XNA 4.0 RPG Tutorials

Part 28

Spells and Effects

I'm writing these tutorials for the new XNA 4.0 framework. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the XNA 4.0 RPG tutorials page of my web site. I will be making my version of the project available for download at the end of each tutorial. It will be included on the page that links to the tutorials.

In this tutorial I'm going to add some meat to the classes related to spells and effects. Spells are similar to talents. I probably could have combined them into a single set of classes rather than dividing them into two classes. I did it because I see talents as different than spells. I don't like the idea of a warrior casting a spell. I do like the idea of a warrior having a special attack where he bashes an enemy with their shield.

First, let's update the **SpellData** class. Change the code for the **SpellData** tclass o the following.

```
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 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();

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

            return toString;
        }

        #endregion
    }
}
```

Everything there should look familiar from that last tutorial. There are several fields in the **SpellData** class. The first is **Name**, the name of the spell. Next is an array of strings, **AllowedClasses**, that holds the classes that can learn the spell. I decided to go this route in the case where more than one class may learn a specific spell rather than have two spells that achieve the same result. Next there is a dictionary with string keys and integer values, **AttributeRequirements**, that holds any attribute values that a character must have to learn the spell. There is then an array of strings, **SpellPrerequisites**, that will hold any spells that must be learned before the spell can be learned. The next field, **LevelRequirement**, will hold what level a character must be to learn the spell. In this way a low level character can't learn a very powerful spell that will unbalance your game, whereas a high level character may need that spell against the stronger foes. The field **SpellType** is the type of talent, whether it is passive, sustained, or activated. I decided to use a passive spell what really isn't a spell. A wizard may, for example, learn a spell that increases the damage they do. The next field, **ActivationCost**, is the cost required to activate the spell. The last, **Effects**, is an array of strings that holds the effects that the spell may cause. Spells will have a variety of effects like causing damage, healing, and a variety of other effects.

The constructor just creates a new **Dictionary<string, int>** for the **AttributeRequirements** field. The one method, **ToString**, just creates a string that represents a **SpellData** object and returns it.

Now for the **Spell** class. Change the **Spell** class to the following.

```
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;
        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 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();

            spell.name = data.Name;

            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);

            spell.levelRequirement = data.LevelRequirement;
            spell.spellType = data.SpellType;
            spell.activationCost = data.ActivationCost;

            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
    }
}
```

Again that should look very familiar because it is basically the same code as the **Talent** class. First thing that I did was include a using statement to bring the **Entity** class into scope in this class. There are fields that match the fields from the **SpellData** class as well as get only properties to expose their values. Instead of arrays I use the generic **List<T>** collection to hold the values. There is a private constructor that takes no parameters. It creates new collections for the fields that are collections. You don't use a constructor to create **Spell** objects, instead you use the static method **FromSpellData**.

The **FromSpellData** method takes a **SpellData** parameter called **data**. The first step is to create a new **Spell** object using the private constructor. In a foreach loop I then loop through all of the entries in the **AllowedClasses** field of the **SpellData** and add each item converted to a lower case string to the **allowedClasses** field of the **Spell** object. I loop over the keys in the **AttributeRequirements** in **data** and add the key converted to a lower case string with the value to **attributeRequirements** in **spell**. I do the same with **SpellPrerequisites** and **spellPrerequisites** for **data** and **spell**. There are then three straight assignments from **data** to **spell**. There is one last loop that assigns values from the **Effects** in **data** to **effects** in **spell**. I then return the **spell** variable.

The last method, **CanLearn**, is also a static method that takes an **Entity** that is trying to learn a spell and a **Spell** that represents the spell to be learned. There is a local variable **canLearn** that is set to true initially. There are then a number of checks to see if all of the prerequisites for the talent have been learned. If one of the checks fails **canLearn** will be set to false. The first check makes sure that the level of the entity passed in is not less than the required level. I then get the name of the class for the entity and convert it to a lower case string. I then use the **Contains** method of **List<T>** to see if the class is not in the list of allowed classes. There is next a foreach loop that loops through all of the keys in **AttributeRequirements**. I then use the **GetAttributeByString** method passing in the entity and the key to get the value of the attribute of the entity and compare it to the value in the dictionary for the key. If it is less than the minimum level I set can learn to false and break out of the loop. The last step is to loop through all of the prerequisite spells for the spell. If a spell in the preequisites is not in the player's spell book I set **canLearn** to false and break out of the loop.

I'm now going work on classes related to effects. I created classes called **BaseEffect**, **BaseEffectData**, and **BaseEffectDataManager**. I had added an enumeration called **EffectType** but instead of using an enumeration for the different types of effects I'm going to create classes that inherit from **BaseEffect** that represent the different types of effects so I'm going to make **BaseEffect** an abstract class. First, change the code for the **BaseEffectDataManager** to the following.

