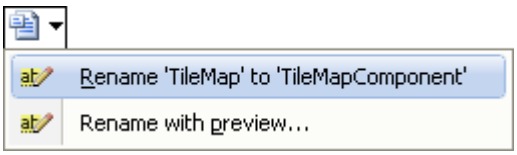
Creating a Role Playing Game with XNA Game Studio 3.0
Part 39
Refactoring - Part 2

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA 3.0 Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG38.zip You can download the graphics from this link: Graphics.zip

I just found out that I made a terrible mistake when I was creating the tile engine. I forgot that when you create the Content Pipeline project that there is no way to create an object of type **TileMap** with out passing in a **Game** object. The reason is because it inherits from **DrawableGameComponent** the base class requires a **Game** object as a parameter. Fortunately there is a way to fix this problem without too much fuss though it will be a bit of a pain.

The first thing you will want to is open the **TileEngine** folder in the solution explorer. Now you should right click the **TileMap.cs** file. Rename this to **TileMapComponent.cs**. This will change the file name of the class from **TileMap.cs** to **TileMapComponent.cs**. Now you need to change all instances of **TileMap** to **TileMapComponent**. Go to the code for the class. In the class declaration change **TileMap** to **TileMapComponent** then press **Shift + Alt + F10** all together. This will bring up this little pop up.



What you want to do is click the **Rename 'TileMap' to 'TileMapComponent'** option. This will rename all instances of **TileMap** to **TileMapComponent** in the project.

What I am going to do next is create a new class called **TileMap** to replace the other class. Right click the **TileEngine** folder in your game and add a new class called **TileMap**. As usual when I'm presenting a lot of code I will explain it after you have read it. This is the code for the **TileMap** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;


namespace New2DRPG
{
    class TileMap
    {
        List<TileMapLayer> mapLayers = new List<TileMapLayer>();
```

```
CollisionLayer collisionLayer;

int mapWidth;
int mapHeight;

public TileMap(int width, int height, List<TileMapLayer> layers)
{
    this.mapWidth = width;
    this.mapHeight = height;
    this.mapLayers = layers;
    collisionLayer = new CollisionLayer(mapWidth, mapHeight);
    foreach (TileMapLayer mapLayer in mapLayers)
    {
        ProcessCollisionLayer(mapLayer);
    }

}

public int MapWidth
{
    get { return mapWidth; }
}

public int MapHeight
{
    get { return mapHeight; }
}

private void ProcessCollisionLayer(TileMapLayer layer)
{
    for (int y = 0; y < mapHeight; y++)
        for (int x = 0; x < mapWidth; x++)
        {
            switch (layer.GetTile(x, y))
            {
                case 32:
                case 33:
                case 34:
                case 40:
                case 41:
                case 42:
                case 48:
                case 49:
                case 50:
                case 63:
                    collisionLayer.SetTile(x, y, CollisionType.Unpassable);
                    break;
            }
        }
}

public CollisionType GetCollisionValue(int x, int y)
{
    return collisionLayer.GetTile(x, y);
}

public bool CheckUpAndLeft(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
```

```
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width,
                        nextRectangle.Y + nextRectangle.Height));

        bool doesCollide = false;

        if (tile1.X < 0 || tile1.Y < 0)
            return !doesCollide;

        for (int y = tile1.Y; y <= tile2.Y; y++)
            for (int x = tile1.X; x <= tile2.X; x++)
                if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                    doesCollide = true;

        return doesCollide;
    }

    public bool CheckUp(Rectangle nextRectangle)
    {
        Point tile1 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width - 1,
                        nextRectangle.Y + nextRectangle.Height));

        bool doesCollide = false;

        if (tile1.Y < 0)
            return !doesCollide;

        int y = tile1.Y;
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

        return doesCollide;
    }

    public bool CheckUpAndRight(Rectangle nextRectangle)
    {
        Point tile1 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width + 1,
                        nextRectangle.Y + nextRectangle.Height));

        bool doesCollide = false;

        if (tile2.X >= mapWidth || tile1.Y < 0)
            return !doesCollide;

        for (int y = tile1.Y; y <= tile2.Y; y++)
            for (int x = tile1.X; x <= tile2.X; x++)
                if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                    doesCollide = true;

