Eyes of the Dragon Tutorials
Part 41
Spells and Talents

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

This tutorial is going to create some spells and talents that can be used in the game. I will be doing it through code rather than another editor. I think you've had enough of editors for a while so I will try and focus on game play for a while. To get started, I want to add cool downs to talents and skills. For now I will focus on the SpellData and TalentData classes. Replace the TalentData class and SpellData classes with the following.

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

namespace RpgLibrary.TalentClasses
{
    public enum TalentType { Passive, Sustained, Activated }

    public class TalentData
    {
        #region Field Region

        public string Name;
        public string[] AllowedClasses;
        public Dictionary<string, int> AttributeRequirements;
        public string[] TalentPrerequisites;
        public int LevelRequirement;
        public TalentType TalentType;
        public int ActivationCost;
        public int CoolDown;
        public string[] Effects;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public TalentData()
        {
            AttributeRequirements = new Dictionary<string, int>();
        }

        #endregion

        #region Method Region
        #endregion
```

```
        #region Virtual Method region

        public override string ToString()
        {
            string toString = Name;

            foreach (string s in AllowedClasses)
                toString += ", " + s;

            foreach (string s in AttributeRequirements.Keys)
                toString += ", " + s + "+" + AttributeRequirements[s].ToString();

            foreach (string s in TalentPrerequisites)
                toString += ", " + s;

            toString += ", " + LevelRequirement.ToString();
            toString += ", " + TalentType.ToString();
            toString += ", " + ActivationCost.ToString();
            toString += ", " + CoolDown.ToString();

            foreach (string s in Effects)
                toString += ", " + s;

            return toString;
        }

        #endregion
    }
}

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

namespace RpgLibrary.SpellClasses
{
    public enum SpellType { Passive, Sustained, Activated }

    public class SpellData
    {
        #region Field Region

        public string Name;
        public string[] AllowedClasses;
        public Dictionary<string, int> AttributeRequirements;
        public string[] SpellPrerequisites;
        public int LevelRequirement;
        public SpellType SpellType;
        public int ActivationCost;
        public int CoolDown;
        public string[] Effects;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
```

```
        public SpellData()
        {
            AttributeRequirements = new Dictionary<string, int>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

        public override string ToString()
        {
            string toString = Name;

            foreach (string s in AllowedClasses)
                toString += ", " + s;

            foreach (string s in AttributeRequirements.Keys)
                toString += ", " + s + "+" + AttributeRequirements[s].ToString();

            foreach (string s in SpellPrerequisites)
                toString += ", " + s;

            toString += ", " + LevelRequirement.ToString();
            toString += ", " + SpellType.ToString();
            toString += ", " + ActivationCost.ToString();
            toString += ", " + CoolDown.ToString();

            foreach (string s in Effects)
                toString += ", " + s;