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

namespace RpgLibrary.EffectClasses
{
    public class BaseEffectDataManager
    {
        #region Field Region

        readonly Dictionary<string, BaseEffectData> effectData;

        #endregion

        #region Property Region

        public Dictionary<string, BaseEffectData> EffectData
        {
            get { return effectData; }
        }

        #endregion

        #region Constructor Region

        public BaseEffectDataManager()
        {
            effectData = new Dictionary<string, BaseEffectData>();
        }

        #endregion

        #region Method Region
        #endregion

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

Fairly straight forward like the other manager classes. There is read only field, **effectData**, that is a **Dictionary<string, BaseEffectData>** that will hold all of the **BaseEffectData** objects. There is a public property, **EffectData**, that exposes the **effectData** field and is a get only property. The constructor just creates a new **Dictionary<string, BaseEffectData>**.

Next is the **BaseEffectData** class. This class is going to be really simple, just a single string field **Name** and a protected constructor. Change **BaseEffectData** to the following.

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

namespace RpgLibrary.EffectClasses
{
    public class BaseEffectData
    {
        #region Field Region

        public string Name;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        protected BaseEffectData()
        {
        }

        #endregion

        #region Method Region
        #endregion

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

The **BaseEffect** class will be similar. It will be an abstract class though and have an abstract method as well. Change the **BaseEffect** class to the following.

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

using RpgLibrary.CharacterClasses;

namespace RpgLibrary.EffectClasses
{
    public abstract class BaseEffect
    {
        #region Field Region

        protected string name;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
            protected set { name = value; }
        }

        #endregion

        #region Constructor Region

        protected BaseEffect()
        {
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method Region

        public abstract void Apply(Entity entity);

        #endregion
    }
}
```

The first thing is I included a using statement to bring the **Enity** class in **CharacterClasses** into scope for the abstract method. There is one protected field, **name**, that is the name of the effect. There is a public property to expose the value with a protected set. I included a protected constructor for the class. I added an abstract method, **Apply**, that takes an **Enity** as a parameter. This method will be implemented in classes that inherit from **BaseEffect** so their effects can be applied to an entity.

I'm going to add in two effects that inherit from **BaseEffect**: **HealEffect** and **DamageEffect**. I will be adding data classes as well. Right click **EffectClasses** in the **XRpgLibrary** project, select **Add** and then **Class**. Name this new class **HealEffect**. Repeat that process three more times and call the classes **DamageEffect**, **HealEffectData** and **DamageEffectData**.

Change the code for the **DamageEffectData** and **HealEffectData** classes to the following.

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

```

```
namespace RpgLibrary.EffectClasses
{
    public enum DamageType { Crushing, Slashing, Piercing, Poison, Disease, Fire, Ice, Lightning,
Earth }
    public enum AttackType { Health, Mana, Stamina }

    public class DamageEffectData : BaseEffectData
    {
        #region Field Region

        public DamageType DamageType;
        public AttackType AttackType;
        public DieType DieType;
        public int NumberOfDice;
        public int Modifier;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

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

namespace RpgLibrary.EffectClasses
{
    public enum HealType { Health, Mana, Stamina }

    public class HealEffectData : BaseEffectData
    {
        #region Field Region

        public HealType HealType;
        public DieType DieType;
        public int NumberOfDice;
        public int Modifier;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

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

I changed the base namespace of the classes from **XRpgLibrary** to **RpgLibrary**. To **HealEffectData** I added in an enumeration for the types of healing. Health, Mana, and Stamina can all be healed, though not necessarily with spells. You could have potions, and we will, that can restore lost mana, stamina and health. I inherit **HealEffectData** from **BaseEffectData**. I added four fields to **HealEffectData**: **HealType**, **DieType**, **NumberOfDice**, and **Modifier**. The first, **HealType**, tells what the effect is healing. The others are used to determine how much is healed. You take a certain number of dice, roll them, and add in a modifier. I will work it so that if **NumberOfDice** is 0 then just **Modifier** will heal, for effects that heal a specific amount.

To **DamageEffectData** I added in two enumerations: **AttackType** and **DamageType**. **DamageType** is the type of damage being done and **AttackType** is what is being attacked. I will eventually be adding in resistance to the game to specific types of damage so I included **DamageType**. I will also be adding in more effects to characters like stunned, asleep, petrified, etc. Those will be specific types of effects and the effects will be curable. There will also be the ability to resist those types of effects. I will get to that in a later tutorial. I inherit **DamageEffectData** from **BaseEffectData** as well. There are five fields in this class: **DamageType**, **AttackType**, **DieType**, **NumberOfDice**, and **Modifier**. The last three work the same as healing effects. **DamageType** is the type of damage being inflicted and **AttackType** is the type of damage being inflicted.

I will deal with **HealEffect** next. This is the code for the **HealEffect** class.

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

using RpgLibrary.CharacterClasses;

namespace RpgLibrary.EffectClasses
{
    public class HealEffect : BaseEffect
    {
        #region Field Region

        HealType healType;
        DieType dieType;
        int numberOfDice;
        int modifier;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        private HealEffect()
        {
        }

        #endregion

        #region Method Region

        public static HealEffect FromHealEffectData(HealEffectData data)
        {
            HealEffect effect = new HealEffect();

            effect.healType = data.HealType;
            effect.dieType = data.DieType;
            effect.numberOfDice = data.NumberOfDice;
            effect.modifier = data.Modifier;

            return effect;
        }

        #endregion

        #region Virtual Method Region

        public override void Apply(Entity entity)
        {

            int amount = modifier;

            for (int i = 0; i < numberOfDice; i++)
                amount += Mechanics.RollDie(dieType);

            if (amount < 1)
                amount = 1;

            switch (healType)
            {
                case HealType.Health:
                    entity.Health.Heal((ushort)amount);
                    break;
                case HealType.Mana:
                    entity.Mana.Heal((ushort)amount);
                    break;
                case HealType.Stamina:
                    entity.Stamina.Heal((ushort)amount);
                    break;
            }
        }

        #endregion
    }
}
```

