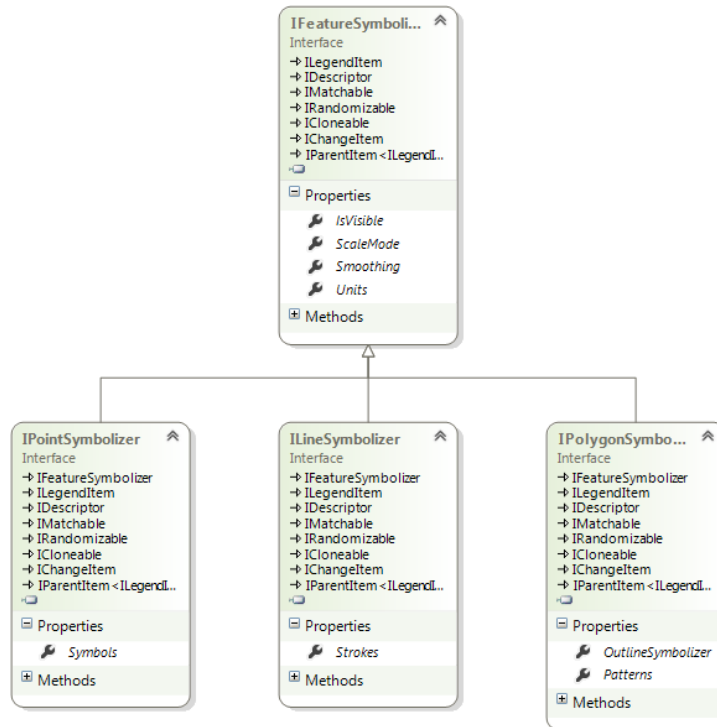
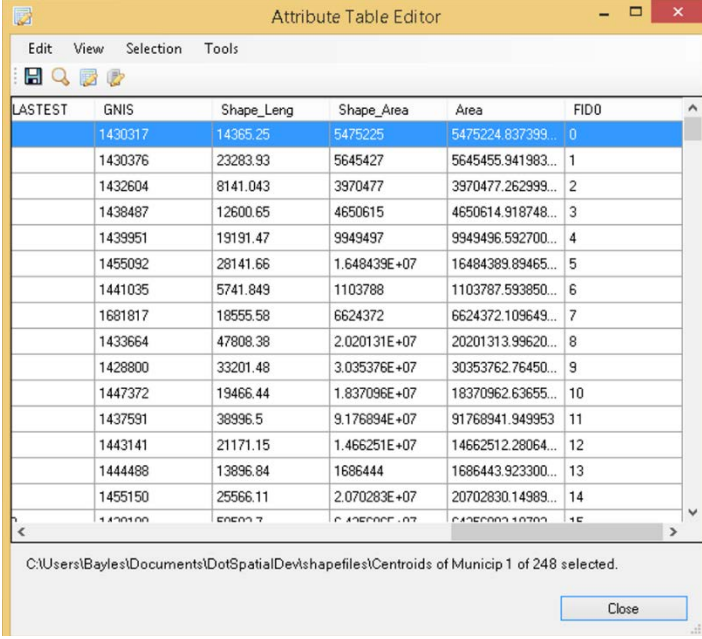


Figure 63: Symbolizer Class Diagram

To illustrate inheritance, the class diagram above shows the three main feature symbolizers, one for points, one for lines, and one for polygons. What is most important here is that there are some characteristics that will be shared. Properties, methods and events on the FeatureSymbolizer class will also appear on each of the specialized classes. Meanwhile, each individual type has a collection of classes that actually do the drawing, but these classes are different depending on the class. In the same way, we can set up categories, but it works much more easily if we know what kind of feature layer we are working with. When working with categories, schemes, and so-on, knowing that we are working with a point layer is the difference between having to cast every single object every time, and only having to cast the feature layer once.

We will be symbolizing based on the Area field. The areas were calculated from the polygons in exercise 2, so we can be assured that the cities with areas shapefile that we are adding has an Area field, and looking at the table below, we can see that a reasonable cutoff for picking the largest cities might be

1e+08 meters.



LASTEST	GNIS	Shape_Leng	Shape_Area	Area	FID0
1430317	14365.25	5475225	5475224.837399...	0	
1430376	23283.93	5645427	5645455.941983...	1	
1432604	8141.043	3970477	3970477.262999...	2	
1438487	12600.65	4650615	4650614.918748...	3	
1439951	19191.47	9949497	9949496.592700...	4	
1455092	28141.66	1.648439E+07	16484389.89465...	5	
1441035	5741.849	1103788	1103787.593850...	6	
1681817	18555.58	6624372	6624372.109649...	7	
1433664	47808.38	2.020131E+07	20201313.99620...	8	
1428800	33201.48	3.035376E+07	30353762.76450...	9	
1447372	19466.44	1.837096E+07	18370962.63655...	10	
1437591	38996.5	9.176894E+07	91768941.949953	11	
1443141	21171.15	1.466251E+07	14662512.28064...	12	
1444488	13896.84	1686444	1686443.923300...	13	
1455150	25566.11	2.070283E+07	20702830.14989...	14	
1430100	59503.7	6.425000E+07	64250000.10700...	15	

Figure 64: Large Area Cities

The source code that we are going to use has several parts to it. First, we are going to cast the layer to a `MapPointLayer` so that we know we are working with point data. After that, we create two separate categories, using filter expressions to separate what is drawn by each category. Finally, we add the new scheme as the layer's symbology. When the map is drawn, it will automatically show the different scheme types in the legend using whatever we specify here as the legend text.

```
IFeatureSet fs = FeatureSet.Open(@"C:\Users\Bayles\Documents\DotSpatialDev\shapefiles\Centroids of Municipalities\Centroids.shp");
IMapFeatureLayer mylayer = map1.Layers.Add(fs);
IMapPointLayer myPointLayer = mylayer as IMapPointLayer;
if (myPointLayer == null) return;
PointScheme myScheme = new PointScheme();
myScheme.Categories.Clear();
PointCategory smallSize = new PointCategory(Color.Blue, DotSpatial.Symbology.PointShape.Rectangle, 4);
smallSize.FilterExpression = "[Area] < 1e+08";
smallSize.LegendText = "Small Cities";
myScheme.AddCategory(smallSize);
PointCategory largeSize = new PointCategory(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 16);
largeSize.FilterExpression = "[Area] >= 1e+08";
largeSize.LegendText = "Large Cities";
largeSize.Symbolizer.SetOutline(Color.Black, 1);
myScheme.AddCategory(largeSize);
myPointLayer.Symbology = myScheme;
```

```
Images.resx Form1.cs* -> Form1.cs [Design]*
Build_A_Map.Form1 OnShown(EventArgs e)

[Export("Shell", typeof(ContainerControl))]
private static ContainerControl Shell;
public Form1()
{
    InitializeComponent();
    if (DesignMode) return;
    Shell = this;
    appManager1.LoadExtensions();
}
protected override void OnShown(EventArgs e)
{
    IFeatureSet fs = FeatureSet.Open(@"C:\Users\Bayles\Documents\DotSpatialDev\shapefiles\Centroids of Municipalities\Centroids.shp");
    IMapFeatureLayer mylayer = map1.Layers.Add(fs);
    IMapPointLayer myPointLayer = mylayer as IMapPointLayer;
    if (myPointLayer == null) return;
    PointScheme myScheme = new PointScheme();
    myScheme.Categories.Clear();
    PointCategory smallSize = new PointCategory(Color.Blue, DotSpatial.Symbology.PointShape.Rectangle, 4);
    smallSize.FilterExpression = "[Area] < 1e+08";
    smallSize.LegendText = "Small Cities"; myScheme.AddCategory(smallSize);
    PointCategory largeSize = new PointCategory(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 16);
    largeSize.FilterExpression = "[Area] >= 1e+08";
    largeSize.LegendText = "Large Cities";
    largeSize.Symbolizer.SetOutline(Color.Black, 1);
    myScheme.AddCategory(largeSize);
    myPointLayer.Symbol = myScheme;
}
```

Figure 65: Example Code

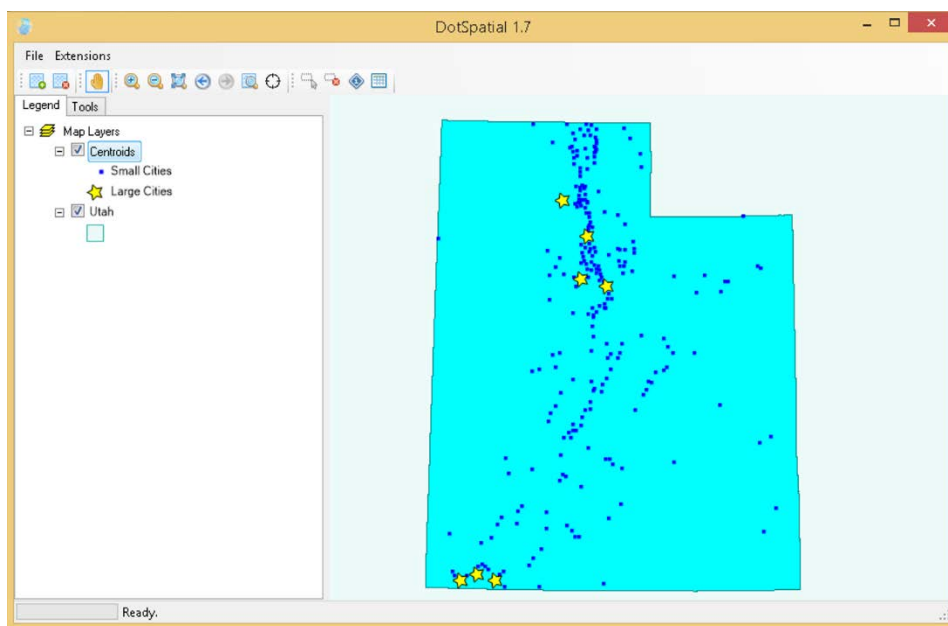


Figure 66: City Categories by Area

The square brackets in the filter expression are optional, but recommended to help clarify fieldnames in the expression. What is significant here is that we did not have to write code to actually loop through all of the city shapes, test the area attribute programmatically, and then assign a symbol scheme based on the character. Instead, we simply allow the built in expression parsing to take over and handle the drawing for us. This allows for programmers to work with the objects in a way that directly mimics how users work with the symbology controls. And just like the layer dialog controls allow you to specify schemes; those schemes can also be controlled programmatically.

```

IMapPointLayer myPointLayer = mylayer as IMapPointLayer;
if (myPointLayer == null) return;
PointScheme myScheme = new PointScheme();
myScheme.Categories.Clear();
myScheme.EditorSettings.ClassificationType = ClassificationType.Quantities;
myScheme.EditorSettings.IntervalMethod = IntervalMethod.Quantile;
myScheme.EditorSettings.IntervalSnapMethod = IntervalSnapMethod.Rounding;
myScheme.EditorSettings.IntervalRoundingDigits = 5;
myScheme.EditorSettings.TemplateSymbolizer =new PointSymbolizer(Color.Yellow,
    DotSpatial.Symbology.PointShape.Star, 16);
myScheme.EditorSettings.FieldName = "Area";
myScheme.CreateCategories(mylayer.DataSet.DataTable);
myPointLayer.Symbology = myScheme;

```

```

public Form1()
{
    InitializeComponent();
    if (DesignMode) return;
    Shell = this;
    appManager1.LoadExtensions();
}

protected override void OnShown(EventArgs e)
{
    IFeatureSet fs = FeatureSet.Open(@"C:\Users\Bayles\Documents\DotSpatialDev\shapefiles\Centroids of Municipalities\Centroids.shp");
    IMapFeatureLayer mylayer = map1.Layers.Add(fs);
    IMapPointLayer myPointLayer = mylayer as IMapPointLayer;
    if (myPointLayer == null) return;
    PointScheme myScheme = new PointScheme();
    myScheme.Categories.Clear();
    myScheme.EditorSettings.ClassificationType = ClassificationType.Quantities;
    myScheme.EditorSettings.IntervalMethod = IntervalMethod.Quantile;
    myScheme.EditorSettings.IntervalSnapMethod = IntervalSnapMethod.Rounding;
    myScheme.EditorSettings.IntervalRoundingDigits = 5;
    myScheme.EditorSettings.TemplateSymbolizer =new PointSymbolizer(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 16);
    myScheme.EditorSettings.FieldName = "Area";
    myScheme.CreateCategories(mylayer.DataSet.DataTable);
    myPointLayer.Symbology = myScheme;
}

```

Figure 67: Example Code

This will be the last figure to include a snap shot of Visual Studio, as the implementation of each block of code follows the same format.

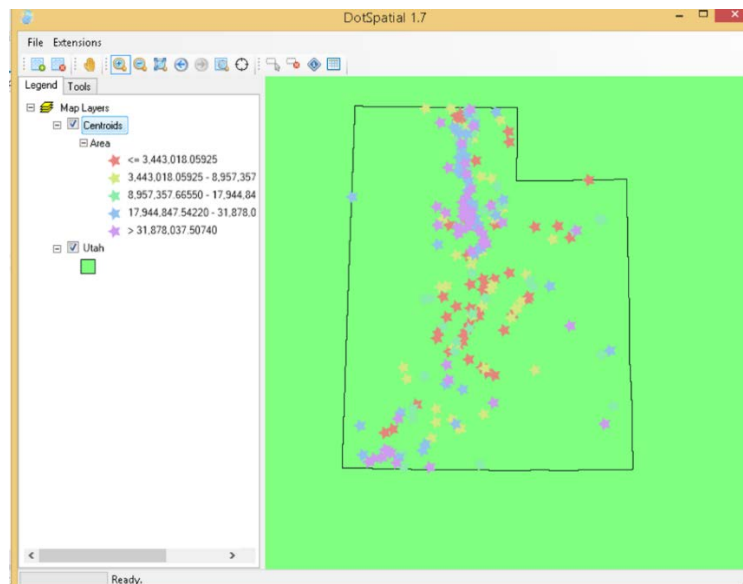


Figure 68: Quantile Area Categories

There are a large number of settings that can be controlled directly using the PointScheme. In this

illustration the classification type is quantities, but this can also be UniqueValues or custom. The categories can always be edited programmatically after they are created, but this simply controls what will happen when the CreateCategories method is ultimately called. The interval snap methods include none, rounding, significant figures, and snapping to the nearest value. These can help the appearance of the categories in the legend, but it can also cause trouble. With Significant figures, the IntervalRoundingDigits controls the number of significant figures instead. One property is deceptive in its power. The TemplateSymbolizer property allows you to control the basic appearance of the categories for any property that is not being controlled by either the size or color ramping. For example, if we wanted to add black borders to the stars above, we would simply add that to the template symbolizer. In this case we chose to make them appear as stars and controlled them to have equal sizes since UseSizeRange defaults to false, but UseColorRange defaults to true.

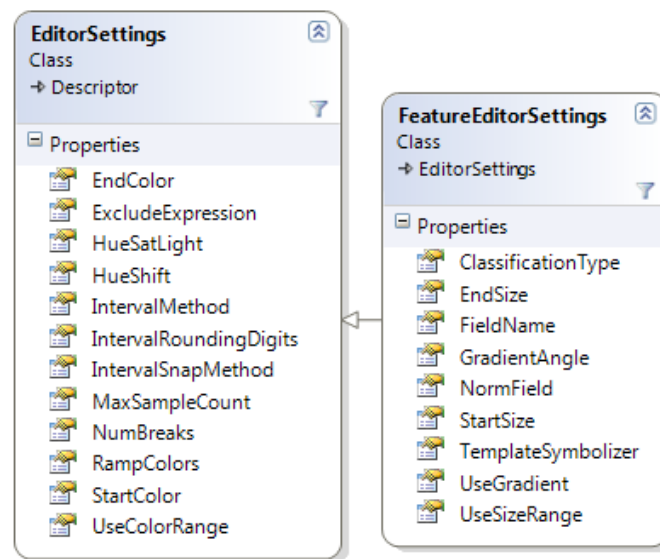


Figure 69: Available Feature Editor Settings

The settings shown in the exercise above represent a small taste of the scheme options that are programmatically available. You can also control the color range, whether or not the colors should be ramped or randomly created, a normalization field, an exclusion expression to eliminate outliers and in the case of polygons, a consistently applied gradient.

1.4.6 Compound Symbols

Objective:

Yellow stars in a Blue Circle

One of the new additions to how symbols work is that you are no longer restricted to representing things using a single symbol. Complex symbols can be created, simply by adding symbols to the Symbolizer.Symbols list. There are three basic kinds of symbols, Simple, Character and Image based. These have some common characteristics, like the Angle, Offset and Size, which are stored on the base class. In the derived classes, the characteristics that are specific to the sub-class control those

aspects of symbology. For creating new symbols, the Subclass can be used. For working with individual symbols in the collection, you may need to test what type of symbol you are working with before you will be able to control its properties.