        return doesCollide;
    }
```

```
public bool CheckLeft(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height - 1));

    bool doesCollide = false;

    if (tile1.X < 0)
        return !doesCollide;

    int x = tile1.X;
    for (int y = tile1.Y; y <= tile2.Y; y++)
    {
        if (GetCollisionValue(x, y) == CollisionType.Unpassable)
            doesCollide = true;
    }

    return doesCollide;
}

public bool CheckRight(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height - 1));

    bool doesCollide = false;
    if (tile2.X >= mapWidth)
        return !doesCollide;

    int x = tile2.X;

    for (int y = tile1.Y; y <= tile2.Y; y++)
    {
        if (GetCollisionValue(x, y) == CollisionType.Unpassable)
            doesCollide = true;
    }

    return doesCollide;
}

public bool CheckDownAndLeft(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile1.X < 0 || tile2.Y >= mapHeight)
        return !doesCollide;
```

```
    for (int y = tile1.Y; y <= tile2.Y; y++)
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

    return doesCollide;
}

public bool CheckDown(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width - 1,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile2.Y >= mapHeight)
        return !doesCollide;

    int y = tile2.Y;
    for (int x = tile1.X; x <= tile2.X; x++)
        if (GetCollisionValue(x, y) == CollisionType.Unpassable)
            doesCollide = true;

    return doesCollide;
}

public bool CheckDownAndRight(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile2.X >= mapWidth || tile2.Y >= mapHeight)
        return !doesCollide;

    for (int y = tile1.Y; y <= tile2.Y; y++)
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

    return doesCollide;
}

public void Draw(SpriteBatch spriteBatch,
                 Texture2D texture, Tileset tileset)
{
    foreach (TileMapLayer layer in mapLayers)
    {
        layer.Draw(spriteBatch, texture, tileset);
    }
}
```

```
    }
}
```

Since this class has to do with XNA I added in using statements for the XNA Framework and XNA Framework Graphics classes. I have moved everything to do with collision detection with the tiles to this class. The fields I added to the class are: **mapLayers** which will be the list of layers for the map, **collisionLayer** which is for the collision layer of the map, and **mapWidth** and **mapHeight** are for the height and width of the map. I moved all of the functionality of checking for tile collisions into this class as well.

There is just the one constructor for this class. It takes as its parameters: **width**, **height** and **layers**. The first two are for the width and the height of the map. The last is the list of layers for the map. The constructor then sets the fields for the class from the parameters, creates the new collision layer object and calls the **ProcessCollisionLayer** method that I moved into this class. There are also two get only properties to access the **mapWidth** and **mapHeight** fields. There really isn't any thing you haven't seen before in the other methods. They just work with the collision layer and render the map.

The code for the **TileMapComponent** will have to change to reflect the new **TileMap** class. A lot of it changed so I will give you the code for the entire class. This is the code for the **TileMapComponent.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using New2DRPG.SpriteClasses;
using System.Xml;


namespace New2DRPG
{
    class TileMapComponent : Microsoft.Xna.Framework.DrawableGameComponent
    {
        TileMap tileMap;

        Random random = new Random();

        SpriteBatch spriteBatch;
        ContentManager Content;

        Tileset tileset;
        Texture2D texture;
        Texture2D itemTexture;

        int mapWidth;
        int mapHeight;
```

```
static int widthInPixels;
static int heightInPixels;

List<ItemSprite> items = new List<ItemSprite>();

public TileMapComponent(TileMap tileMap, Tileset tileset, Game game)
    : base(game)
{
    spriteBatch = Game1.TileSpriteBatch;
    Content =
        (ContentManager)Game.Services.GetService(typeof(ContentManager));
    this.tileMap = tileMap;
    this.tileset = tileset;
    mapWidth = tileMap.MapWidth;
    mapHeight = tileMap.MapHeight;
    LoadContent();
}

public static int WidthInPixels
{
    get { return widthInPixels; }
}

public static int HeightInPixels
{
    get { return heightInPixels; }
}

protected override void LoadContent()
{
    texture = Content.Load<Texture2D>(@"TileSets\" + tileset.TextureName);
    itemTexture = Content.Load<Texture2D>(@"Items\chest");
    base.LoadContent();
}

public override void Initialize()
{
    base.Initialize();
}

public override void Update(GameTime gameTime)
{
    widthInPixels = mapWidth * TileEngine.TileWidth;
    heightInPixels = mapHeight * TileEngine.TileHeight;

    base.Update(gameTime);
}

public override void Draw(GameTime gameTime)
{
    tileMap.Draw(spriteBatch, texture, tileset);

    foreach (ItemSprite item in items)
    {
        item.Draw(gameTime);
    }

    base.Draw(gameTime);
```

```
        }

        public void Hide()
        {
            Visible = false;
            Enabled = false;
        }

        public void Show()
        {
            Visible = true;
            Enabled = true;
        }
    }
}
```