            return toString;
        }

        #endregion
    }
}
```

The change was the same in both classes. I added a field CoolDown that is the number of turns it takes for using a spell or talent for it to be ready again. I also updated the ToString methods to write out the CoolDown fields.

Now I'm going to add some talents. Open the TalentDataManager class and replace the contents with the following.

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

namespace RpgLibrary.TalentClasses
{
    public class TalentDataManager
    {
```

```
#region Field Region

readonly Dictionary<string, TalentData> talentData;

#endregion

#region Property Region

public Dictionary<string, TalentData> TalentData
{
    get { return talentData; }
}

#endregion

#region Constructor Region

public TalentDataManager()
{
    talentData = new Dictionary<string, TalentData>();
}

#endregion

#region Method Region

public void FillTalents()
{
    TalentData data = new TalentData
    {
        Name = "Bash",
        TalentPrerequisites = new string[0],
        LevelRequirement = 1,
        TalentType = TalentType.Activated,
        ActivationCost = 5,
        AllowedClasses = new string[] { "Warrior" },
        AttributeRequirements = new Dictionary<string, int>() { { "Strength", 10 } }
    };

    DamageEffectData effect = new DamageEffectData
    {
        AttackType = AttackType.Health,
        DamageType = DamageType.Crushing,
        DieType = DieType.D8,
        Modifier = 2,
        Name = "Bash",
        NumberOfDice = 2
    };

    data.Effects = new string[] { effect.ToString() };

    talentData.Add("Bash", data);

    data = new TalentData()
    {
        Name = "Below The Belt",
        TalentPrerequisites = new string[0],
        LevelRequirement = 1,
        TalentType = TalentType.Activated,
        ActivationCost = 5,
```

```
                AllowedClasses = new string[] { "Rogue" },
                AttributeRequirements = new Dictionary<string, int>() { { "Dexterity", 10 } }
            };

            effect = new DamageEffectData()
            {
                AttackType = AttackType.Health,
                DamageType = DamageType.Piercing,
                DieType = DieType.D4,
                Modifier = 2,
                Name = "Below The Belt",
                NumberOfDice = 3
            };

            data.Effects = new string[] { effect.ToString() };

            talentData.Add("Below The Belt", data);
        }

        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

What is new here is I added a method FillTalents that fills the Dictionary of talents with some values. In that method I create a TalentData object. I set the Name to Bash. There are no prerequisites so that array is set to an empty array. It is a level one talent so the LevelRequirement is set to 1. It is an activated talent with a cost of 5 stamina. It is a warrior talent and you need a minimum strength of 10, which all characters start with. I then create a DamageEffectData for the talent. It attacks health. It is crushing damage. The die type is D8 with a modifier of 2. The name of the effect is Bash and the number of dice is 2. I then add the effect to the array of effects. The talent is then added to the Dictionary of talents.

I then repeat the process for a new talent. This is a Rogue talent called Below The Belt. It has no talent prerequisites either. It is a level one talent that is activated with an activation cost of 5 stamina. As I mentioned this is a Rogue talent. It has a prerequisite of 10 Dexterity, which all characters have. I then create another effect the same way as before with new values. It is 3 D4 effect with a modifier of 2 for 5 to 14 damage. It is also piercing instead of crushing. The effect is added to the array of effects and the talent is added to the Dictionary of talents.

I did the same thing with the SpellDataManager. Replace that class with the following code.

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

namespace RpgLibrary.SpellClasses
{
    public class SpellDataManager
    {
```

```
#region Field Region

readonly Dictionary<string, SpellData> spellData;

#endregion

#region Property Region

public Dictionary<string, SpellData> SpellData
{
    get { return spellData; }
}

public object EffectData { get; private set; }

#endregion

#region Constructor Region

public SpellDataManager()
{
    spellData = new Dictionary<string, SpellData>();
}

#endregion

#region Method Region

public void FillSpellData()
{
    SpellData data = new SpellData()
    {
        Name = "Spark Jolt",
        SpellPrerequisites = new string[0],
        SpellType = SpellType.Activated,
        LevelRequirement = 1,
        ActivationCost = 8,
        AllowedClasses = new string[] { "Wizard" },
        AttributeRequirements = new Dictionary<string, int>() { { "Magic", 10 } }
    };

    DamageEffectData effect = new DamageEffectData
    {
        Name = "Spark Jolt",
        AttackType = AttackType.Health,
        DamageType = DamageType.Air,
        DieType = DieType.D6,
        NumberOfDice = 3,
        Modifier = 2
    };

    data.Effects = new string[] { effect.ToString() };
    spellData.Add("Spark Jolt", data);

    data = new SpellData()
    {
        Name = "Mend",
        SpellPrerequisites = new string[0],
        SpellType = SpellType.Activated,
        LevelRequirement = 1,
```

```
                ActivationCost = 6,
                AllowedClasses = new string[] { "Priest" },
                AttributeRequirements = new Dictionary<string, int>() { { "Magic", 10 } }
            };

            HealEffectData healEffect = new HealEffectData()
            {
                Name = "Mend",
                HealType = HealType.Health,
                DieType = DieType.D8,
                NumberOfDice = 2,
                Modifier = 2
            };

            data.Effects = new string[] { healEffect.ToString() };
            spellData.Add("Mend", data);
        }

        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

