A Summoner's Tale
Chapter 11
It's Clobbering Time, Almost

Welcome to my Summoner's Tale tutorial series on creating a Pokémon-inspired game with MonoGame. I'm writing these tutorials for the MonoGame 3.8.1 framework using Visual Studio 2022. The code should work on previous versions of MonoGame and Visual Studio. I plan on creating the editors on macOS and Windows. I'm unfamiliar with Linux, so a few projects may not be done for that platform.

The tutorials will make more sense if they are read in order. You can find the list of tutorials on my web blog, A Summoner's Tale page. In addition to the PDFs, I will make the code for each tutorial available on GitHub here: https://github.com/Synammon/summoners-tale. It will also be included on the page that links to the tutorials.

As The Thing from Fantastic Four would say, It's Clobbering Time. We have a character on the map, and they can talk to people. However, the whole point of the game is to battle other people who have shadow monsters. Wait a minute, we don't even have shadow monsters in the game. Whatever are we going to do? Well, the first step is to add a class that represents the moves shadow monsters can do. First, right-click the Psilibrary project in the solution explorer, select Add and then New Folder. Name this new folder ShadowMonsters. Now, right-click the ShadowMonsters folder in the solution explorer, select Add and then Class. Name this new class MoveData. Before I give you the code for that class, right-click the ShadowMonsters folder, select Add and then Class. Name this new class ShadowMonsterData. Here is the code for the MoveData class.

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

namespace Psilibrary.ShadowMonsters
{
    public enum TargetType { Self, Enemy }
    public enum TargetAttribute
    {
        Health,
        Attack,
        Defence,
        SpecialAttack,
        SpecialDefense,
        Speed,
        Accuracy
    }
    public class MoveData
    {
        public string Name { get; set; }
        public int Elements { get; set; }
        public TargetType Target { get; set; }
        public TargetAttribute TargetAttribute { get; set; }
        public Point Mana { get; set; }
        public Point Range { get; set; }
        public int Status { get; set; }
        public bool Hurts { get; set; }
        public bool IsTemporary { get; set; }
    }
}
```

There are two enumerations in this file. The first enumeration is TargetType, which is the target of a move, either the enemy or the self. The next is TargetAttribute which is what attributes the move targets. There is an element for all of the attributes. A move has one or more elements associated with it. It is an integer instead of an enumeration so that a move could have multiple elements. So, a move Steam Blast could be water and fire. The Elements enumeration is in the ShadowMonsterData class. These two data classes are collections of properties. There are properties for who the move targets and what the move targets. There is also a property for the amount of mana for the move and how much mana is remaining. This is like the PP in pokemon. It requires one mana to use a move. When that mana is spent, the move cannot be used. There is also a range for the move, the minimum and maximum values. Next is an integer that is a combination of statuses the move can cause. Typically, there is only one, but two or more should be a possibility. There is also a bool, Hurts, that determines if a move hurts or heals. Finally, there is a property if the move is permanent or temporary. For example, a physical attack would be permanent, but a buff that raises speed would only be temporary.

Now that you have read that code, here is the code for the ShadowMonsterData class.

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

namespace Psilibrary.ShadowMonsters
{
    public enum Element
    {
        Normal = 0,
        Fire = 1,
        Water = 2,
        Earth = 4,
        Wind = 8,
        Light = 16,
        Dark = 32
    }

    public class ShadowMonsterData
    {
        public const int Normal = 0;
        public const int Asleep = 1;
        public const int Confused = 2;
        public const int Poisoned = 4;
        public const int Paralyzed = 8;
        public const int Burn = 16;
        public const int Frozen = 32;

        public string Name { get; set; }
        public int Elements { get; set; }
        public int Level { get; set; }
        public int Experience { get; set; }
        public Point Health { get; set; }
        public int Attack { get; set; }
        public int Defence { get; set; }
        public int SpecialAttack { get; set; }
        public int SpecialDefence { get; set; }
        public int Speed { get; set; }
        public int Accuracy { get; set; }
        public int AttackMod { get; set; }
        public int DefenceMod { get; set; }
        public int SpecialAttackMod { get; set; }
        public int SpecialDefenceMod { get; set; }
        public int SpeedMod { get; set; }
        public int AccuracyMod { get; set; }
        public int Status { get; set; }
        public List<MoveData> Moves { get; set; }
    }
}
```

The element enumeration is in this file. Currently, there are seven elements. One for shadow monsters with no elements. Next are the four base elements: Fire, Water, Earth and Wind. There are also Light and Dark elements. You will see that they are numbered. That is because we will be combining elements. We can use logical or, |, to combine them, and logical and, &, to test for them. So, let's say you wanted a steam shadow monster. You could combine them as a 3. Then, using a logical and of 2, find out that they are a water shadow monster and local and of 1 to find out they are also a fire shadow monster.