First, there is a using statement to bring **CharacterClasses** into scope, for **Entity**. There are fields that match the fields in the **HealEffectData** class but they start with a lower case letter rather than an upper case letter. This is how I generally write my code. Private fields generally start with a lower case letter while public start with upper case letters. I included a private constructor because again I'll be using a static method to create instance of **HealEffect**.

The **FromHealEffectData** method takes a **HealEffectData** object and returns a **HealEffect** object. It consists of creating an object, assigning values, and returning the object. The other method is the override of the **Apply** method. First there is a local variable, **amount**, that holds the amount that will be healed set to the **modifier** field. Then there is a for loop that loops **numberOfDice** times. Each pass through the loop adds the result of rolling **dieType**. An effect will always heal at least 1 point so if the amount is less than 1 I set the amount to 1. If you specify **numberOfDice** to be zero then **modifer** will be healed allowing a set value to be healed. Finally there is a switch statement on **healType**. If **healType** is **Health** then I call the **Heal** method of **Health** for the entity passed in. I do similar for the mana and stamina cases. If you recall, from way back when, the **Heal** method will add the value passed in to the current value and if the current value is larger than the maximum it will set the current value to the maximum.

DamageEffect is similar to **HealEffect** but works in reverse. It damages an entity rather than healing it. The code for **DamageEffect** follows next.

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

namespace RpgLibrary.EffectClasses
{
    public class DamageEffect : BaseEffect
    {
        #region Field Region

        DamageType damageType;
        AttackType attackType;
        DieType dieType;
        int numberOfDice;
        int modifier;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        private DamageEffect()
        {
        }

        #endregion

        #region Method Region

        public static DamageEffect FromDamageEffectData(DamageEffectData data)
        {
            DamageEffect effect = new DamageEffect();

            effect.damageType = data.DamageType;
            effect.attackType = data.AttackType;
            effect.dieType = data.DieType;
            effect.numberOfDice = data.NumberOfDice;
            effect.modifier = data.Modifier;

            return effect;
        }

        #endregion

        #region Virtual Method Region

        public override void Apply(Entity entity)
        {
            int amount = modifier;

            for (int i = 0; i < numberOfDice; i++)
                amount += Mechanics.RollDie(dieType);

            if (amount < 1)
                amount = 1;

            switch (attackType)
            {
                case AttackType.Health:
                    entity.Health.Damage((ushort)amount);
                    break;
                case AttackType.Mana:
                    entity.Mana.Damage((ushort)amount);
                    break;
                case AttackType.Stamina:
                    entity.Stamina.Damage((ushort)amount);
                    break;
            }
        }
        #endregion
    }
}
```

There is a using statement to bring **CharacterClasses** into scope, for **Entity**. There are fields that match the fields in the **DamageEffectData** class but they start with a lower case letter rather than an upper case letter. I included a private constructor because again I'll be using a static method to create instance of **DamageEffect**.

The **FromDamageEffectData** method takes a **DamageEffectData** object and returns a **DamageEffect** object. It consists of creating an object, assigning values, and returning the object. The other method is the override of the **Apply** method. First there is a local variable, amount, that holds the amount of damage that will be inflicted set to the modifier field. Then there is a for loop that loops **numberOfDice** times. Each pass through the loop adds the result of rolling **dieType**. An effect will always do at least at least 1 point so if the amount is less than 1 I set the amount to 1. If you specify **numberOfDice** to be zero then **modifer** will be inflicted allowing a specific amount of damage to be inflicted. Finally there is a switch statement on **attackType**. If **healType** is **Health** then I call the **Damage** method of **Health** for the entity passed in. I do similar for the mana and stamina cases. If you recall, from way back when, the **Damage** method will subtract the value passed in from the current value and if the value is negative the current value will be set to 0.

I think I'm going to end this tutorial here. I wanted to flesh out classes related to spells and start on the classes related to effects. Things are starting to take form but there is a long way to go yet. I encourage you to visit the news page of my site, XNA Game Programming Adventures, for the latest news on my tutorials.

Good luck in your game programming adventures!

Jamie McMahon