The method creates two new spells and adds them to the Dictionary of spells the same way as I did for the talents. The first spell, Spark Jolt, is for wizards and is a damage type spell. The second spell, Mend, is a healing spell and is for priests.

The next step is to call the methods and store their values so that they can be used. They need to be global so that they can be used anywhere within the game. Ideally, the player will be able to choose between three or four talents/spells when they create a new character. For now we will assign them one when they start the game. What I did was add to the DataManager class in the EyesOfTheDragon project. Replace that class with the following class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using RpgLibrary.CharacterClasses;
using RpgLibrary.ItemClasses;
using RpgLibrary.SkillClasses;
using RpgLibrary.SpellClasses;
using RpgLibrary.TalentClasses;
using RpgLibrary.TrapClasses;

namespace EyesOfTheDragon.Components
{
    static class DataManager
    {
        #region Field Region

        static readonly Dictionary<string, ArmorData> armor = new Dictionary<string,
ArmorData>();
```

```
        static readonly Dictionary<string, WeaponData> weapons = new Dictionary<string,
WeaponData>();
        static readonly Dictionary<string, ShieldData> shields = new Dictionary<string,
ShieldData>();
        static readonly Dictionary<string, KeyData> keys = new Dictionary<string, KeyData>();
        static readonly Dictionary<string, ChestData> chests = new Dictionary<string,
ChestData>();
        static readonly Dictionary<string, EntityData> entities = new Dictionary<string,
EntityData>();
        static readonly Dictionary<string, SkillData> skills = new Dictionary<string,
SkillData>();
        static readonly SpellDataManager SpellDataManager = new SpellDataManager();
        static readonly TalentDataManager TalentDataManager = new TalentDataManager();

        #endregion

        #region Property Region

        public static Dictionary<string, ArmorData> ArmorData
        {
            get { return armor; }
        }

        public static Dictionary<string, WeaponData> WeaponData
        {
            get { return weapons; }
        }

        public static Dictionary<string, ShieldData> ShieldData
        {
            get { return shields; }
        }

        public static Dictionary<string, EntityData> EntityData
        {
            get { return entities; }
        }

        public static Dictionary<string, ChestData> ChestData
        {
            get { return chests; }
        }

        public static Dictionary<string, KeyData> KeyData
        {
            get { return keys; }
        }

        public static Dictionary<string, SkillData> SkillData
        {
            get { return skills; }
        }

        #endregion

        #region Constructor Region

        static DataManager()
        {
            SpellDataManager.FillSpellData();
```

```
            TalentDataManager.FillTalents();
        }
        #endregion

        #region Method Region

        public static void ReadEntityData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Classes", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Classes\" + Path.GetFileNameWithoutExtension(name);

                EntityData data = Content.Load<EntityData>(filename);
                EntityData.Add(data.EntityName, data);
            }
        }

        public static void ReadArmorData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Armor", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Armor\" +
Path.GetFileNameWithoutExtension(name);

                ArmorData data = Content.Load<ArmorData>(filename);
                ArmorData.Add(data.Name, data);
            }
        }

        public static void ReadWeaponData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Weapon", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Weapon\" +
Path.GetFileNameWithoutExtension(name);

                WeaponData data = Content.Load<WeaponData>(filename);
                WeaponData.Add(data.Name, data);
            }
        }

        public static void ReadShieldData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Items\Shield", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Items\Shield\" +
Path.GetFileNameWithoutExtension(name);

                ShieldData data = Content.Load<ShieldData>(filename);
                ShieldData.Add(data.Name, data);
            }
        }
```

```
        public static void ReadKeyData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Keys", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Keys\" + Path.GetFileNameWithoutExtension(name);

                KeyData data = Content.Load<KeyData>(filename);
                KeyData.Add(data.Name, data);
            }
        }

        public static void ReadChestData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Chests", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Chests\" + Path.GetFileNameWithoutExtension(name);

                ChestData data = Content.Load<ChestData>(filename);
                ChestData.Add(data.Name, data);
            }
        }

        public static void ReadSkillData(ContentManager Content)
        {
            string[] filenames = Directory.GetFiles(@"Content\Game\Skills", "*.xnb");

            foreach (string name in filenames)
            {
                string filename = @"Game\Skills\" + Path.GetFileNameWithoutExtension(name);
                SkillData data = Content.Load<SkillData>(filename);
                SkillData.Add(data.Name, data);
            }
        }

        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

What I did was update all of the fields so that they are readonly because Visual Studio was complaining about it. I also added fields for a TalentDataManager and a SpellDataManager to the class. The last thing that I did was add a constructor to the class that calls the FillSpellData and FillTalents methods to fill the data managers that were just added.

Now I'm going to update the Spell and Talent classes to handle cool downs. Replace the Spell and Talent classes with these new versions.

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