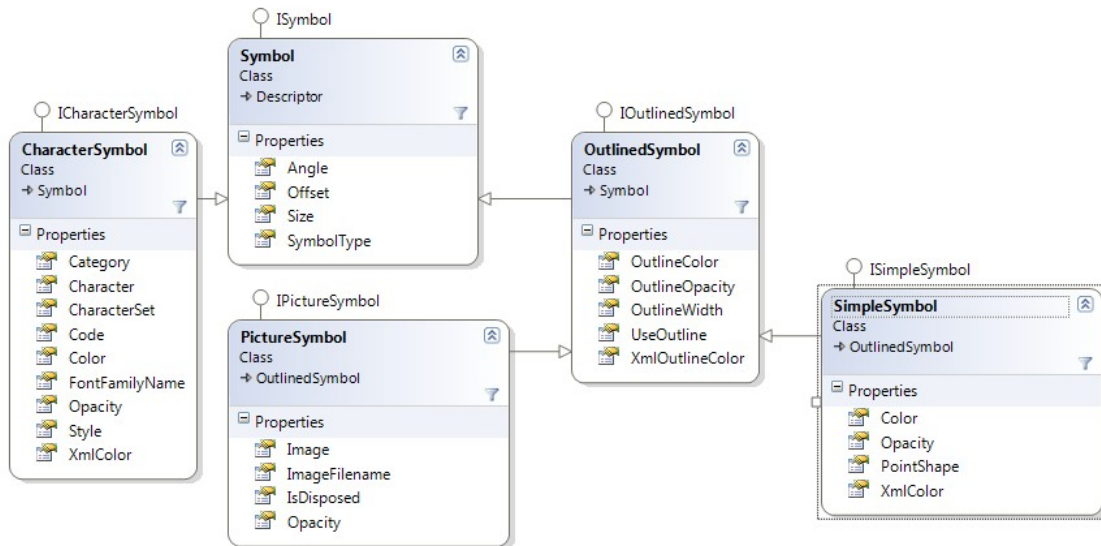


Figure 70: Point Symbol Class Diagram

The class diagram above shows the organization of the individual symbols.

```

PointSymbolizer myPointSymbolizer = new PointSymbolizer(Color.Blue, DotSpatial.Symbology.PointShape.Ellipse, 16);
myPointSymbolizer.Symbols.Add(new SimpleSymbol(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 10));
mylayer.Symbolizer = myPointSymbolizer;

```

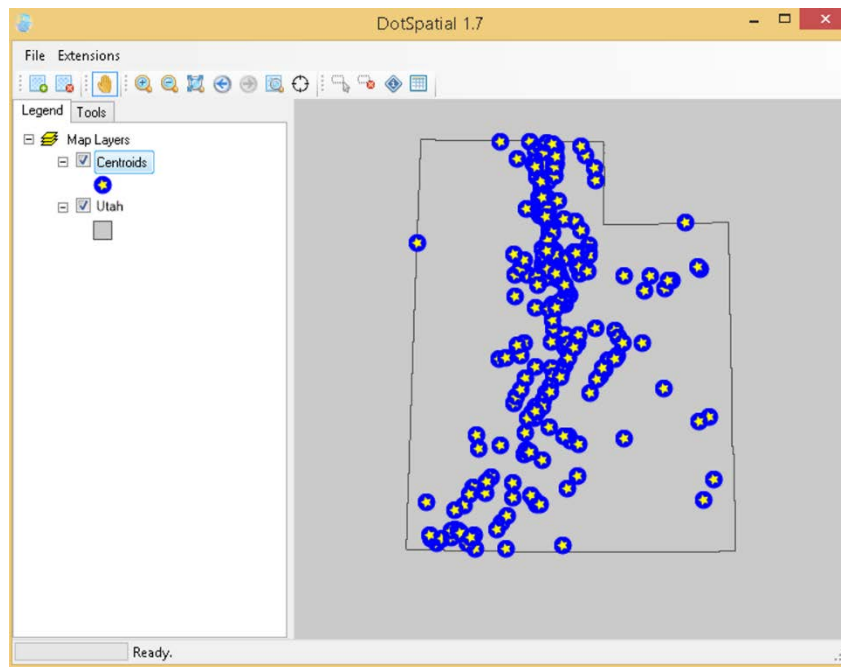


Figure 71: Blue Circles with Yellow Stars

1.5 Programmatic Line Symbology

1.5.1 Adding Line Layers

Objective:

Add Line Layer

Line layers operate according to the same rules as points for the most part, except that instead of individual symbols, we can have individual strokes. The default symbology is to have a single line layer of a random color that is one pixel wide.

```
IFeatureSet fs = FeatureSet.Open(@"C:\[YourFolder]\DotSpatialDev\shapefiles\UDOTRoutes_LRS\UDOTRoutes_LRS.shp");  
IMapFeatureLayer mylayer = map1.Layers.Add(fs);
```

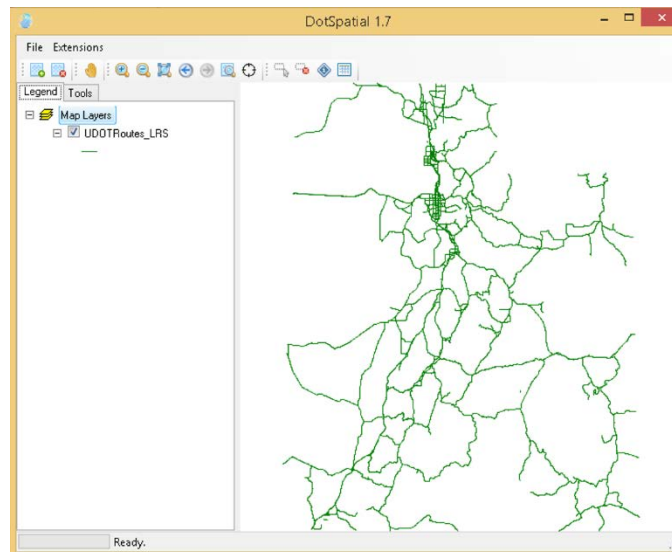


Figure 72: Add Line Layer

1.5.2. Simple Line symbols

Objective:

Brown Roads

```
private void BrownRoads(IMapFeatureLayer mylayer)  
{  
    mylayer.Symbolizer = new LineSymbolizer(Color.Brown, 1);  
}
```

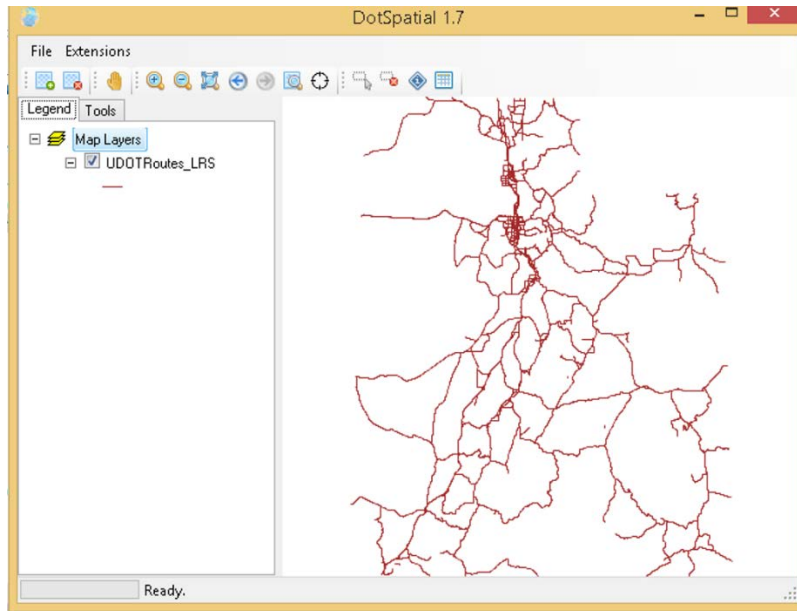


Figure 73: Brown Lines

1.5.3. Outlined Symbols

Objective:

Yellow Roads with Black Outlines

The line symbology is similar to the point symbology in that it also shares certain shortcut methods like “SetOutline”. The distinction is that unlike the simple symbol, strokes cannot come pre-equipped with an outline. Instead, the appearance of an outline is created by making two passes with two separate strokes. The first stroke is wider, and black. The second stroke is narrower and yellow. The result is a set of lines that appear to be connected. In order to get a clean look at the intersections, all the black lines are drawn first. Then, all the yellow lines are drawn. This way, the intersections appear to have continuous paths of yellow, rather than every individual shape being terminated by a curving black outline.

```
LineStyle road = new LineStyle(Color.Yellow, 5);
road.SetOutline(Color.Black, 1);
mylayer.Symbolizer = road;
```

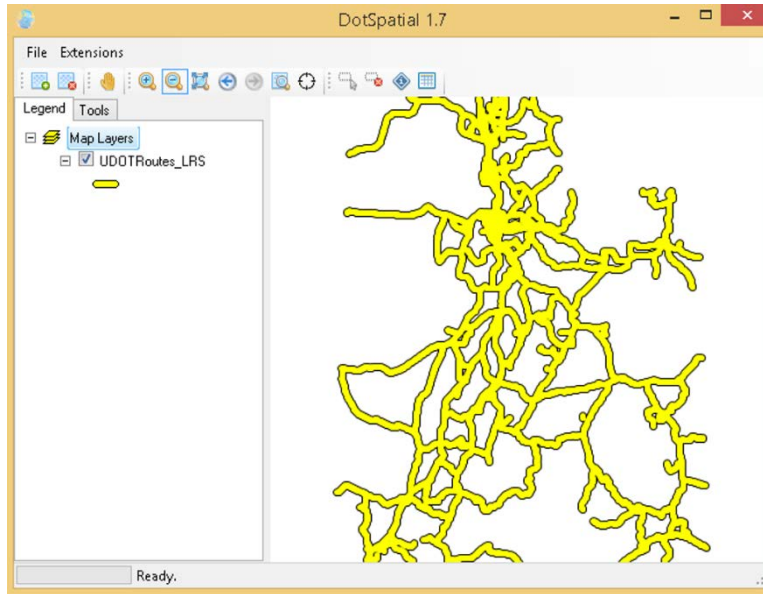


Figure 74: Unique Values

1.5.4 Unique Values

Objective:

Roads Colored by Unique Tile ID

One of the more useful abilities is to be able to programmatically apply symbology by unique values, without having to worry about what those values are or negotiate the actual color in each case. Simply specify a classification type of UniqueValues and a classification field, and DotSpatial does the rest. In this case, the default editor settings will create a hue ramp with a saturation and lightness in the range from .7 to .8. The editor settings can be used to control the acceptable range using the Start and End color. There is a Boolean property called HueSatLight. If this is true, then the ramp is created by adjusting the hue, saturation and lightness between the start and end colors. If this is false, then the red, blue and green values are ramped instead. In both cases, alpha (transparency) is ramped the same way.

```

LineScheme myScheme = new LineScheme(); myScheme.EditorSettings.ClassificationType =
ClassificationType.UniqueValues; myScheme.EditorSettings.FieldName = "CARTO";
myScheme.CreateCategories(mylayer.DataSet.DataTable);
mylayer.Symbology = myScheme;

```

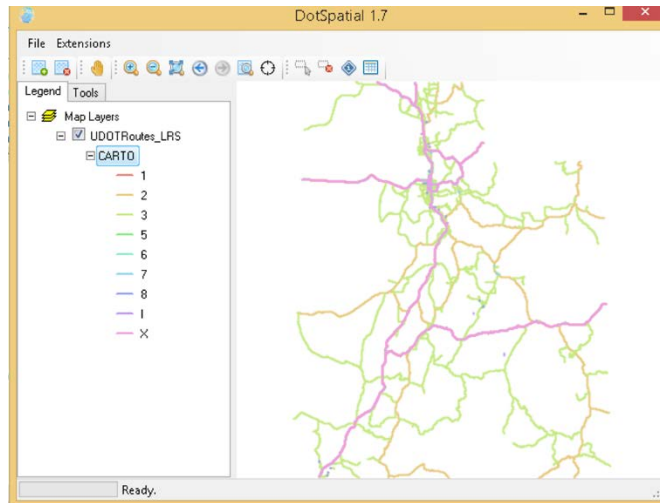


Figure 75: Roads with Unique Values

1.5.5 Custom Categories

Objective:

Custom Road Categories

In the previous example, the legend shows a collapsible field name in order to clarify the meaning of the values appearing for each category. This can also be accomplished manually by controlling the “AppearsInLegend” property on the scheme. If this is false, the categories will appear directly below the layer. When it is true, you can control the text in the legend using the scheme itself. Showing this principal in action, in this sample we will show the code that will programmatically set up two categories, and also have them appear under a scheme in the legend.

```

LineScheme myScheme = new LineScheme(); myScheme.Categories.Clear();
LineCategory low = new LineCategory(Color.Blue, 2);
low.FilterExpression = "[CARTO] = 3";
low.LegendText = "Low";
LineCategory high = new LineCategory(Color.Red, Color.Black, 6, DashStyle.Solid,
LineCap.Triangle);
high.FilterExpression = "[CARTO] = 2";

```

```
high.LegendText = "High";  
myScheme.AppearsInLegend = true;  
myScheme.LegendText = "CARTO";  
myScheme.Categories.Add(low);  
myScheme.Categories.Add(high);  
mylayer.Symbology = myScheme;
```

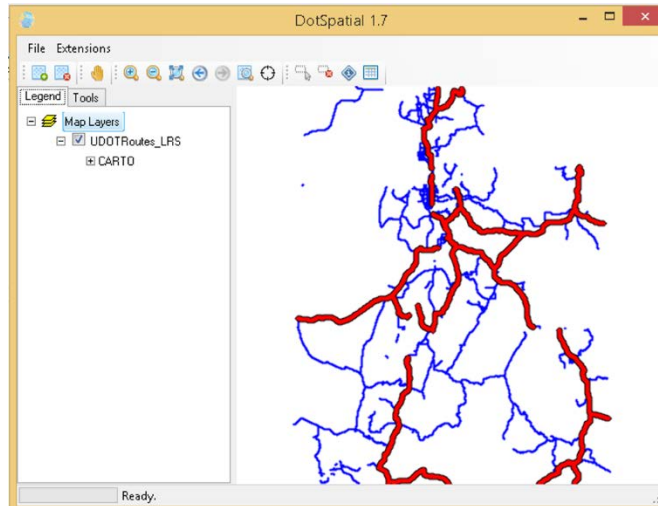


Figure 76: Custom Line Categories

1.5.6 Compound Lines

Objective:

Lines with Multiple Strokes

Each individual LineSymbolizer is made up of at least one, but potentially several strokes overlapping each other. The two main forms of strokes that are supported natively by DotSpatial are Simple Strokes and Cartographic Strokes. Cartographic strokes have a few more options that allow for custom dash configurations as well as specifying line decorations. Decorations are basically just point symbols that can appear at the end of the stroke, or evenly arranged along the length of the stroke.

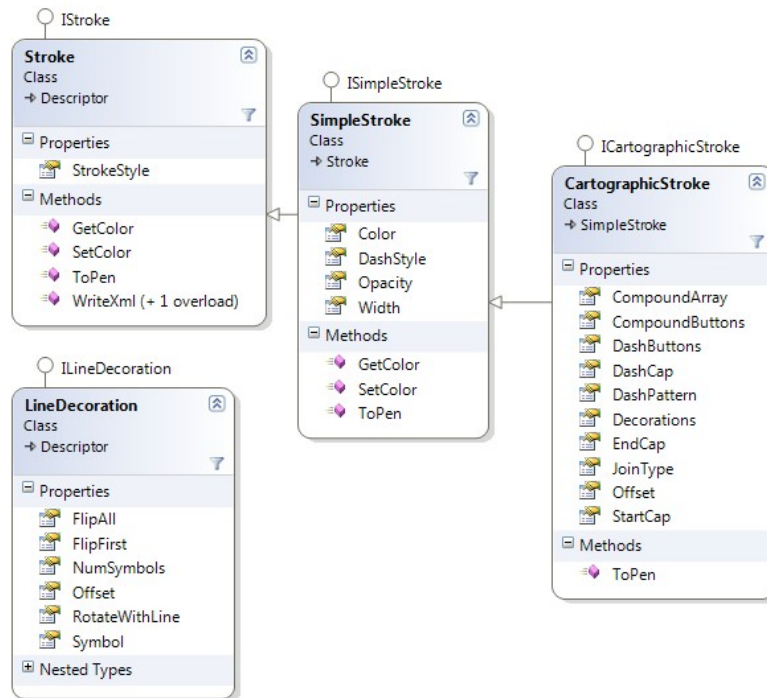


Figure 59: Stroke Class Hierarchy

In this example, we will take advantage of several powerful symbology options. We will use cartographic strokes in order to create two very different type of line styles. In the first case, we will create brown railroad ties. Using the standard “Dot” option for a simple cartographic line would not work because the dots created are proportional to the line width. However, with a custom dash pattern, it is possible to set the lines so that the dashes are thinner than the line width itself. The two numbers used in the dash pattern do not represent offsets, but rather the lengths of the dash and non-dash elements that alternate. This is convenient, since for our repeating ties, we really only need to specify two numbers.