There are some constants in the file that act much like the enumeration does. They are in powers of two so that they can be combined and separated using Boolean logic. In fact, there is little difference between the two.

Aside from the constants, there are a number of properties that a shadow monster has. They have a name, elements, level, experience, health, attack, defence, special attack, special defence, speed, accuracy, status and a list of possible moves. There are also modifiers for each of the attributes. These should be set to zero at the start and end of combat as they are temporary.

Now it is time to implement some of the logic. Let's start with a move. First, we need a folder to hold our classes. Right-click the SummonersTale class, select Add and then New Folder. Name this new folder ShadowMonsters. Now, right-click the ShadowMonsters folder, select Add and then Class. Name this new class Move. The code for that class follow next.

```
using Microsoft.Xna.Framework;
using Psilibrary.ShadowMonsters;
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Xml.Serialization;

namespace SummonersTale.ShadowMonsters
{
    public class Move : MoveData
    {
        protected static Random Random { get; set; } = new Random();

        public Move()
        {
        }

        public virtual void Apply(ShadowMonster target)
        {
            int amount = Random.Next(Range.X, Range.Y);

            if (Hurts)
            {
                amount *= -1;
            }

            string propertyName = !IsTemporary ? TargetAttribute.ToString() :
string.Format("{0}Mod",TargetAttribute.ToString());
            var property = target.GetType().GetProperty(propertyName);

            if (property.PropertyType != typeof(Point))
            {
                int value = (int)property.GetValue(this);
                property.SetValue(this, value + amount, null);
            }
            else
            {
                Point p = ((Point)(property.GetValue(this)));
                p.X += amount;

                if (p.X > p.Y)
                {
                    p.X = p.Y;
                }

                property.SetValue(this, p, null);
            }
        }

        public override string ToString()
        {
            StringBuilder sb = new();
            Type moveType = typeof(Move);
            bool first = true;

            foreach (var property in moveType.GetProperties())
            {
                if (property.Name == "Random")
                {
                    continue;
                }

                if (first)
                {
                    first = false;
                }
                else
                {
                    sb.Append('\t');
                }
                if (property.PropertyType != typeof(Point))
                {
                    object o = moveType.GetProperty(property.Name).GetValue(this);
                    sb.Append($"{property.Name}+{o.ToString()} ");
                }
                else if (property.PropertyType == typeof(Point))
                {
                    Point p =
(Point)moveType.GetProperty(property.Name).GetValue(this);
                    sb.Append($"{property.Name}+{p.X}:{p.Y}");
                }
            }
            return sb.ToString();
        }

        public static Move FromString(string value)
        {
            Move move = new();
            Type moveType = typeof(Move);

            value = value.Replace("[", "");
            value = value.Replace("]", "");

            string[] parts = value.Split('\t');

            foreach (string part in parts)
            {
                string[] attributes = part.Split('+');

                if (moveType.GetProperty(attributes[0]).PropertyType !=
typeof(Point))
                {
                    if (attributes[0] != "Target" && attributes[0] !=
"TargetAttribute")
                    {
                        if (moveType.GetProperty(attributes[0]).PropertyType ==
typeof(string))
                        {
                            moveType.GetProperty(attributes[0]).SetValue(move,
attributes[1], null);
                        }
                        else if (moveType.GetProperty(attributes[0]).PropertyType ==
typeof(Int32))
                        {
                            moveType.GetProperty(attributes[0]).SetValue(move,
int.Parse(attributes[1]), null);
                        }
                    }

                    if (attributes[0] == "Target")
                    {
                        if (Enum.TryParse<TargetType>(attributes[1], out TargetType
target))
                        {
                            moveType.GetProperty(attributes[0]).SetValue(move,
target, null);
                        }
                    }

                    if (attributes[0] == "TargetAttribute")
                    {
                        if (Enum.TryParse<TargetAttribute>(attributes[1], out
TargetAttribute target))
                        {
                            moveType.GetProperty(attributes[0]).SetValue(move,
target, null);
                        }
                    }
                }

                if (moveType.GetProperty(attributes[0]).PropertyType ==
typeof(Point))
                {
                    string[] components = attributes[1].Split(':');
                    Point point;

                    _ = int.TryParse(components[0], out point.X);
                    _ = int.TryParse(components[1], out point.Y);

                    moveType.GetProperty(attributes[0]).SetValue(move, point, null);
                }
            }

            return move;
        }
    }
}
```

The class inherits the MoveData class, so it has access to all of its members. Come to think of it. I haven't explained why I've split the classes in two. Well, it is because we need to be able to save and load the items. I could have used flat files or binary files, but I decided to go with the intermediate serializer. I really love that class, as I am sure you can see. It just makes managing content so much cleaner. In order to use it, we need a reference to the class, and we don't get one with shared projects. So, I've split the class into a data part and a code part.