```
using RpgLibrary.CharacterClasses;

namespace RpgLibrary.SpellClasses
{
    public class Spell
    {
        #region Field Region

        string name;
        List<string> allowedClasses;
        Dictionary<string, int> attributeRequirements;
        List<string> spellPrerequisites;
        int levelRequirement;
        SpellType spellType;
        int activationCost;
        int coolDown;
        List<string> effects;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public List<string> AllowedClasses
        {
            get { return allowedClasses; }
        }

        public Dictionary<string, int> AttributeRequirements
        {
            get { return attributeRequirements; }
        }

        public List<string> SpellPrerequisites
        {
            get { return spellPrerequisites; }
        }

        public int LevelRequirement
        {
            get { return levelRequirement; }
        }

        public SpellType SpellType
        {
            get { return spellType; }
        }

        public int ActivationCost
        {
            get { return activationCost; }
        }

        public int CoolDown
        {
            get { return coolDown; }
```

```
}

public List<string> Effects
{
    get { return effects; }
}

#endregion

#region Constructor Region

private Spell()
{
    allowedClasses = new List<string>();
    attributeRequirements = new Dictionary<string, int>();
    spellPrerequisites = new List<string>();
    effects = new List<string>();
}

#endregion

#region Method Region

public static Spell FromSpellData(SpellData data)
{
    Spell spell = new Spell
    {
        name = data.Name,
        levelRequirement = data.LevelRequirement,
        spellType = data.SpellType,
        activationCost = data.ActivationCost,
        coolDown = data.CoolDown
    };

    foreach (string s in data.AllowedClasses)
        spell.allowedClasses.Add(s.ToLower());

    foreach (string s in data.AttributeRequirements.Keys)
        spell.attributeRequirements.Add(
        s.ToLower(),
        data.AttributeRequirements[s]);

    foreach (string s in data.SpellPrerequisites)
        spell.SpellPrerequisites.Add(s);

    foreach (string s in data.Effects)
        spell.Effects.Add(s);

    return spell;
}

public static bool CanLearn(Entity entity, Spell spell)
{
    bool canLearn = true;

    if (entity.Level < spell.LevelRequirement)
        canLearn = false;

    string entityClass = entity.EntityClass.ToLower();
```

```
            if (!spell.AllowedClasses.Contains(entityClass))
                canLearn = false;

            foreach (string s in spell.AttributeRequirements.Keys)
            {
                if (Mechanics.GetAttributeByString(entity, s) < spell.AttributeRequirements[s])
                {
                    canLearn = false;
                    break;
                }
            }

            foreach (string s in spell.SpellPrerequisites)
            {
                if (!entity.Spells.ContainsKey(s))
                {
                    canLearn = false;
                    break;
                }
            }

            return canLearn;
        }

        #endregion

        #region Virtual Method Region
        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RpgLibrary.CharacterClasses;
namespace RpgLibrary.TalentClasses
{
    public class Talent
    {
        #region Field Region

        string name;
        List<string> allowedClasses;
        Dictionary<string, int> attributeRequirements;
        List<string> talentPrerequisites;
        int levelRequirement;
        TalentType talentType;
        int activationCost;
        int coolDown;
        List<string> effects;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }
```

```
public List<string> AllowedClasses
{
    get { return allowedClasses; }
}

public Dictionary<string, int> AttributeRequirements
{
    get { return attributeRequirements; }
}

public List<string> TalentPrerequisites
{
    get { return talentPrerequisites; }
}

public int LevelRequirement
{
    get { return levelRequirement; }
}

public TalentType TalentType
{
    get { return talentType; }
}

public int ActivationCost
{
    get { return activationCost; }
}

public int CoolDown
{
    get { return coolDown; }
}

public List<string> Effects
{
    get { return effects; }
}

#endregion

#region Constructor Region

private Talent()
{
    allowedClasses = new List<string>();
    attributeRequirements = new Dictionary<string, int>();
    talentPrerequisites = new List<string>();
    effects = new List<string>();
}

#endregion

#region Method Region

public static Talent FromTalentData(TalentData data)
{
    Talent talent = new Talent
```

```
            {
                name = data.Name,
                levelRequirement = data.LevelRequirement,
                talentType = data.TalentType,
                activationCost = data.ActivationCost,
                coolDown = data.CoolDown
            };

            foreach (string s in data.AllowedClasses)
                talent.allowedClasses.Add(s.ToLower());

            foreach (string s in data.AttributeRequirements.Keys)
                talent.attributeRequirements.Add(
                    s.ToLower(),
                    data.AttributeRequirements[s]);

            foreach (string s in data.TalentPrerequisites)
                talent.talentPrerequisites.Add(s);

            foreach (string s in data.Effects)
                talent.Effects.Add(s);
            return talent;

        }
        public static bool CanLearn(Entity entity, Talent talent)
        {
            bool canLearn = true;

            if (entity.Level < talent.LevelRequirement)
                canLearn = false;

            string entityClass = entity.EntityClass.ToLower();

            if (!talent.AllowedClasses.Contains(entityClass))
                canLearn = false;

            foreach (string s in talent.AttributeRequirements.Keys)
            {
                if (Mechanics.GetAttributeByString(entity, s) <
talent.AttributeRequirements[s])
                {
                    canLearn = false;
                    break;
                }
            }

            foreach (string s in talent.TalentPrerequisites)
            {
                if (!entity.Talents.ContainsKey(s))
                {
                    canLearn = false;
                    break;
                }
            }

            return canLearn;
        }

        #endregion
```

```
        #region Virtual Method Region
        #endregion
    }
}
```

If you build and run now everything will work as before. There will be no visible difference but we have the first piece for the battle engine in place. The next piece will be done in two parts. That is buying items from a merchant and then equipping items from inventory. I might combine the two parts into one depending on how long merchants takes. If it's not two long I will combine the two.

I'm going to wrap up the tutorial here because I don't want to something new at this point. I encourage you to visit my blog at, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials and other goodness.

Good luck in your Game Programming Adventures!

Cynthia