I will just go over the changes briefly. The first is that the list of layers and the collision layer have been replaced with just one field, a **TileMap** object called **tileMap**. There is now just a single constructor. This constructor takes as its parameters: a **TileMap**, a **TileSet** and a **Game** object. The constructor just sets the various fields and calls the **LoadContent** method to load in the content for the component. The component is not responsible for dealing with the collision layer so everything that had to deal with that is not in the class. The last change is in the **Draw** method. Instead of drawing all of the layers in a foreach loop I now just call the **Draw** method of the **TileMap** class. You still have to do this before rendering any other objects or it will be drawn over top of them.

What needs to change next is the **ActionScreen**. What changed here is that there are new fields for a **TileMap** object named **tileMap** and a **TileMapComponent** called **tileMapComponent**. are a few of the fields, the constructor for the class and the **CheckForUnWalkableTiles** method. Because the **TileMap** class checks for collision with the tiles and the old **TileMap** class did and the new **TileMap** field has the same name as before I didn't need to update the **CheckForUnWalkableTiles** method. The changes to the constructor were it now takes as parameters: a **Game** object, a **SpriteFont** object, a **TileMap** object and string with the name of the texture for the tile set. The constructor works pretty much the same as before. The only real difference is that it now creates a **TileMapComponent** and adds it the list of components and calls the **Show** method to make sure when the **ActionScreen** is visible it will render the map.

```
SpriteFont gameFont;
SpriteFont interfaceFont;
Texture2D chest;
Texture2D characterHUDTexture;
string tilesetName;
Tileset tileset;
int viewportWidth;
int viewportHeight;
int screenWidth;
int screenHeight;

PlayerComponent player;

TileMapComponent tileMapComponent;
TileMap tileMap;

List<AnimatedSprite> animatedSprites = new List<AnimatedSprite>();
string[] assetNames =
    {
```

```
        @"Sprites\knt1",
        @"Sprites\nja1",
        @"Sprites\skl1" };
Texture2D[] spriteTextures;
Random random = new Random();

public ActionScreen(Game game, SpriteFont gameFont, TileMap tileMap, string
tilesetName)
    : base(game)
{
    player = new PlayerComponent(game, null, null);
    this.gameFont = gameFont;

    this.tilesetName = tilesetName;
    this.tileMap = tileMap;
    LoadContent();

    tileMapComponent = new TileMapComponent(tileMap, tileset, game);
    Components.Add(tileMapComponent);
    tileMapComponent.Show();

    viewportWidth = TileEngine.ViewPortWidth;
    viewportHeight = TileEngine.ViewPortHeight;
    screenWidth = game.Window.ClientBounds.Width;
    screenHeight = game.Window.ClientBounds.Height;
    CreateSprites(game);
}
```

Next I will add the Content Pipeline classes to the project. The first thing you will want to do is add a new folder to the **New2DRPGContent** called **TileEngine**. I will add the importer, processor and writer to this folder. The importer is the easiest of the three so I will add that first. Right click the **TileEngine** folder select **Add** and then **New Item**. Make sure that the **XNA Game Studio** node is selected on the left and choose **Content Importer** and call it **TilemapImporter**. This is the code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline;
using Microsoft.Xna.Framework.Content.Pipeline.Graphics;

using TImport = System.Xml.XmlDocument;

namespace New2DRPGContent.Tilemap
{
    [ContentImporter(".tmap", DisplayName = "Tile Map Importer", DefaultProcessor =
"TilemapProcessor")]
    public class TilemapImporter : ContentImporter<TImport>
    {
        public override TImport Import(string filename, ContentImporterContext
context)
        {
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load(filename);
            return xmlDoc;
        }
```