The second layer of the symbol will be dark gray rails. In this case, the dash pattern is continuous, so we will not need to change it. However, the rails don’t persist the whole way across the line the way the ties do. Instead, we want to have two thin lines that appear along the path width. To do this, we take advantage of a CompoundArray. With the compound array, you are expressing the actual offsets for the start and end positions along the compound array, where 0 is the left of the line and 1 is the right. In some cases, lines that are two thin may not get drawn at all, so try to ensure that the width of the lines represented in the Compound array work out to be just slightly larger than 1 to ensure that the lines ultimately get drawn.

In the code below, the start and end caps are also specified. By default these are set to round, which will end up producing gray circles at each of the intersections. By specifying that the end caps should be flat, no extension will be added to ends of the lines. Rounded caps look the best for solid lines because it creates a kind of rounded, buffered look to roads that are wider than one pixel.

```
using System.Drawing.Drawing2D;
```

```
LineSymbolizer mySymbolizer = new LineSymbolizer();  
mySymbolizer.Strokes.Clear();  
CartographicStroke ties = new CartographicStroke(Color.Brown);  
ties.DashPattern = new float[] {1/6f, 2/6f};  
ties.Width = 6;  
ties.EndCap = LineCap.Flat; ties.StartCap = LineCap.Flat;  
CartographicStroke rails = new CartographicStroke(Color.DarkGray);  
rails.CompoundArray = new float[] {.15f, .3f, .6f, .75f};  
rails.Width = 6;  
rails.EndCap = LineCap.Flat;  
rails.StartCap = LineCap.Flat;  
mySymbolizer.Strokes.Add(ties);  
mySymbolizer.Strokes.Add(rails);  
mylayer.Symbolizer = mySymbolizer;
```

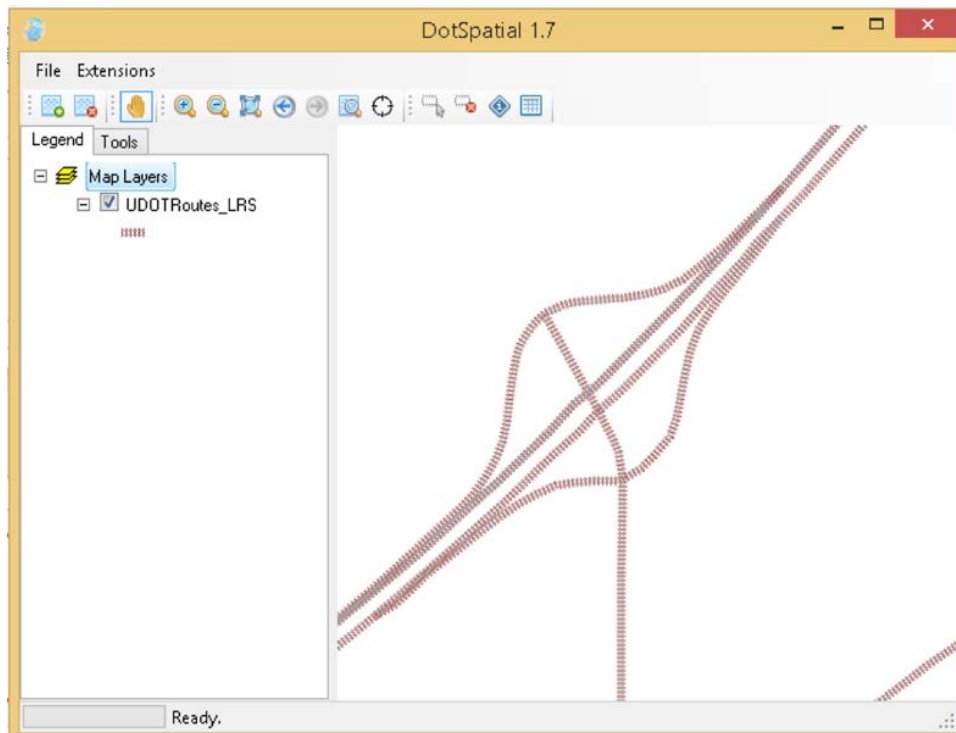


Figure 77: Multi-Stroke Railroads

1.5.7. Line Decorations

Objective:

Lines Decorated by Stars

One of the features with this generation of DotSpatial is the ability to add point decorations to lines. Each decoration has one symbolizer and can operate with several positioning options. Each stroke can support multiple decorations, so there is a great deal of customizable patterns available. The decorations can also be given an offset so that the decoration can appear on one side of the line or another. In this case, we will be adding yellow stars to a blue line.

```
LineDecoration star = new LineDecoration();
star.Symbol = new PointSymbolizer(Color.Yellow, DotSpatial.Symbology.PointShape.Star, 16);
star.Symbol.SetOutline(Color.Black, 1);
star.NumSymbols = 1;
CartographicStroke blueStroke = new CartographicStroke(Color.Blue);
blueStroke.Decorations.Add(star);
LineStyle starLine = new LineSymbolizer();
starLine.Strokes.Clear();
starLine.Strokes.Add(blueStroke); mylayer.Symbolizer =
starLine;
```

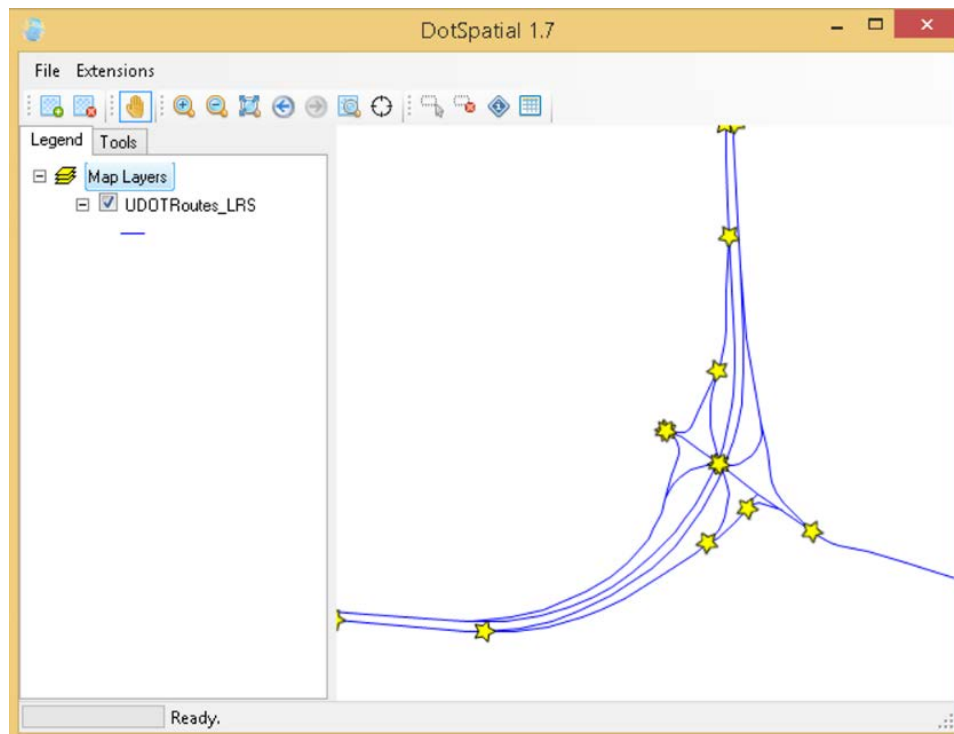


Figure 78: Lines with Star Decorations

1.6 Programmatic Polygon Symbology

1.6.1 Add Polygon Layers

Objective:

Add a Polygon Layer to the Map

Polygon layers are another representation of vector content where there is an area being surrounded by a boundary. Polygons can have any number of holes, which are represented as inner rings that should not be filled. However, in order to represent a shape like Hawaii, which has several islands, as a single shape, you would use a MultiPolygon instead. A MultiPolygon is still considered to be a geometry and will respond to all of the geometry methods, like Intersects. We can add the polygon shapefile the same way that we added the point or line shapefiles.

Polygon symbolizers are slightly different from the other two symbolizers because in the case of polygons, we have to describe both the borders and the interior. Since the borders are basically just lines, rather than replicating all the symbology options as part of the polygon symbolizer directly, each polygon symbolizer references a line symbolizer in order to describe the borders. This is a similar strategy to re-using the PointSymbolizer in order to describe the decorations that can appear on lines.

```
IFeatureSet fs = FeatureSet.Open(@"[Your Folder]\DotSpatialDev\shapefiles\Counties\Counties.shp")  
IMapFeatureLayer mylayer = map1.layers.Add(fs);
```

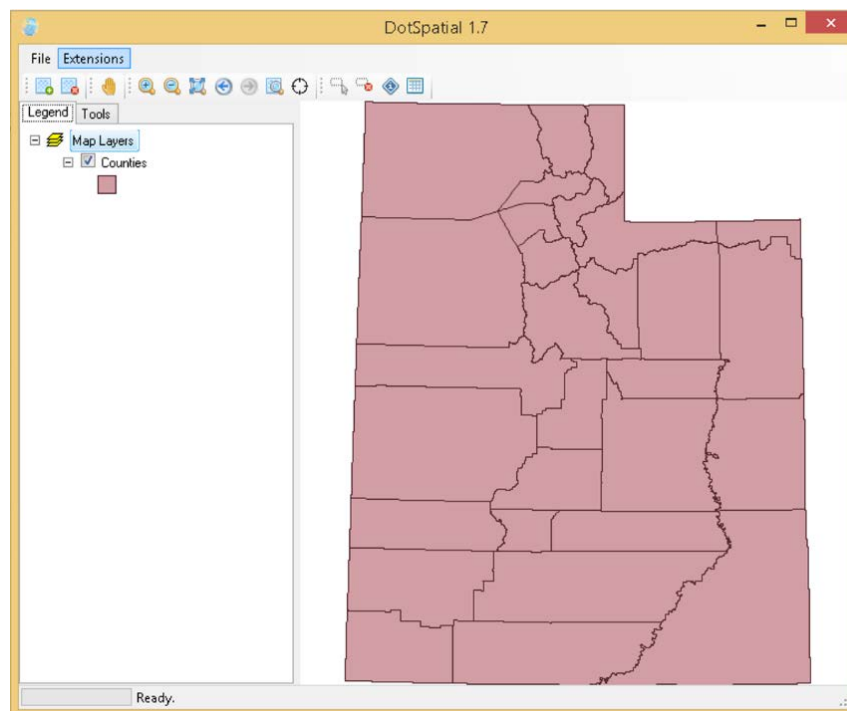


Figure 79: Add County Boundaries

1.6.2. Simple Patterns

Objective:

Specify Blue Polygons

The simplest task with polygons is to set the fill color for those polygons. You will see that specifying only an interior fill creates a continuous appearance, since the normal boundaries are adjacent and all the same color.

```
private void BluePolygons(IMapFeatureLayer mylayer)
{
    PolygonSymbolizer lightblue = new PolygonSymbolizer(Color.LightBlue);
    mylayer.Symbolizer = lightblue;
}
```

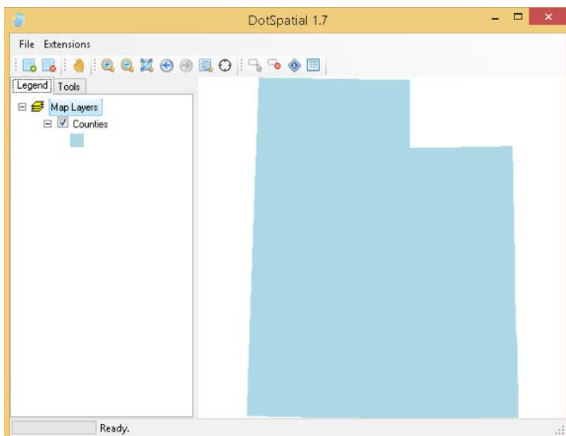


Figure 80: Blue Fill Only

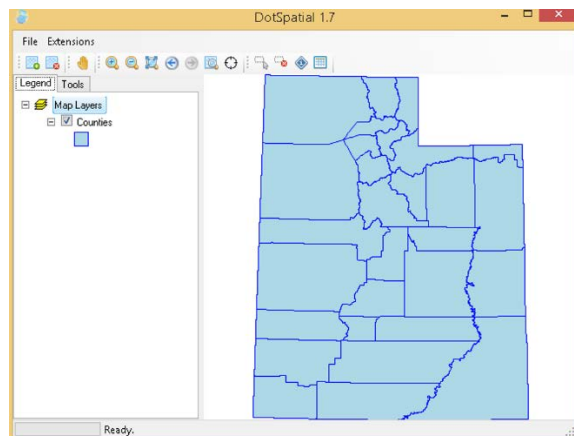


Figure 81: With Blue Border

```
PolygonSymbolizer lightblue = new PolygonSymbolizer(Color.LightBlue); lightblue.OutlineSymbolizer
= new LineSymbolizer(Color.Blue, 1);
mylayer.Symbolizer = lightblue;
```

1.6.3. Gradients

Objective:

Full Layer Gradient

One of the more elegant symbology options is to apply a gradient. These can vary in type from linear, to circular to rectangular, with the most frequently used type of gradient by far being linear. If you want to apply a continuous gradient across the entire layer, you can use the default category and simply specify the symbolizer. Unlike the previous example where we directly set up the outline symbolizer, in this example we are taking advantage of the shared method "SetOutline" which does the same thing. For points, this method controls the symbols themselves. For lines, this adds a

slightly larger stroke beneath the existing strokes. For polygons, this controls the line symbolizer that is used to draw the outline. The gradient angle is specified in degrees, moving counter-clockwise from the positive x axis.

```
PolygonSymbolizer blueGradient =  
new PolygonSymbolizer(Color.LightSkyBlue, Color.DarkBlue, 45, GradientType.Linear);  
blueGradient.SetOutline(Color.Yellow, 1);  
mylayer.Symbolizer = blueGradient;
```

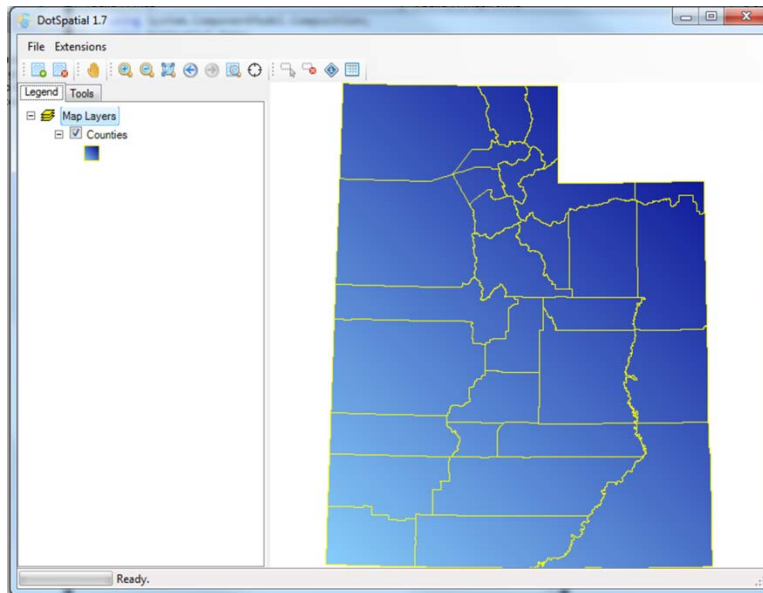


Figure 82: Continuous Blue Gradient

1.6.4. Individual Gradients

Objective:

Shape Specific Gradients

Another possible symbology is to create the gradients so that they are shape specific. This is not really recommended in the case of large numbers of polygons because the drawing gets linearly slower for each specific drawing call. You can draw thousands of polygons with one call by having only one symbolic class to describe all the polygons. In the case of a few hundred classes, this distinction is not really noticeable. To rapidly create different categories, we can take advantage of the “name” field which is different for each of the major shapes in the shapefile.

```

PolygonSymbolizer blueGradient = new PolygonSymbolizer(Color.LightSkyBlue, Color.DarkBlue,
    -45, GradientType.Linear);
PolygonScheme myScheme = new PolygonScheme(); myScheme.EditorSettings.TemplateSymbolizer =
blueGradient; myScheme.EditorSettings.UseColorRange = false;
myScheme.EditorSettings.ClassificationType = ClassificationType.UniqueValues;
myScheme.EditorSettings.FieldName = "nam";
myScheme.CreateCategories(mylayer.DataSet.DataTable);
mylayer.Symbology = myScheme;

```

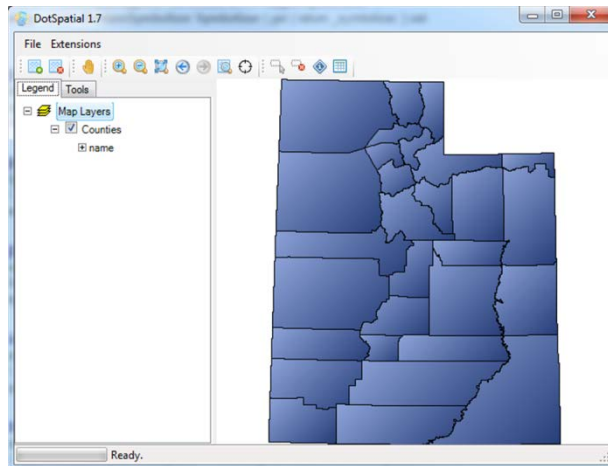


Figure 83: Individual Gradients

1.6.5 Multi-Colored Gradient

Objective:

Cool Colors with Gradient

One thing that may seem less than obvious is that in the previous exercise we specified that the UseGradient property should be false. This does not prevent the template symbolizer from having a gradient. Instead, it prevents the symbolizer from overriding the original, presumably simpler, template with a gradient symbol. The gradient symbol will be calculated using a color from the color range, but then will make the upper left a little lighter and the lower right a little darker. That way, you can have the same subtle gradient applied, but still use different colors for each category. To boot, the default polygon symbolizer has a border that is the same hue, but slightly darker, which tends to create a nice outline color.

```

PolygonScheme myScheme = new PolygonScheme();
myScheme.EditorSettings.StartColor = Color.LightGreen;
myScheme.EditorSettings.EndColor = Color.LightBlue;
myScheme.EditorSettings.ClassificationType = ClassificationType.UniqueValues;
myScheme.EditorSettings.FieldName = "name";
myScheme.EditorSettings.UseGradient = true;
myScheme.CreateCategories(mylayer.DataSet.DataTable);

```

```
mylayer.Symbology = myScheme;
```

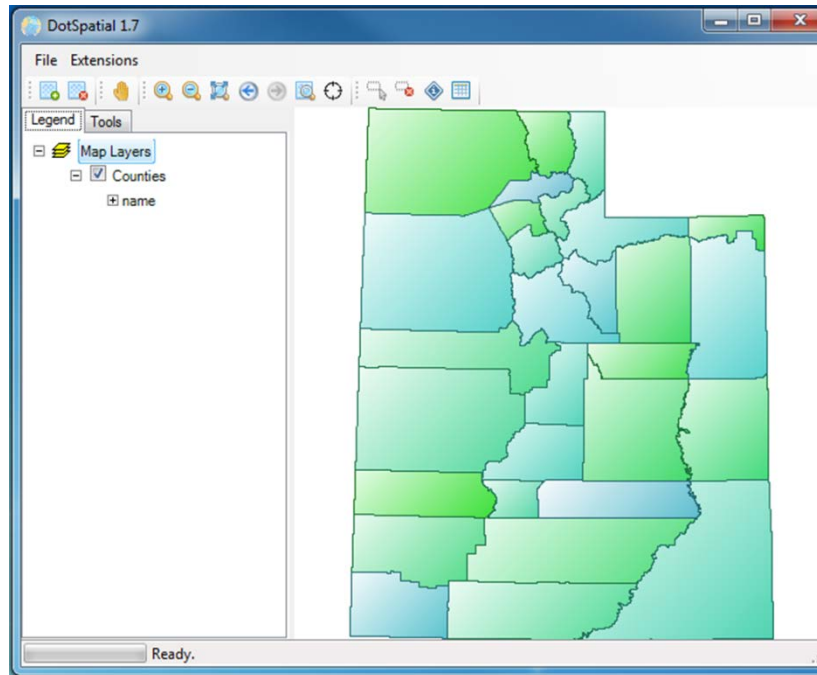


Figure 84: Unique Cool Colors with Gradient

1.6.6. Custom Polygon Categories

Objective:

Custom Polygon Categories

An important terminology difference here that we have been using is the difference between Symbolizer and Symbology. With Symbology, we are always referring to a scheme, which can have many categories. With a Symbolizer, we are talking about controlling how those shapes are drawn for one category. By default, all the feature layers start with a scheme that has exactly one category, which has a symbolizer with exactly one drawing element (symbol, line or pattern). The Symbolizer property on a layer is a shortcut to the top-most category. If you have several categories, it may be better to control the symbolizers explicitly than to use the shortcut. Labels have been added to the layer below in order to illustrate that the two pink shapes are in fact shapes that start with G. The actual labeling code will be illustrated under a separate section under labeling.

```
PolygonScheme scheme = new PolygonScheme();
PolygonCategory washington = new PolygonCategory(Color.LightBlue,
    Color.DarkBlue, 1);
washington.FilterExpression = "[name] = 'Washington'";
washington.LegendText = "Washington";
PolygonCategory gWords = new PolygonCategory(Color.Pink, Color.DarkRed, 1);
gWords.FilterExpression = "[name] Like 'G*'";
gWords.LegendText = "G - Words";
```

```
scheme.ClearCategories();  
scheme.AddCategory(washington);  
scheme.AddCategory(gWords);  
mylayer.ShowLabels = true;  
mylayer.Symbology = scheme;
```

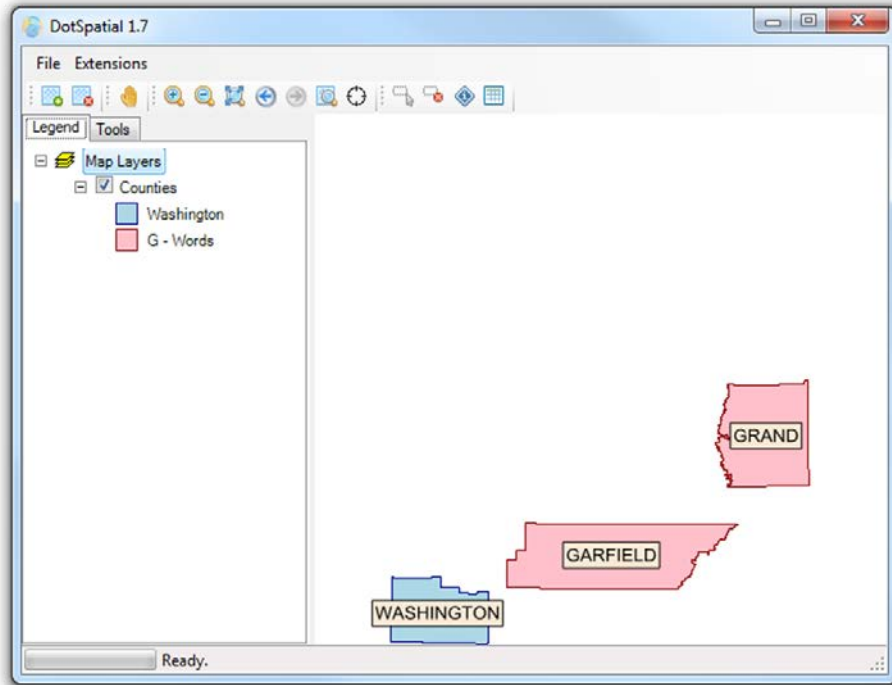


Figure 85: Custom Categories

1.6.7 Compound Patterns

Objective:

Multi-LayerPatterns

Like the other previous symbolizers, polygon symbolizers can be built out of overlapping drawing elements. In this case they are referred to as patterns. The main patterns currently supported are simple, gradient, picture and hatch patterns. Simple patterns are a solid fill color, while gradient patterns can work with gradients set up as an array of colors organized in floating point positions from 0 to 1. The angle controls the direction of linear and rectangular gradients. Hatch patterns can be built from an enumeration of hatch styles. Picture patterns allow for scaling and rotating a selected picture from a file.

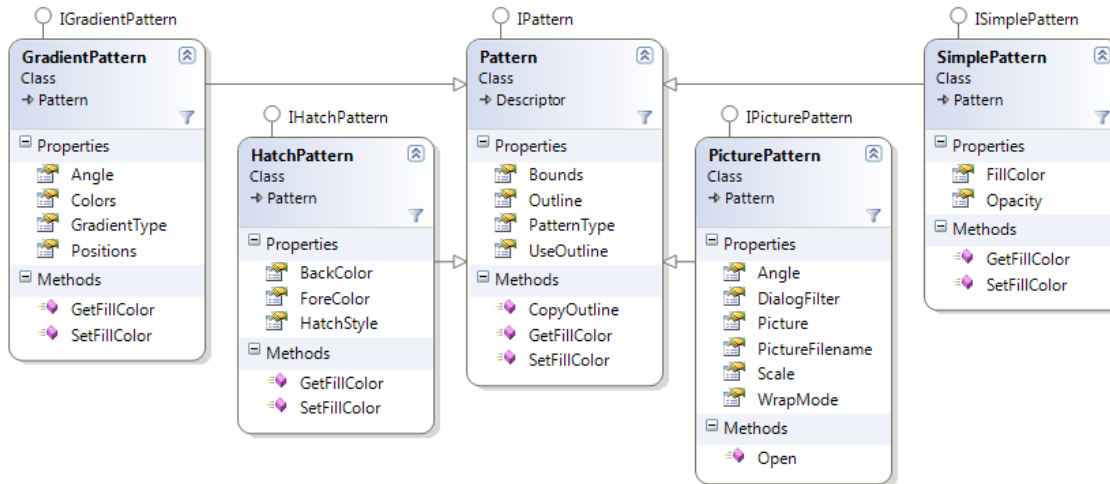


Figure 86: Pattern Class Diagram

One thing in particular to note about this next example is that in all the previous examples with multiple patterns, we simply cleared out the default pattern that was automatically created as part of the symbolizer. When we add a new pattern, the new pattern gets drawn on top of the previous patterns, so the last pattern added has the highest drawing priority. In the code below, the new pattern has its background color set to transparent, yet in the image below we see that the coloring is red stripes against a blue background. The pattern below the red-stripe pattern is the default pattern, and will be randomly generated as a different color each time.

```

PolygonSymbolizer mySymbolizer = new PolygonSymbolizer();
mySymbolizer.Patterns.Add(new HatchPattern(HatchStyle.WideDownwardDiagonal,
    Color.Red, Color.Transparent));
myLayer.Symbolizer = mySymbolizer;
  
```

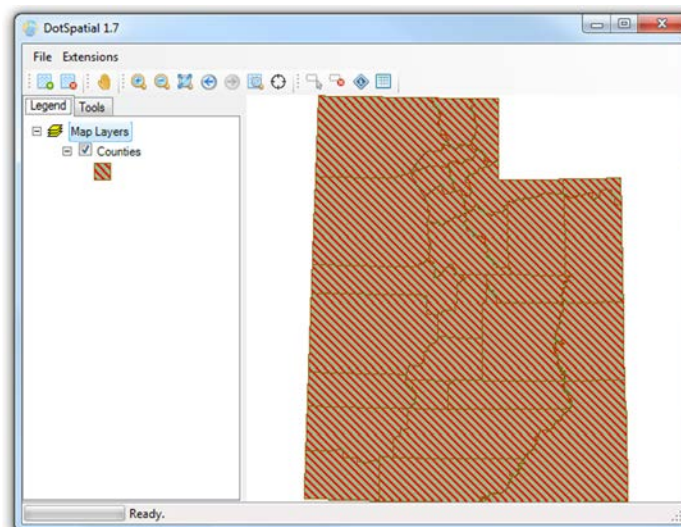


Figure 87: Hatch Patterns

1.7 Programmatic Labels

Objective:

Labels

For the first example with labels, we will show adding your own text in a way so that the same text gets added to all the features. This uses the default settings, and you can see from the default settings that there is no background, and each label has the left side aligned with the center point by default.

```
IMapLabelLayer labelLayer = new MapLabelLayer();  
labelLayer.Symbology.Categories[0].Expression = "Test";  
myLayer.ShowLabels = true;  
myLayer.LabelLayer = labelLayer;
```

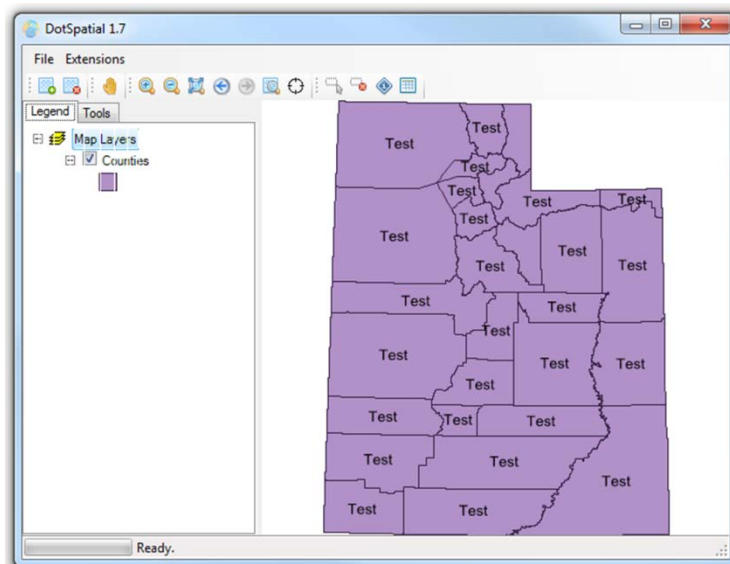


Figure 88: Adding Test Labels

1.7.1 Field Name Expressions

Objective:

Field Name Labels

The field name in this case describes the name of the territory. In order for the name field to appear in the label text, we simply enclose it in square brackets. This puts together a versatile scenario where you can build complex expressions with various field names. You can also use escape characters to create multi-line labels.


```

IMapLabelLayer labelLayer = new MapLabelLayer(); ILabelCategory
category = labelLayer.Symbology.Categories[0]; category.Expression
= "[NAME]";
category.Symbolizer.Orientation = ContentAlignment.MiddleCenter;
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;

```

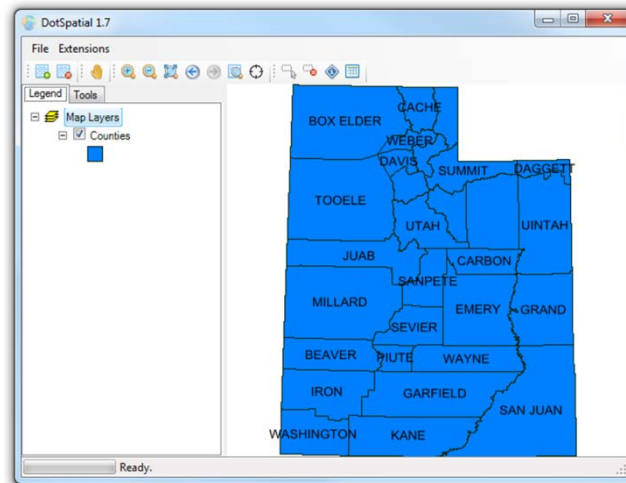


Figure 89: Field Name Labels

1.7.2 Multi-Line Labels

Objective:

Multi-Line Labels

Creating multi-line labels is simple, since all you have to do is use the standard .Net new-line character, which in C# is added using the `\n`, while in visual basic you would combine the two strings with a `vbNewLine` element between them. The relative position of the multiple lines is controlled by the `Alignment` property on the label `Symbolizer`. In order to minimize confusion, the labels follow the same organization with a scheme, categories and symbolizers. A filter expression also allows us to control which labels are added.

```

IMapLabelLayer labelLayer = new MapLabelLayer(); ILabelCategory
category = labelLayer.Symbology.Categories[0]; category.Expression
= "[NAME]\nPopulation: [POP_LASTCE]"; category.FilterExpression =
"[NAME] Like 'G*'"; category.Symbolizer.BackgroundColorEnabled = true;

```

```

category.Symbolizer.BorderVisible = true;
category.Symbolizer.Orientation = ContentAlignment.MiddleCenter;
category.Symbolizer.Alignment = StringAlignment.Center;
myLayer.ShowLabels = true;
myLayer.LabelLayer = labelLayer;

```

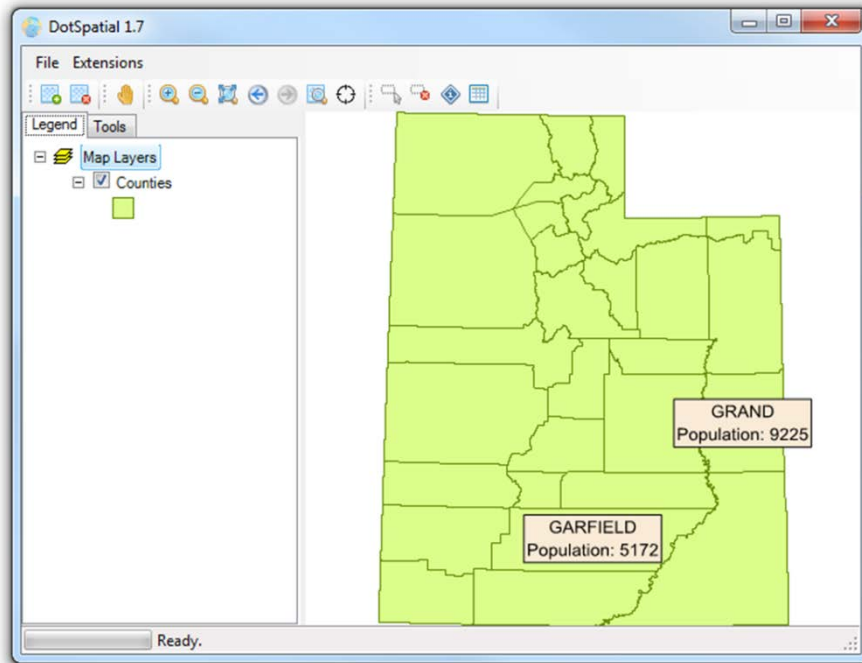


Figure 90: Multi-Line Labels

Multiple label categories can be created and added to the scheme in the same way that the featuresets were able to add different categories. The labels also allow for the font to be controlled, as well as the text color, background color and opacity of either.