There is a static property in this class that is a Random object. We will use it to calculate the amount that a move does from the range.

There is a virtual method, Apply, that applies the move to the target so that it can be overridden in any child classes. First, it calculates the amount that the move will do using the overload of the method of the Random class that takes a minimum and maximum value. If it Hurts, we want negative that amount. Rather than use a switch to determine which attribute to apply the amount to, I used reflection. What is reflection? Basically, it is the ability to get information about the class at runtime, like what properties and methods it has. To determine which attribute to apply it to, I either use the target attribute cast to a string or the target attribute cast to a string plus Mod. I then use reflection to get the property using the property name.

If the property type is not a point, I get the current value of the property using the property and this. I then set the value to the current value plus the amount using the SetValue method, passing in this for the current object, the value to be set and null. If it is a point, we have to do a little more work. First, we get the value of the attribute using GetValue on the property. The X value, or current amount, is updated by the amount the move does. If it is greater than the maximum, it is set to the maximum. It then uses SetValue to set the value of the property.

Next, I override the ToString method. It might have been better just to append the properties manually. Instead, I used reflection to get all of the properties at run time and append them that way. If you want to go that route in your game, it is a good approach. So, instead of using strings, I used a string builder. In general, it is more efficient than concatenating strings. I get the type of the Move class as it is required for use in reflection. I then set a Boolean variable to true to determine if it is the first time through looping over the properties of the class. In a foreach loop, I loop through all of the properties of the class. If the property is Random, we don't want to write that, so we go to the next property. If it is the first time through, we toggle first to false. Otherwise, we append a tab character. So, why that? Well, shadow monsters are separated by commas, so I couldn't use that. The next most often used character is a tab. Also, it isn't commonly used.

The simplest case in building the string is if the property is not a Point. The PropertyType property of the property exposes the type of the property. I compare that to the type of the Point class. If it is not a point, I get the value of the property. I use the type of the move class and call the GetProperty method to get the property we are interested in using its Name and then call GetValue, passing in this for the instance. I then append the name and the value to the string builder. I separated the values by a plus sign. I do something similar for a Point. I capture the value of the point in a local variable. I then append a three-part value, the name and the components separated by a plus sign and the two components of the point separated by a colon.

As you've probably guessed, FromString takes a string and returns a Move. The first step is to create a Move and then to create a Type for reflection. The one benefit of this approach is that the string doesn't have to be in a specific order or complete, though in the latter case, you will get unexpected results.

After creating the Move and Type, it is time to work on the string. The first step is to remove the square braces. Now, split the string on the tab character to get the individual parts. After we have the parts, we iterate over them.

We then split each part on the plus sign. The check to see if the type is not a point. Inside of that if statement is a check to see that the property is not Target and not TargetAttribute because enumerations are handled differently. The simplest case is if the property is a string. In this case, you call the SetValue method passing in the value of the attribute. Integers are similarly simple. In this case, you just parse them. For enumerations, I use the TryParse method of the Enum class to parse them. So, just a little more complex. Points are the most complex case. You need to split the second half of the attribute pair into parts on the colon. Next, you create a point to hold the value you want to set. Next, you parse the X and Y components of the point. Finally, you set the value using the new point. Now, you return the move.

Not terribly complex, but still a little over-baked for our purposes. It is more than we need for moves, but it is easily extensible. You just need to add a new property, and the code will take care of the rest. Unless you add something other than a string, integer, point or one of the already defined enumerations. In this case, you will need to update the code for parsing.