```
    }
}
```

When you create a new importer there is a using statement that says **TImport = System.String**. **TImport** is the return type of the importer. You don't have to use this but I decide to use it. Our importer reads in an XML document so I changed **System.String** to **System.Xml.XmlDocument**. I also added in a using statement of **System.Xml**.

The class uses the attribute **ContentImporterAttribute**. This attribute is applied to the class. It associates files with a **.tmap** file extension with the importer. The **DisplayName** parameter give the processor a friendly name. The **DefaultProcessor** parameter says that when a **tmap** file is found to use the **TilemapProcessor**.

The constructor takes as a parameter the **filename** of the file to be imported. The second parameter provides properties that define logging behaviour for the importer. Inside the constructor I create an instance of **XmlDocument**. I call the **Load** method of the document passing the **filename** parameter of the constructor. Finally I return the **XmlDocument**. That is all for the importer.

Before I get to the **Content Processor** I want to add two classes to the **TilemapContent** folder to hold the output of the **Content Processor**. I will include both of these classes in the same file as they are tied together. Right click the **TilemapContent** folder and add a new class called **TilemapContent**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace New2DRPGContent.Tilemap
{
    public class MapLayer
    {
        int[,] map;

        public MapLayer(int width, int height)
        {
            map = new int[height, width];
        }

        public void SetTile(int x, int y, int value)
        {
            map[y, x] = value;
        }

        public int GetTile(int x, int y)
        {
            return map[y, x];
        }
    }

    public class TilemapContent
    {
        public List<MapLayer> layers = new List<MapLayer>();
        public int mapWidth;
```

```
        public int mapHeight;
    }
}
```

This code should look familiar to you. It is bare bones needed to hold the map for the tile engine. The first class **MapLayer** is a 2D array of integers to hold the map. It also has **GetTile** and **SetTile** methods to get the tiles in the map. The second class has a **List<MapLayer>** to hold the layers of the map and fields for the width and the height of the map. Right click the **TilemapContent** folder and select **Add** and then **New Item**. Again, make sure that the **XNA Game Studio** node is selected and choose **Content Processor** and call it **TilemapProcessor**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline;
using Microsoft.Xna.Framework.Content.Pipeline.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline.Processors;

using TInput = System.Xml.XmlDocument;
using TOutput = New2DRPGContent.Tilemap.TilemapContent;

namespace New2DRPGContent.Tilemap
{
    [ContentProcessor(DisplayName = "Tile Map Processor")]
    public class TilemapProcessor : ContentProcessor<TInput, TOutput>
    {
        public override TOutput Process(TInput input, ContentProcessorContext
context)
        {
            TilemapContent tileMap = new TilemapContent();
            XmlNode rootNode = input.FirstChild;

            if (rootNode.Name != "TileMap")
            {
                throw new Exception("Invalid tile map format!");
            }

            tileMap.mapWidth = Int32.Parse(rootNode.Attributes["Width"].Value);
            tileMap.mapHeight = Int32.Parse(rootNode.Attributes["Height"].Value);

            XmlNode layersNode = rootNode.FirstChild;

            if (layersNode.Name != "Layers")
            {
                throw new Exception("Invalid tile map format!");
            }

            MapLayer layer;

            foreach (XmlNode node in layersNode.ChildNodes)
            {
                if (node.Name == "Layer")
                {
                    try
```

```
                    {
                        layer = new MapLayer(tileMap.mapWidth, tileMap.mapHeight);
                        LoadLayer(node, layer, tileMap.mapWidth);
                        tileMap.layers.Add(layer);
                    }
                    catch
                    {
                        throw new Exception("Error reading in map layer!");
                    }
                }
            }

            return tileMap;
        }

        private void LoadLayer(XmlNode layerNode, MapLayer layer, int mapWidth)
        {
            int rowCount = 0;

            foreach (XmlNode node in layerNode)
            {
                if (node.Name == "Row")
                {
                    string row = node.InnerText;
                    row.Trim();
                    string[] cells = row.Split(' ');
                    for (int i = 0; i < mapWidth; i++)
                        layer.SetTile(i, rowCount, Int32.Parse(cells[i]));
                    rowCount++;
                }
            }
        }