1.7.3. Translucent Labels

Objective:

Translucent Labels

The background color in this case has been set to transparent by specifying an alpha value of something less than 255 when setting the BackColor. Frequently, there are opacity properties available in addition to the actual color, but that is just there for serialization purposes. This example also illustrates the use of the compound conjunction “OR” in the filter expression. Other powerful terms that can be used are “AND” and “NOT” as well as the combined expression “Is Null” which is case insensitive and can identify null values separately from empty strings for instance. Notice that is not “= null” which doesn’t work with .Net DataTables. For the negative you could use “NOT [NAME] is null”.

```

IMapLabelLayer labelLayer = new MapLabelLayer();
ILabelCategory category = labelLayer.Symbology.Categories[0];
category.Expression = "[NAME]\nPopulation: [POP_LASTCE]";

```

```
category.FilterExpression = "[NAME] = 'Tooele' OR [NAME] = 'Kane'";
category.Symbolizer.BackgroundColorEnabled = true;
category.Symbolizer.BackgroundColor = Color.FromArgb(128, Color.LightBlue);
category.Symbolizer.BorderVisible = true;
category.Symbolizer.FontStyle = FontStyle.Bold;
category.Symbolizer.FontColor = Color.DarkRed;
category.Symbolizer.Orientation = ContentAlignment.MiddleCenter;
category.Symbolizer.Alignment = StringAlignment.Center;
mylayer.ShowLabels = true;
mylayer.LabelLayer = labelLayer;
```

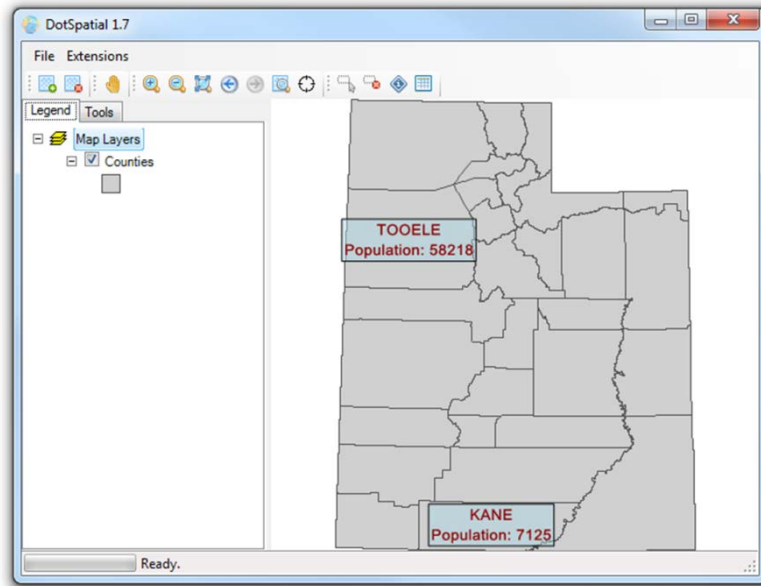


Figure 91: Translucent Labels

1.8. Programmatic Raster Symbology

1.8.1 Download Data

Objective:

Download a Raster Layer

In addition to the vector data that we have been looking at so far, DotSpatial also supports a number of raster formats. Rasters are considered distinct from images for us in that the visual representation that we see is derived from the values much in the way that we derive the polygon images from the actual data. With rasters, you typically have a rectangular arrangement of values that are organized in rows and columns. For datasets that don't have complete sampling, a "No-Data" value allows the raster to only represent a portion of the total area.

Because of the existence of extensible data format providers, there is no way to tell just how many formats DotSpatial will support at the time you are reading this. We have created a plug-in that is exclusive to the windows platform using Frank Warmerdan's GDAL libraries and C# linkage files. The

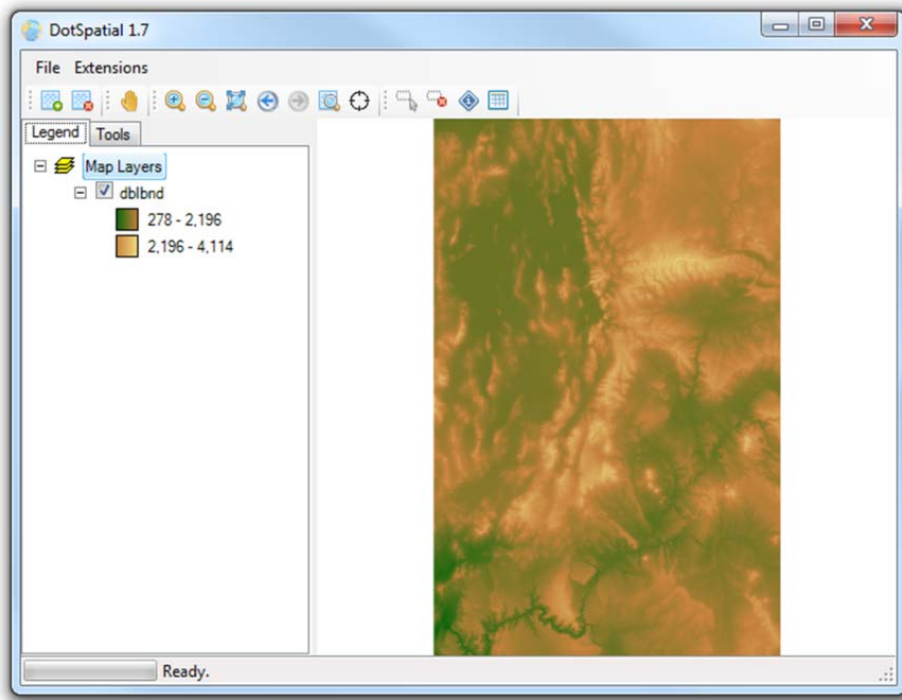


Figure 94: Default Raster

Sometimes the symbology is thrown off by an unexpected no-data value. We can use the symbolizer interface for rasters in DotSpatial by double clicking next to the elevation layer in the legend. This will launch a dialog like the one below.

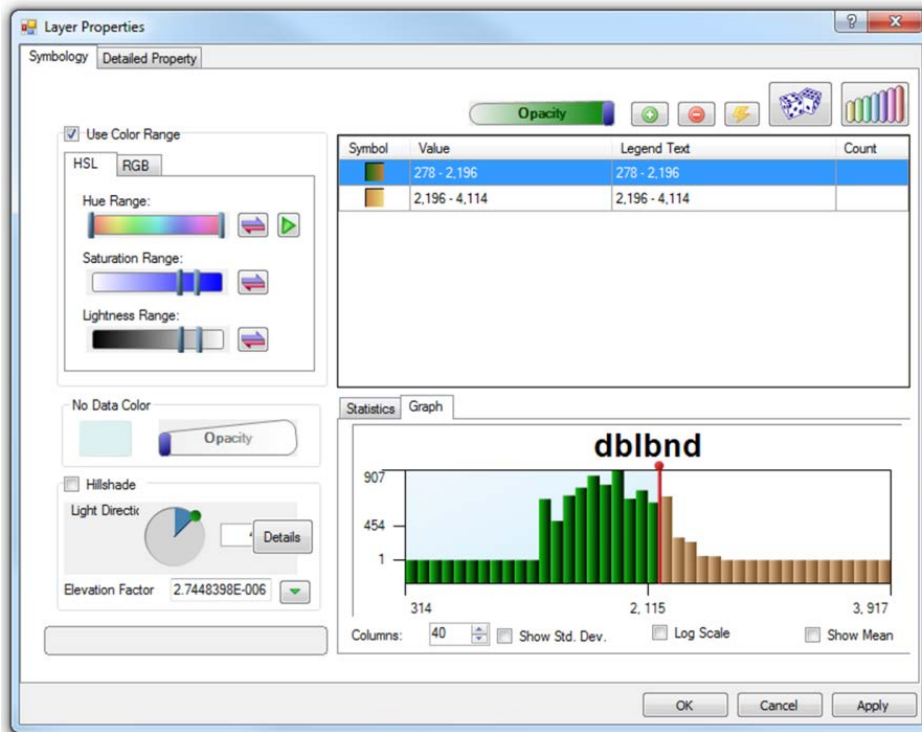


Figure 95: Graph

Sometimes the graph will not display properly. In order to have it fix this simply change one of the data values such as the number of separations and click cancel. Once you reopen the interface, the graph should display properly.

We can see right away that of the many sample points that were taken from this image that the majority of them lie in the 1-4000 range. We can change the value of the separation by dragging the bars along the graph. We can also change the values numerically by double clicking "Value" column on the table. By right clicking on the graph we can also zoom to a specific category. In addition we change the color options from this menu.

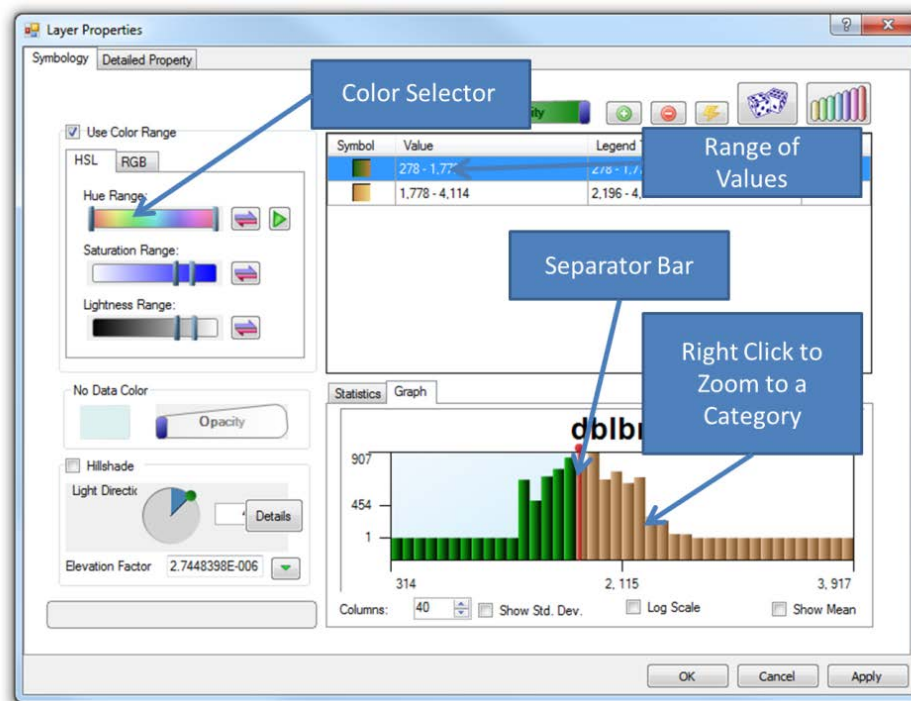


Figure 96: Layer Properties

The next step is to modify these values programmatically. Unfortunately for unknown reasons we cannot modify the view range of the DEM of Utah. It appears that there is a problem with the DEM itself. Even after converting it to different file formats it does not function. For the following examples we will use a DEM downloaded from <http://www.chartiff.com/Index.htm> on their sample page.

1.8.3 Control Category Range

Objective:

Control Category Range

The only thing to notice when controlling the range is that you can control the range values independently from modifying the legend text. In order to update the legend text based on other settings, we can use the ApplyMinMax settings. Alternately, we could have set the legend text directly, just as we can for the other categories. Since the new DEM we downloaded has a range of about 3,200 to 3700, we will split the range values in half. The first category will go from 3,200 to 3,450 and the second category will go from 3,440 to 3,700.

```
private void ControlRange(IMapRasterLayer myLayer)
{
    myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(3200, 3450);
    myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
    myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(3440, 3700);
    myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
    myLayer.WriteBitmap();
}
```

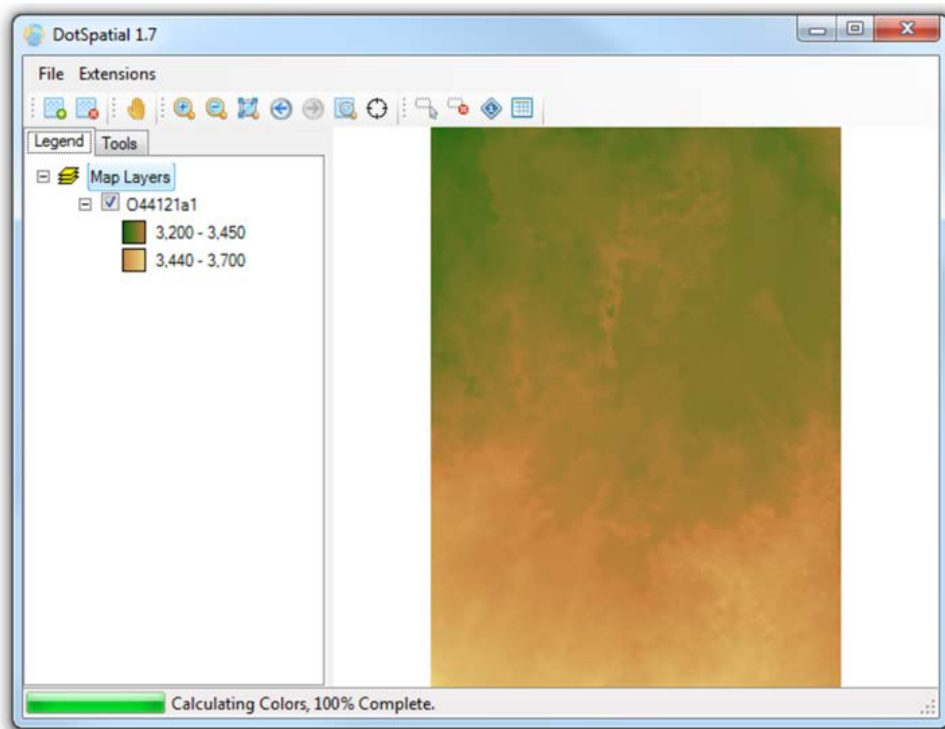


Figure 97: Programmatic Restrict Range

1.8.4. Shaded Relief

Objective:

Add Lighting

```
myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(3200, 3450);
myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(3440, 3700);
```



```

myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1; myLayer.Symbolizer.ShadedRelief.IsUsed
= true;
myLayer.WriteBitmap();

```

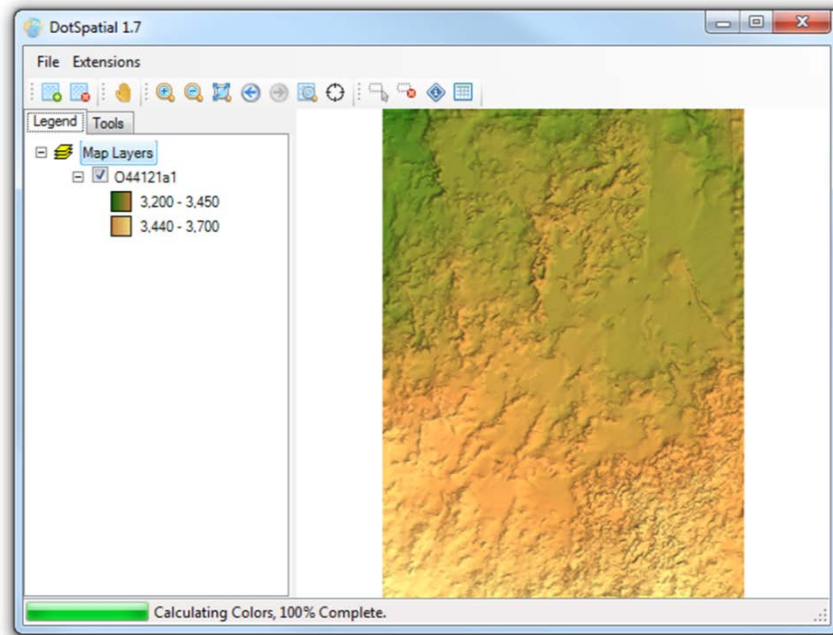


Figure 98: With Lighting

1.8.5. Predefined Schemes

Objective:

Use Glacier Coloring

There are several pre-defined color schemes that can be used. All of the pre-set schemes basically use two separate ramps that subdivide the range and apply what is essentially a coloring theme to the two ranges. Those ranges are easily adjustable using the range characteristics on the category, but should be adjusted after the scheme has been chosen, or else applying the new scheme will overwrite the previous range choices.

```

myLayer.Symbolizer.Scheme.ApplyScheme(ColorSchemeType.Glaciers, myLayer.DataSet);
myLayer.Symbolizer.Scheme.Categories[0].Range = new Range(3200, 3450);
myLayer.Symbolizer.Scheme.Categories[0].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.Scheme.Categories[1].Range = new Range(3440, 3700);
myLayer.Symbolizer.Scheme.Categories[1].ApplyMinMax(myLayer.Symbolizer.EditorSettings);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1;
myLayer.Symbolizer.ShadedRelief.IsUsed = true;
myLayer.WriteBitmap();