    }
}
```

I needed a using statement for XML. Like when you create a **Content Importer** the **Content Processor** creates a few using statements. **TInput = System.String** and **TOutput = System.String**. These are for the input to the processor and the output of the processor. I change the first one to **System.Xml.Xmldocument** because that is what the importer returns. For the second I used **New2DRPGContent.Tileset.TilemapContent** the class that I just wrote. The class has an attribute applied to it **ContentProcessor** this attribute has a parameter **DisplayName** which holds the friendly name for the processor.

The overload of the **Process** method is where I process the tile map. There isn't anything new here as it was simply copied and pasted from the game. The **Process** method was taken out of the **Tile Map Editor** line for line. The only change was instead that it uses **input** instead of **xmlDoc** to get the root node of the **XmlDocument**. There is also a **LoadLayer** method to load in the layers. Again, it is the same code you have already seen.

There is just one other class to add to this, the **Content Type Writer**. The job of the **Content Type Writer** is to write a binary **XNB** file. This will help protect your custom content from being hacked. For example if you have script files that are in XML format somebody can easily change the text of the XML files. That is one of the reasons why I chose to use these classes. Right click the **TilemapContent** folder, select **Add** and then **New Item**. Make sure that the **XNA Game Studio** node is selected on the

left and choose **Content Type Writer** and name it **TilemapWriter**. This is the code for that class.

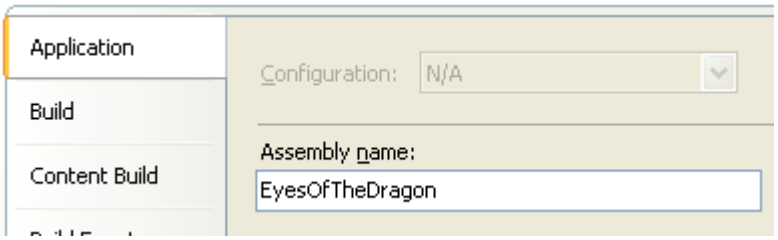
```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline;
using Microsoft.Xna.Framework.Content.Pipeline.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline.Processors;
using Microsoft.Xna.Framework.Content.Pipeline.Serialization.Compiler;

using TWrite = New2DRPGContent.Tilemap.TilemapContent;

namespace New2DRPGContent.Tilemap
{
   [ContentTypeWriter]
   public class TilemapWriter : ContentTypeWriter<TWrite>
   {
      protected override void Write(ContentWriter output, TWrite value)
      {
         output.Write(value.mapWidth);
         output.Write(value.mapHeight);
         output.Write(value.layers.Count);
         foreach (MapLayer layer in value.layers)
            WriteLayer(output, layer, value.mapWidth, value.mapHeight);
      }

      private void WriteLayer(ContentWriter output, MapLayer layer, int width, int height)
      {
         for (int y = 0; y < height; y++)
            for (int x = 0; x < width; x++)
               output.Write(layer.GetTile(x, y));
      }