```

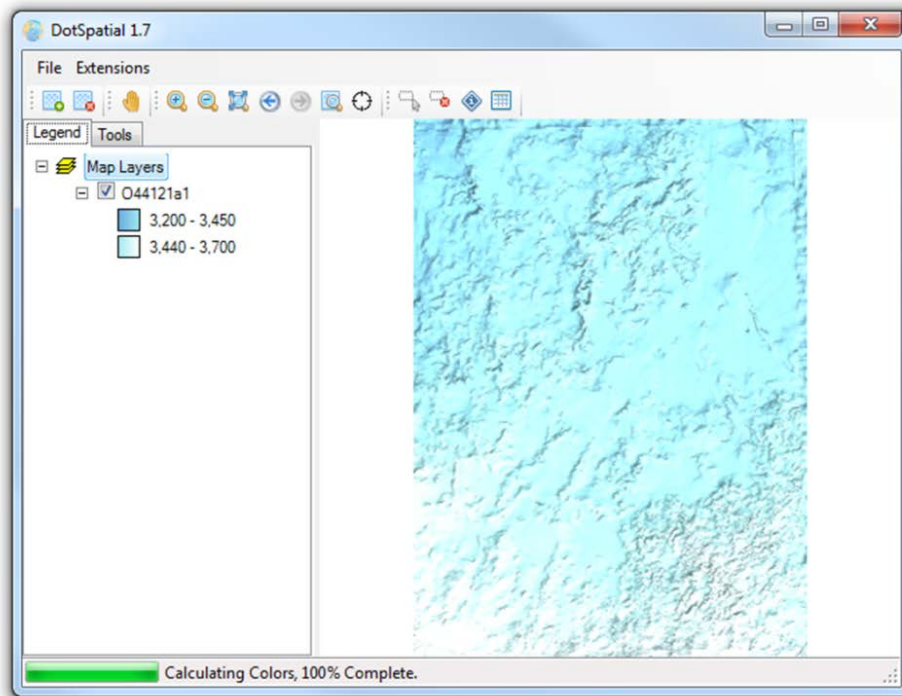



Figure 99: Glaciers

1.8.6 Edit Raster Values

Objective:

Fix Raster Values

Previously, we have been using tricks to color the elevation but where we directly ignored the values that were either no-data values that read 0, or else apparently bad data values that showed impossible values like 65,000. In this section we will use the raster data class itself in order to repair the values programmatically. This will only alter the copy that we have in memory and will not overwrite the values to the disk unless we specifically instruct it to do so. In the example below, we can cycle through all of the values in the raster using the NumRows and NumColumns properties to give us an idea of what the bounds are on the loops. The Value property takes a double index, and will work with whatever the real data type is and convert that data type into doubles. We can use this to quickly clean up the values on the raster before we ever create a layer. We can also assign the no data value on the raster so that it matches the 0 values that cover a large portion of the raster. This will automatically eliminate it from the statistical calculations so that our default symbology should look better.

```

IRaster r = Raster.Open(@"C:\Users\Matt\Documents\DotSpatialDev\Rasters\sampleDEM\044121a1.dem");
r.NoDataValue = 0;
for (int row = 0; row < r.NumRows; row++)
{
    for (int col = 0; col < r.NumColumns; col++)
    {
        if (r.Value[row, col] > 3700) r.Value[row, col] = 3700;
    }
}
IMapRasterLayer mylayer = map1.Layers.Add(r);

```

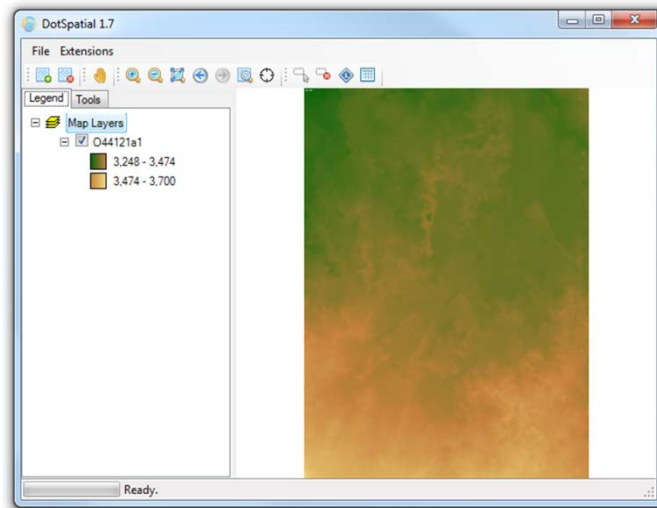


Figure 100: Default Symbology of Fixed Raster

This looks better, and if we double click on the elevation layer, we can take a look at what the statistical plot automatically shows. Assigning the no-data value can be risky because there may be values that were using the old no-data value. This can easily be fixed by cycling through the raster in the same way and adjusting values so that they work with the given statistics.

1.8.7. Quantile Breaks

Objective:

Quantile Break Values

Just like the FeatureSets, Rasters can use the EditorSettings property in order to customize how to build schemes, rather than having to specify the schemes directly. This is where our previous edits to fix the raster values become more important. If we had tried to apply quantile breaks before, instead of coloring the raster appropriately, we would have had all but one of the ranges read 0-0. Now, we get a reasonable range.

```

myLayer.Symbolizer.EditorSettings.IntervalMethod = IntervalMethods.Quantile;
myLayer.Symbolizer.EditorSettings.NumBreaks = 5;
myLayer.Symbolizer.Scheme.CreateCategories(myLayer.DataSet);
myLayer.Symbolizer.ShadedRelief.ElevationFactor = 1; myLayer.Symbolizer.ShadedRelief.IsUsed =
true;

```

```
myLayer.WriteBitmap();
```

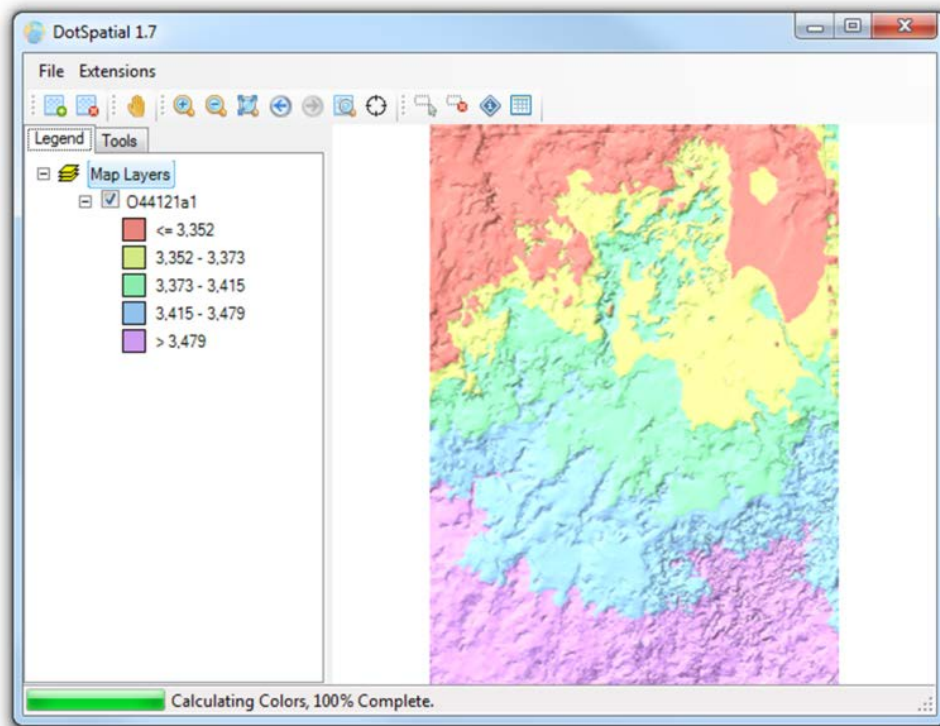


Figure 101: Quantile Breaks

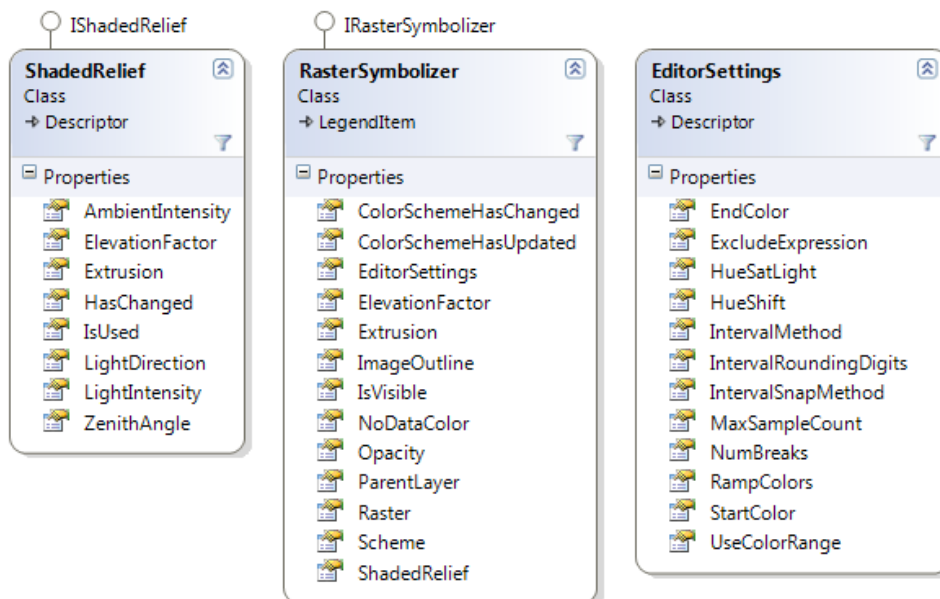


Figure 102: Raster Symbology Classes

1.9. Extension Methods

Objective:

Extension Methods

Rasters

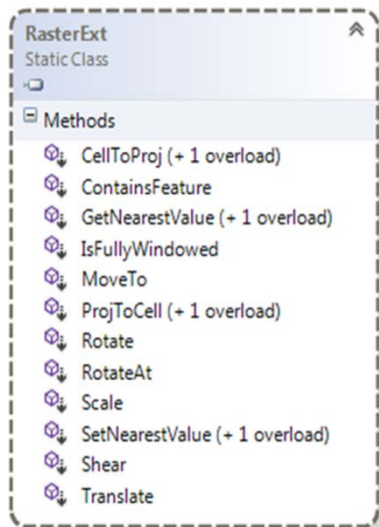


Figure 103: Raster Extension

on a raster. Many useful methods are actually supported in the form of “Extension Methods”. For the raster, this includes useful methods that can alter the raster bounds, like Translate, or Scale, or Rotate. But other useful abilities like CreateHillShade actually use the raster values themselves in order to calculate a floating point value that helps to control the shaded relief aspect of any image that is created from a raster. Other methods like GetRandomValues, and GetNearestValue are helpful for doing analysis, but one of the most critical methods is CellToProj and ProjToCell, which allows the developer to easily go back and forth between geospatial coordinates and the row and column indices.

Not all of the methods are supported directly

Feature

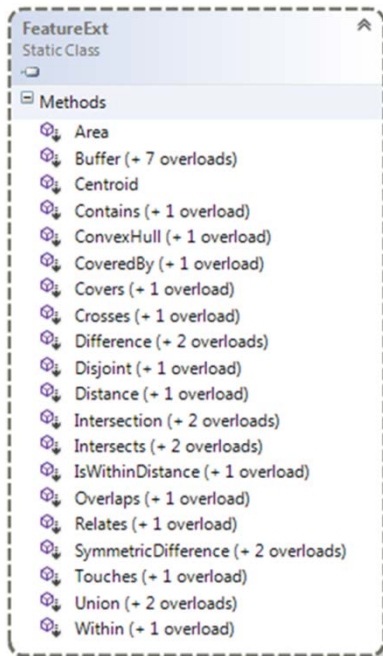


Figure 104: Feature Extension

Feature, and in fact any class that implements the IFeature interface will be extended with the geometry methods that are so critical to vector calculations. These not only include the overlay operations like Intersection, Union and Symmetric Difference, but all the tests that you might want to use like touches or within. Some of the bonus methods are methods like Area, which calculates the areas of polygons. Another is Centroid, which calculates the center of mass for geometries. ConvexHull can be used to simplify a geometry in the same way that you would simplify something by wrapping it with an elastic band. It draws straight lines past concave sections, and follows around with the convex portions. The Distance tool finds the minimum distance between two geometries, and the IsWithinDistance simply changes the Distance calculation to test it against a threshold.

1.10 Adding Additional Plugins and Extensions

Objective:

Plugins and Extensions

Throughout this tutorial we have used several plugins and extensions in order to make a basic GIS application. Since DotSpatial uses a “plugin architecture” it is relatively easy to add extensions and plugins to your project. In this section we will cover step by step how to install some of the more useful and common plugins and extensions available for DotSpatial. If you have already gone through this whole document then much of this will be review. For each tutorial we will start with a blank project, however, for complete details on creating the project see 1.10.1.

1.10.1 GDAL

Objective:

Add GDAL Extension

As mentioned previously, adding the GDAL extension will allow your project to use many more types of data. GDAL stands for “Geospatial Data Abstraction Layer” and is an Open source project by Frank Warmarda. It is used in nearly every GIS application including ArcGIS. In DotSpatial it is used through an extension and an “AppManger.” Be sure to download DotSpatial before beginning. The first thing we need to do is to start a new Visual Studio Project. We will name our project GDAL. Make sure to use the C# template and to select “Windows Forms Application.”

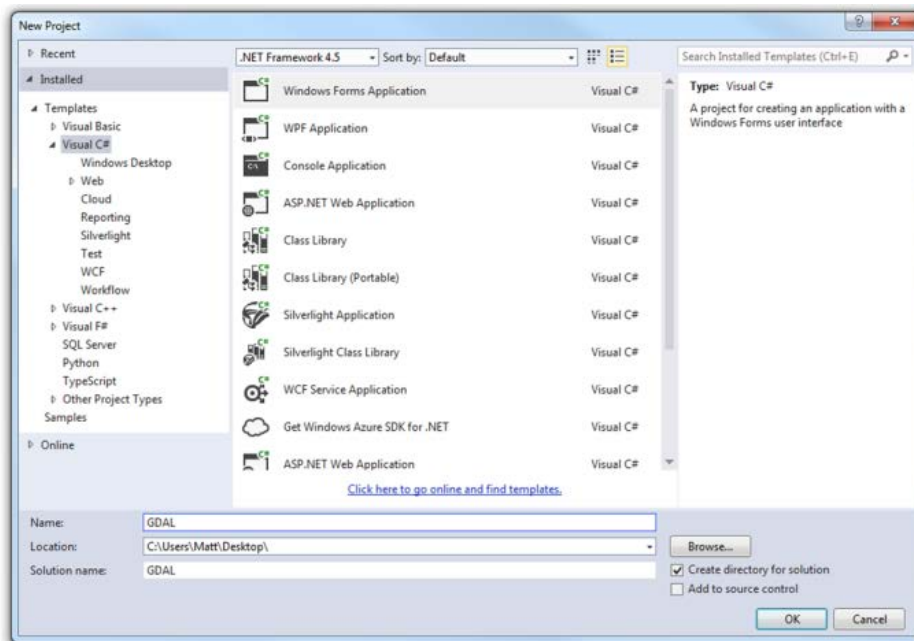


Figure 105: New Project

We next need to add a .Net Framework Assembly reference to “System.ComponentModel.Composition.” Right click on the “References” folder in the “Solution Explorer” and click “Add Reference.”

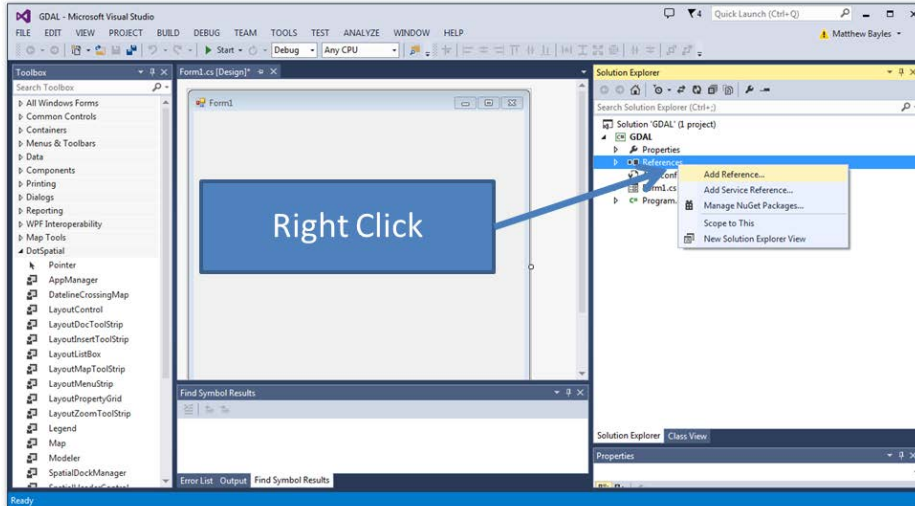


Figure 106: Reference Folder

Go the “Assemblies” tab and click on “Framework.” Scroll down until you come to “System.ComponentModel.Composition” and select it.