      public override string GetRuntimeReader(TargetPlatform targetPlatform)
      {
         return "New2DRPG.TilemapReader, EyesOfTheDragon";
      }
   }
}
```

There is yet another using statement in this class. This one is called **TWrite**. It is the type that the **Content Processor** returned. There is an attribute applied to this class. You must apply the attribute as well as extend the **ContentTypeWriter** class. There are two methods in this class. The first one is an override of the **Write** method. This is the method you will use to write out the **XNB** file. It takes as a parameter a **ContentWriter** object that is used to write the **XNB** file and a parameter **TWrite**. This parameter is the values from the **Content Processor**. The second method is called **GetRuntimeReader**. This method has a parameter an enum of type **TargetPlatform**. This paremeter holds the platform the game is targeting.

In the **Write** method I write out the values from the **value** parameter. It is important to note the order that you write things out because in the **Content Type Reader** you will need to read them in the same order.

I first write out the width and the height of the map. Next I write out the number of layers in the map. This will just make reading in the layers easier. There is then a foreach loop that loops through all of the layers and calls the **WriteLayer** method. The order of the loops in the **WriteLayer** method is important. The outer loop is for the **Y** index of the tiles and the inner loop is for the **X** index of the tiles. You will have to use the same order when you read them in the **Content Type Reader** method for the same reason I already mentioned.

The **GetRuntimeReader** method is used to determine what the name of the **Content Type Reader** is. There was a little confusion on some in the tutorial that I implemented the **Content Type Reader** for reading in the tile sets. The first part is the name of the **Content Type Reader**. I will be calling it **TilemapReader** and it will be in the **New2DRPG** namespace so that goes before the **TilemapReader** part. The next part is what confused a few people. This is the name of the assembly the **TilemapReader** is in. In simple terms an assembly is what your game will be packaged in. I have mine as **EyesOfTheDragon**. To find out what yours is right click the game in the solution explorer and select **Properties**. Under the **Application** tab there will be an entry **Assembly name**. If this is not **EyesOfTheDragon** you will have to replace **EyesOfTheDragon** in the **GetRuntimeReader** method what is there. This is a clipping of my **Properties**.



Now it is time to implement the **Content Type Reader**. Unlike the other **Content Pipeline** classes you implement these in your game. I will be adding the **Content Type Reader** to the **TileEngine** folder of the game. Right click the **TileEngine** folder in the game select **Add** and the **New Item**. Again make sure that the **XNA Game Studio** node is selected and choose **Content Type Reader** and name it **TilemapReader**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

using TRead = New2DRPG.TileMap;

namespace New2DRPG
{
    class TilemapReader : ContentTypeReader<TRead>
    {
        protected override TRead Read(ContentReader input, TRead existingInstance)
        {
            List<TileMapLayer> layers = new List<TileMapLayer>();
            int mapWidth = input.ReadInt32();
            int mapHeight = input.ReadInt32();
            int layerCount = input.ReadInt32();
            for (int i = 0; i < layerCount; i++)
```

```
            {
                TileMapLayer layer = new TileMapLayer(mapWidth, mapHeight);
                for (int y = 0; y < mapHeight; y++)
                    for (int x = 0; x < mapWidth; x++)
                        layer.SetTile(x, y, input.ReadInt32());
                layers.Add(layer);
            }

            return new TRead(mapWidth, mapHeight, layers);
        }
    }
}
```

What this does is actually read in a **TileMap** in **XNB** format. There is a using statement that will hold the type that the **Content Type Reader** will return. It will return a **TileMap** object. The constructor for the **TileMap** class has as parameters: the width of the map, the height of the map and a list of layers so I created a list to hold the layers. I then read in in order, the width of the map, the height of the map, the number of layers and then the layers of the map. I then return a new map.

Almost done, there are two things left to do. The first thing is to change the **LoadActionScreen** method in the **Game1** class because I changed the constructor for the **ActionScreen** class. The **ActionScreen** class now takes as its parameters: the current **Game** object, a **SpriteFont** object, a **TileMap** object and the name of the **Tileset** for the map. This is the code for that method.

```
private void LoadActionScreen()
{
    actionScreen = new ActionScreen(this,
        normalFont,
        Content.Load<TileMap>(@"TileMaps\tilemap"),
        "tileset1");

    Components.Add(actionScreen);
    actionScreen.Hide();
}
```

Before you can continue you will need to build the project. As you might of guessed from the code of the method I added a new folder to the **Content** folder called **TileMaps** that will hold the tile maps for the game. Right click the **Content** folder and add a new folder called **TileMaps**. To this folder select **Add** and then **Existing Item**. In the **Objects of type** combo box you will have to select **All files**. Navigate to the **Tilesets** folder in the game and select **tilemap.tmap** to add the map to the **TileMaps** folder. You can go to the **Tilesets** folder in the **Content** folder and right click **tilemap.tmap** and select **Exclude from project**.

That is it for this tutorial. I am already onto the next part so I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on my tutorials.