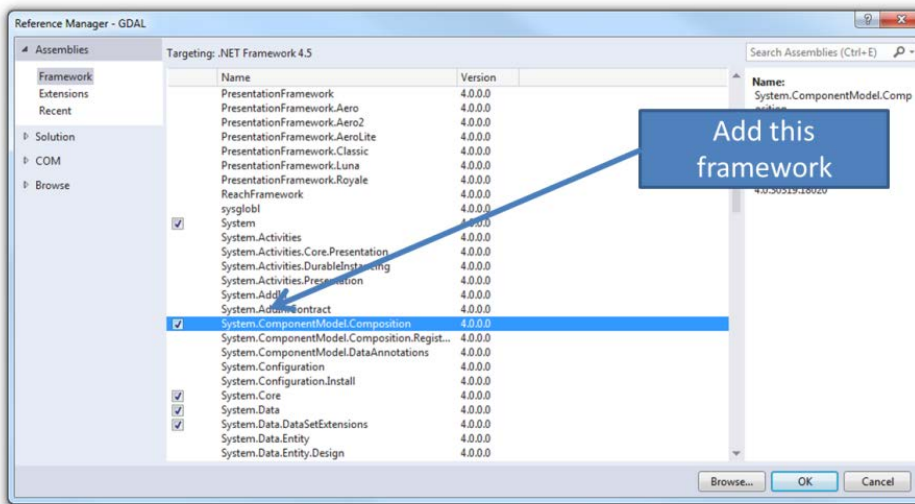


Figure 107: System.ComponentModel.Composition

Next go to the “Browse” tab and browse to where you saved the DotSpatial download. Be sure to use the unzipped folder and make sure that it is not blocked. In this folder you will find the DotSpatial dlls. Select all of them and click “Add.”

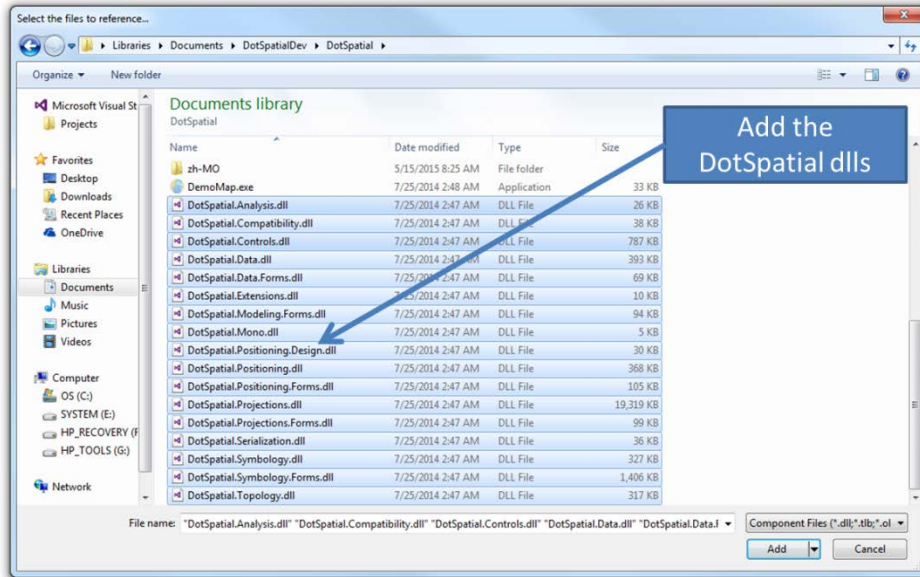


Figure 108: DotSpatial dlls

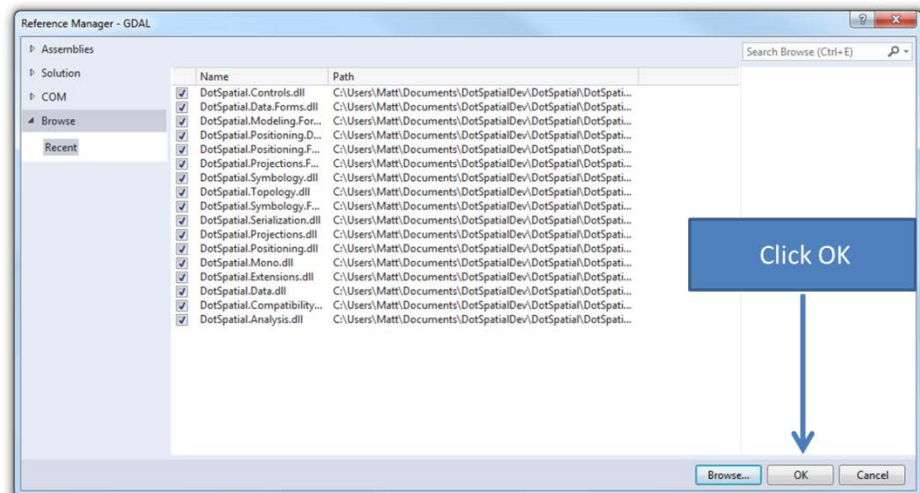


Figure 109: Add the dlls

You're references folder should now look like Figure 110.

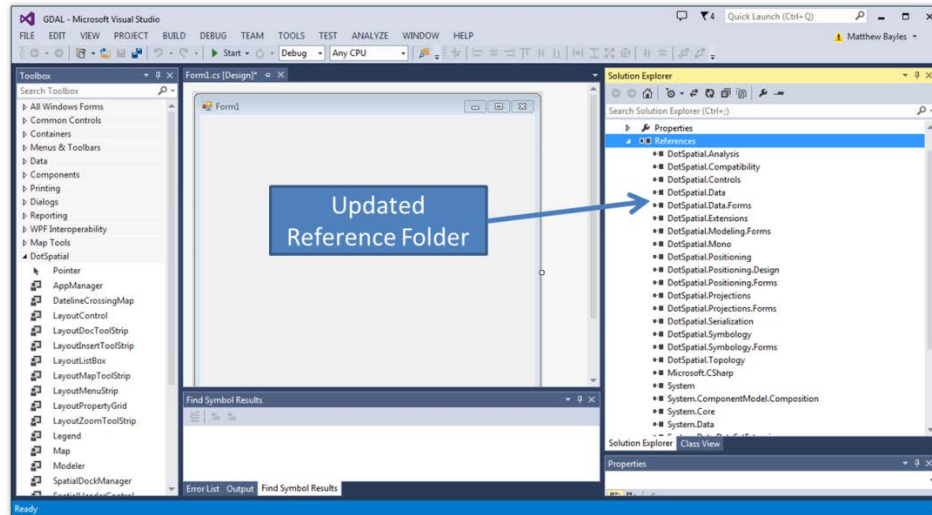


Figure 110: Updated Reference Folder

Before we can use the “AppManger” and other map components we will need to add the DotSpatial Controls to our toolbox. Go to the toolbox tab and right click to add a new tab. We will call this tab “DotSpatial.”

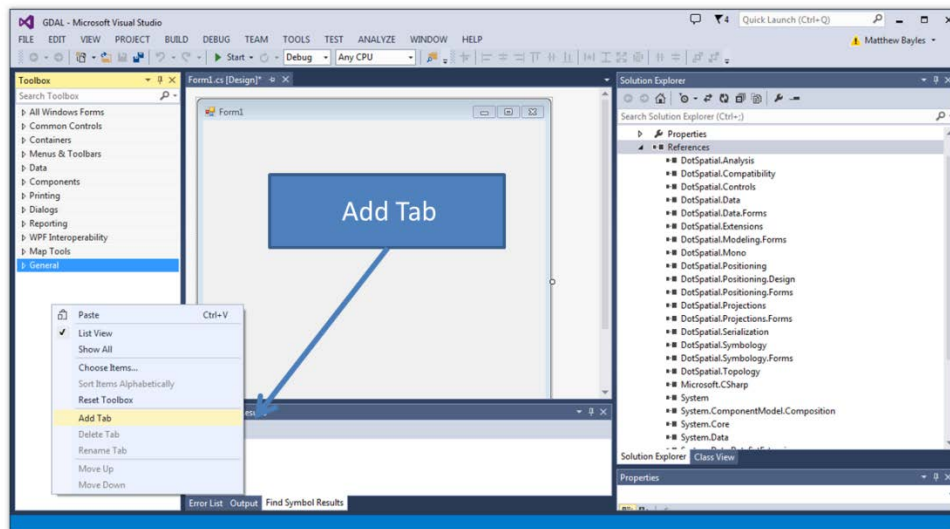


Figure 111: Toolbox

Right click on your new tab and select “Choose Items.”

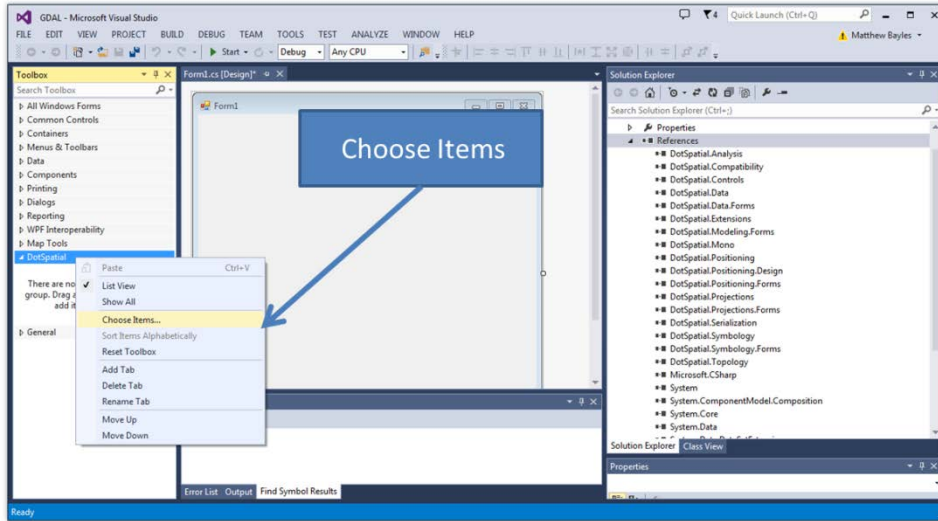


Figure 112: Choose Items

A new window will appear. Click on “Browse.”

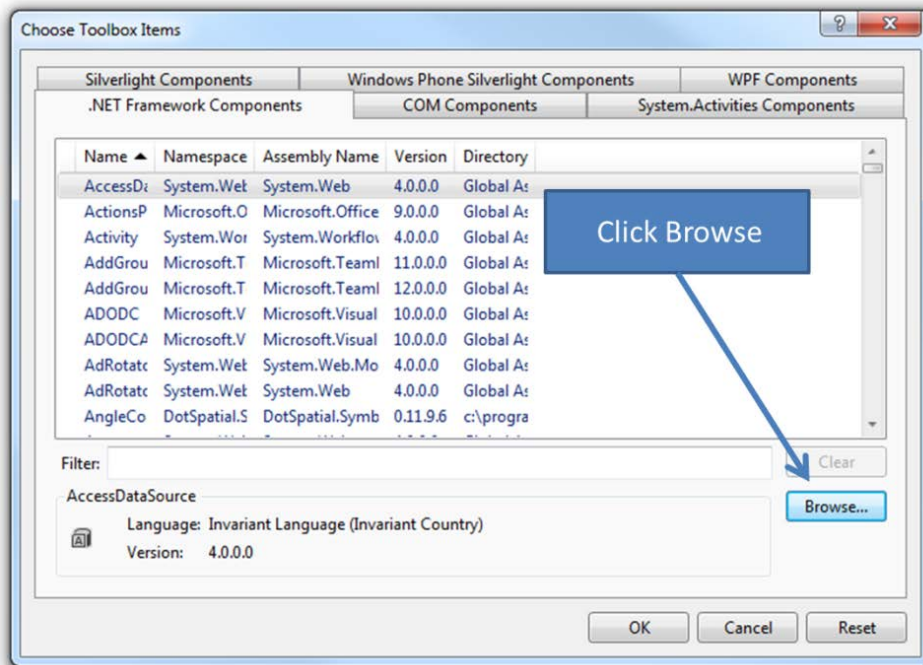


Figure 113: Toolbox Items

Browse once again to your DotSpatial folder and select “DotSpatial.Controls.dll”

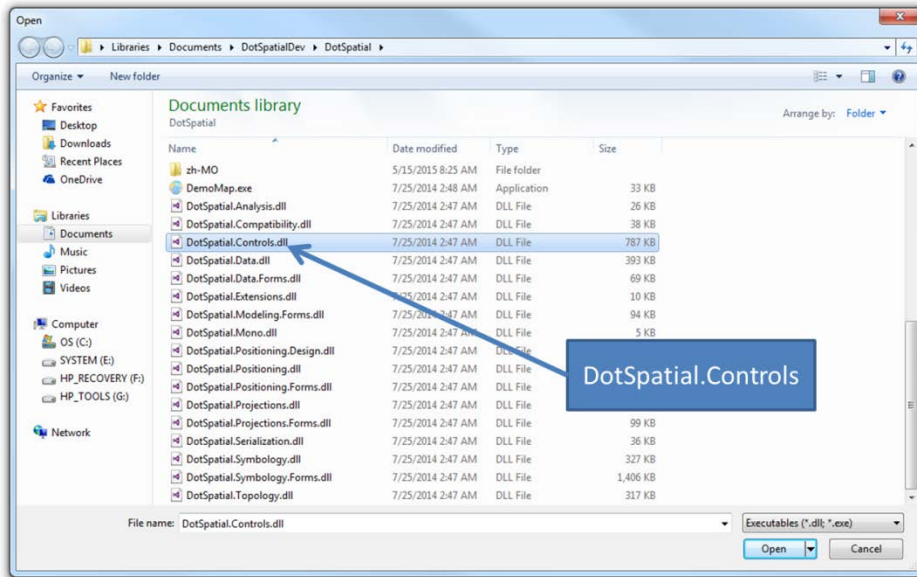


Figure 114: DotSpatial.Controls

Once you add the controls you should have access to the DotSpatial controls including the “AppManger.” You will need to drag and drop the manger unto the form.

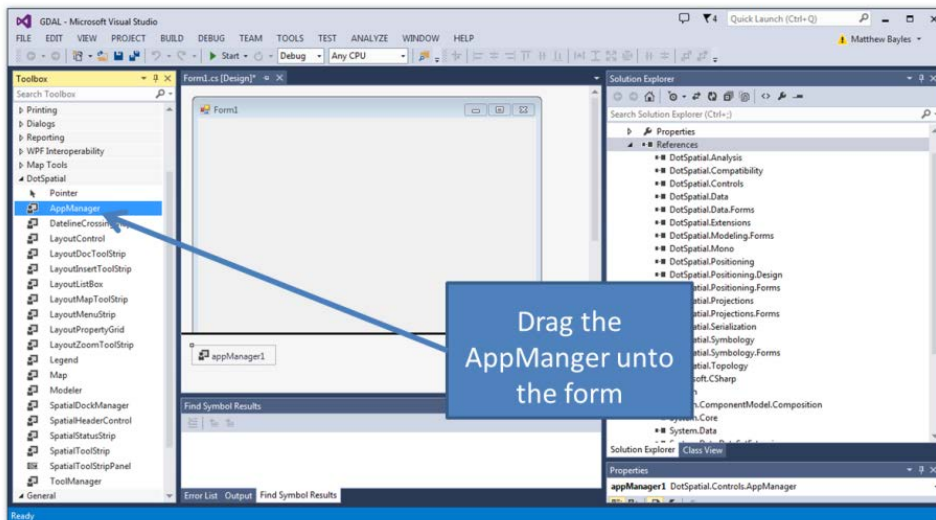


Figure 115: AppManger

In addition to the AppManger we will need to add a SpatialDockManger, Map, Legend, SpatialHeaderControl, SpatialStatusStrip, and a button so that we can add a layer to our map. Our project should now look like Figure 116.

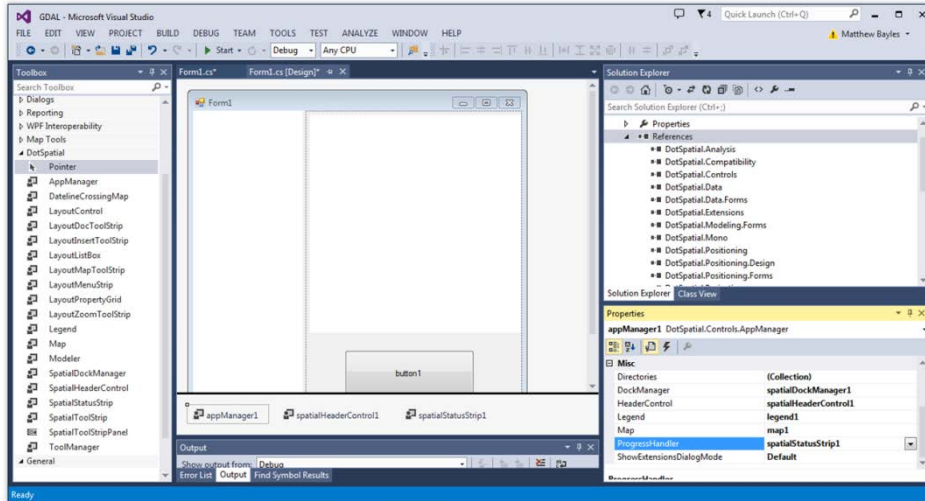


Figure 116: Adding Controls

Now that we have added all the controls we will need to link them to the AppManger. Each of the controls (except for the button) we just added, should link to the AppManger through the properties tab like in Figure 117. We also need to link our map to our legend through the properties tab as well.

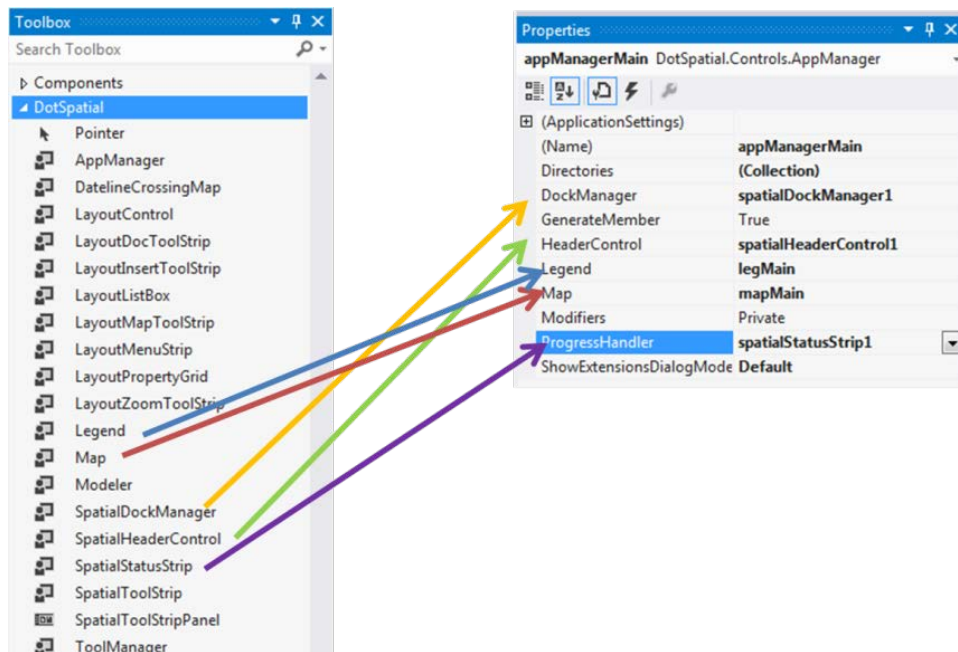


Figure 117: Linking Controls

Now that our controls are linked to our AppManger, we need to add some code. First we need to tell the AppManger to load extensions. We do that by adding “appManger1.LoadExtensions();” as seen on Figure 117. Next we need to add code to our button. Double clicking the button on the form page will allow us to tie code to a button click event. We want our button to add a layer to our map when clicked so we add “map1.AddLayer();” to our button click event.

```
namespace GDAL
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            appManager1.LoadExtensions();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            map1.AddLayer();
        }
    }
}
```

100 %

Output

Show output from: Debug

Error List Output Find Symbol Results

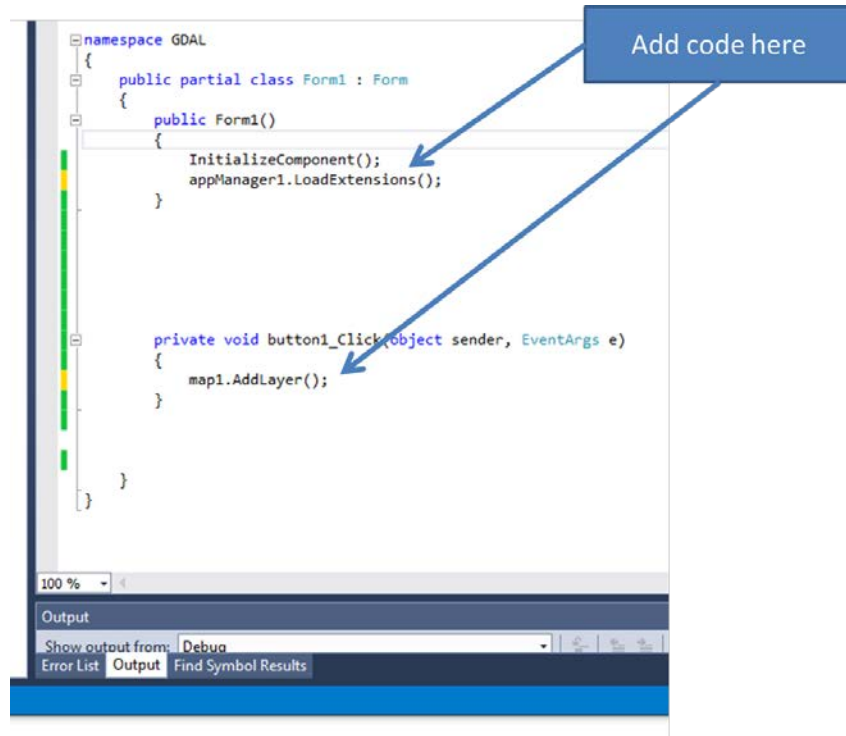


Figure 118: Adding Code

Now that we have added the code we need to create a new folder in our project. When we run our project the AppManger will now look for folders named “Plugins” and “Application Extensions” in our Visual Studio project. However, these folders do not yet exist. In the next step we will add them. Navigate to your bin/Debug folder in your project. On this computer the file path was C:\Users\Matt\Desktop\GDAL\GDAL\bin\Debug. In the Debug folder create a new folder named “Application Extensions.” We will copy the GDAL extension to this folder.

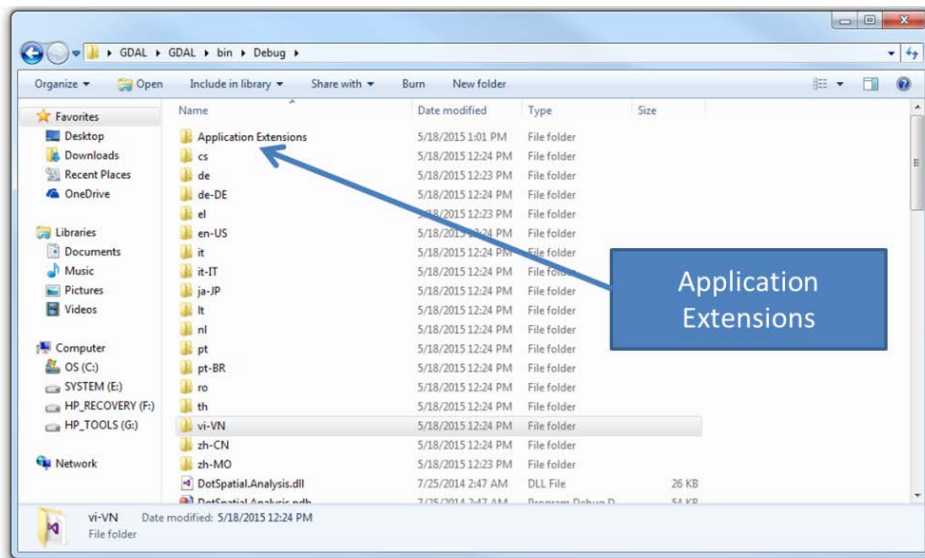


Figure 119: Application Extensions

1.10.2. New Ribbon

Objective:

Add a New Ribbon

One of the advantages of having DotSpatial support plugins is that we can easily add a more complex ribbon than the stock DotSpatial ribbon. It still has the same functionality but has the advantage of being more attractive and user friendly.

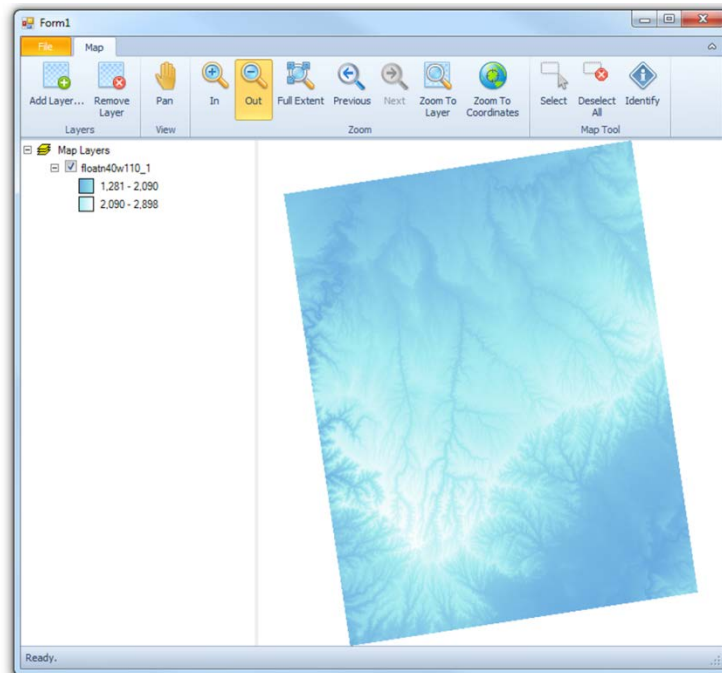


Figure 123: New Ribbon

If you would like to have this ribbon for your project you will first need to go to <http://dotspatialapp.codeplex.com> and download the project to your computer. Unzip and unblock the file. Next we will start a new Visual Studio project using a windows form. As we have done previously we will need to add the usual DotSpatial dlls in addition to four new ones. The DotSpatialApp that we just downloaded includes the DotSpatial dlls in addition to the new dlls we need.

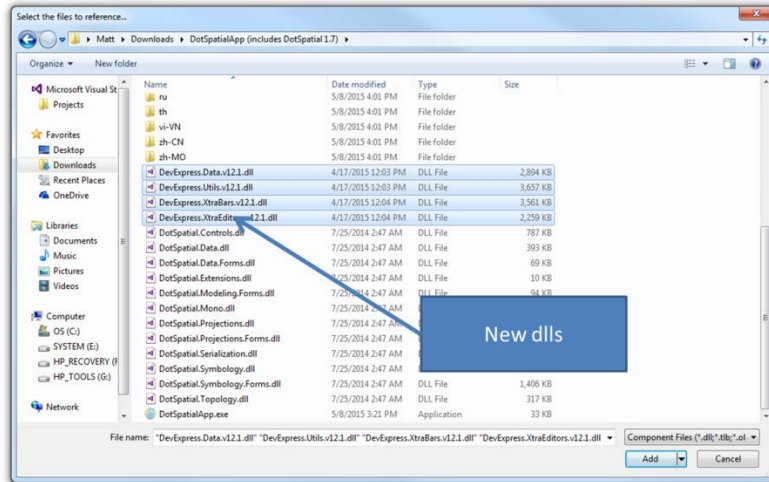


Figure 124: DevExpress dlls

We also need to include “System.ComponentModel.Composition” so that we can use the AppManger. Next we need to set up the user interface. Add a SpatialDockManger, Map, Legend, and an AppManger. Link the AppManger to the other controls as well as link the Map control to the Legend. Now that we have set up our user interface we can add some new code. We need to add the following using statements:

```
using System.ComponentModel.Composition;
using DotSpatial.Controls.Docking;
using DevExpress.XtraBars;
```

Next we need to add following code after our class declaration:

```
[Export("Shell", typeof(ContainerControl))]
private static ContainerControl Shell;
```

After the “InitializeComponent();” add the following:

```
if (DesignMode) return;
Shell = this;
appManager1.LoadExtensions();
this.appManager1.DockManager.Add( new DockablePanel( "kMap", "Map", map1, DockStyle.Fill ) );
new DotSpatial.Controls.DefaultMenuBars( appManager1 ).Initialize( appManager1.HeaderControl );
```

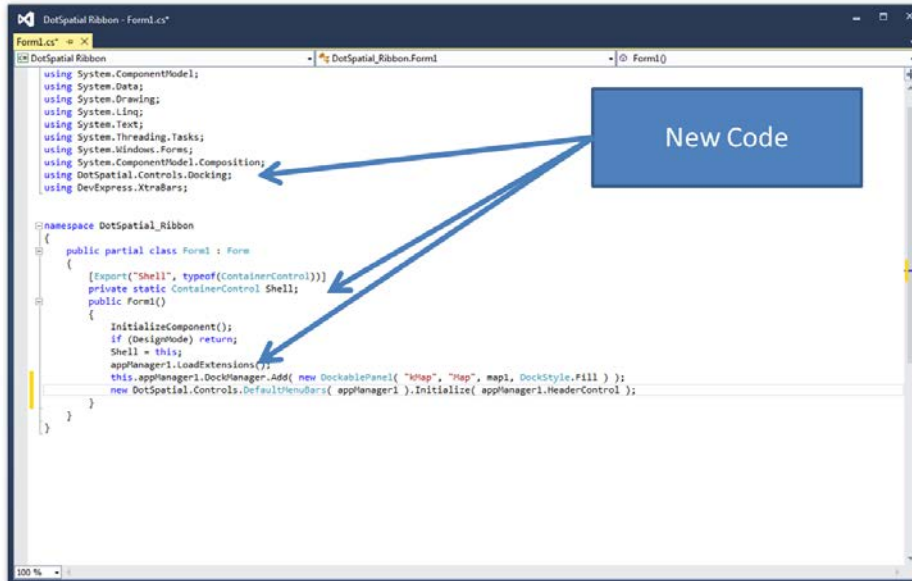



Figure 125: New Code

Now that we have the required code we just need to copy the ribbon extension to our “Application Extensions” folder in our projects bin/Debug folder. In the DotSpatialApp folder there is an “Application Extensions” folder. Inside this folder there is a “Ribbon” folder. Copy this folder to our project’s “Application Extensions” folder.

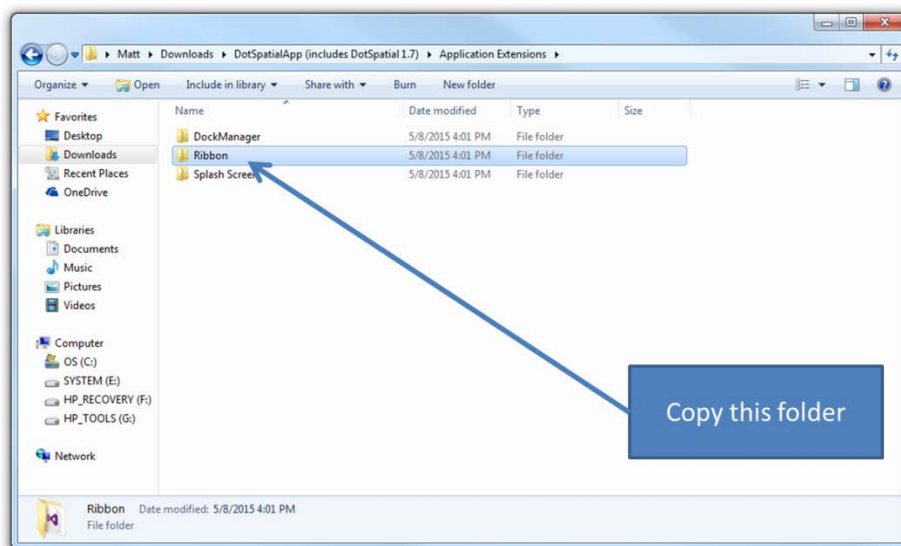


Figure 126: Ribbon Folder

Now when you run the project you should have a new ribbon.

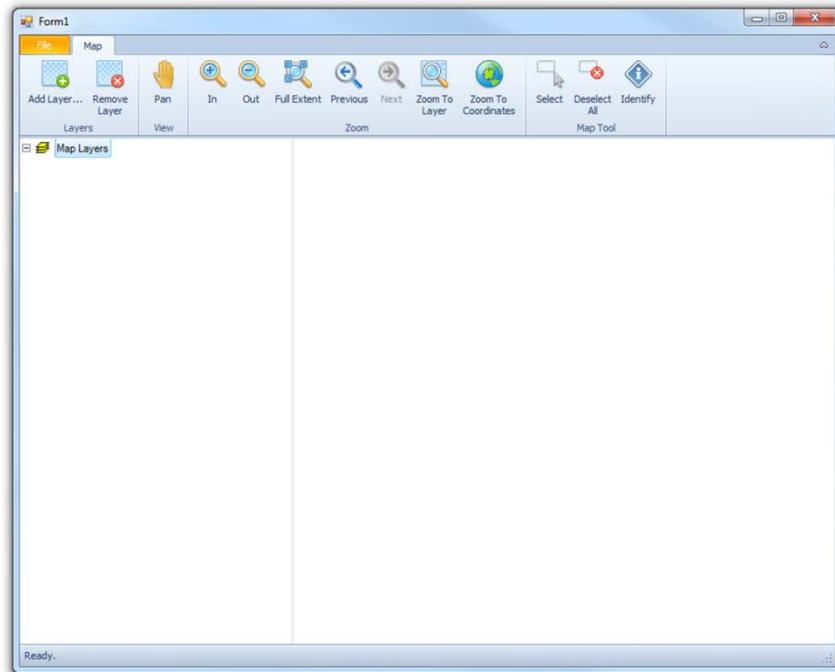


Figure 127: New Ribbon

2. Acknowledgements

We would like to thank the following for their contributions:

- Harold (Ted) Dunsford Jr.
- Mark Van
- Orden Jiří Kadlec