That brings us to shadow monsters. They are like moves, just a little more complex. Right-click the ShadowMonsters folder in SummonersTale project, select Add and then Class. Name this new class ShadowMonster. Replace that class with the following code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using Psilibrary.ShadowMonsters;
using SummonersTale.SpriteClasses;
using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;
using System.Linq;
using System.Reflection;
using System.Text;

namespace SummonersTale.ShadowMonsters
{
    public class ShadowMonster : ShadowMonsterData
    {
        public Texture2D Sprite;
        public List<MoveData> UnlockedMoves { get; protected set; } = new();
        public List<MoveData> LockedMoves { get; protected set; } = new();

        private string[] assignedMoves = new string[4];

        public ShadowMonster()
        {
        }

        public void LoadContent(ContentManager Content)
        {
            Sprite =
Content.Load<Texture2D>(string.Format("ShadowMonsterSprites/{0}", Name));
        }

        public override string ToString()
        {
            StringBuilder sb = new();

            Type type = typeof(ShadowMonster);

            bool first = true;

            foreach (var property in type.GetProperties())
            {
                if (!first)
                {
                    sb.Append(',');
                }
                else
                {
                    first = false;
                }

                if (property.Name != "Health" && !property.Name.Contains("Moves"))
                {
                    sb.Append(
                        string.Format("{0}={1}",
                        property.Name,
                        type.GetProperty(property.Name).GetValue(this).ToString()));
                }
                else if (property.Name == "Health")
                {
                    sb.Append($"Health={Health.X}:{Health.Y}");
                }
                else if (property.Name.Contains("Moves"))
                {
                    // Serialize moves to string here
                    sb.Append($"{property.Name}=");

                    if (type.GetProperty(property.Name).GetValue(this) is
List<MoveData> moveData)
                    {
                        if (moveData != null)
                        {
                            bool firstMove = true;

                            foreach (var move in moveData)
                            {
                                if (!firstMove)
                                {
                                    sb.Append('&');
                                }
                                else
                                {
                                    firstMove = false;
                                }

                                sb.Append($"[{move.ToString()}]");
                            }
                        }
                    }
                }
            }

            return sb.ToString();
        }

        public static ShadowMonster FromString(string value)
        {
            ShadowMonster monster = new();
            Type type = typeof(ShadowMonster);

            string[] parts = value.Split(',');

            foreach (string part in parts)
            {
                string[] attributes = part.Split('=');

                if (attributes.Length < 2 ||
string.IsNullOrWhiteSpace(attributes[1]))
                {
                    continue;
                }

                PropertyInfo property = type.GetProperty(attributes[0]);

                if (property != null && !attributes[0].Contains("Moves") &&
attributes[0] != "Health")

                {
                    if (!int.TryParse(attributes[1], out int converted))
                    {
                        property.SetValue(monster, attributes[1], null);
                    }
                    else
                    {
                        property.SetValue(monster, converted, null);
                    }
                }
                else if (property != null && (attributes[0] == "Health"))
                {
                    string[] health = attributes[1].Split(":");

                    if (health.Length == 2)
                    {
                        if (int.TryParse(health[0], out int currentHealth) &&
int.TryParse(health[1], out int maxHealth))
                        {
                            monster.Health = new(currentHealth, maxHealth);
                        }
                    }
                }
                else if (property != null && (attributes[0].Contains("Moves")))
                {
                    property.SetValue(monster, new List<MoveData>(), null);
                    string[] moveString = attributes[1].Split('&');
```

```
                    foreach (string move in moveString)
                    {
((List<MoveData>)property.GetValue(monster)).Add(Move.FromString(move));
                    }
                }
            }

            return monster;
        }
    }
}
```

There are two additional properties and two fields in this class. For properties, there is a list of unlocked and locked moves. For fields, there is a Texture2D for the shadow monster and an array of moves that are the moves the shadow monster can currently use. The constructor takes no parameters, and the LoadContent method takes a ContentManager that is used to load the texture.

The ToString method serializes a shadow monster to a string. It first creates a string builder to hold the data. Next, it gets the type of the ShadowMonster class for use in reflection. Like in a move, there is a first variable to tell if we are appending a comma to separate the properties. Now, we loop over the properties.

Inside that loop, I check to see that we are not on the first item. If not, I append a comma. If we are on the first item, I set the first variable to false. Other than health and moved, serializing this class is pretty straightforward. So, I check to see if the current property is not Health or does not contain Moves. If that is the case, I append the property using the string.Format method to format the property separating the name and the value by an equals sign. If the current property is Health, I append it using string interpolation writing Health, an equals sign, and the X and Y components of the point separated by a colon.

Because they are a complex type, moves require a little bit more work. The first step is to append the name of the property. Next, I use pattern matching to check to see if the property is a List<MoveData>. If it is, I check to make sure it is not null. If it is not null, I loop over all of the moves in the list. If it is not the first move, I append an ampersand to separate them. Otherwise, I set the first move to false. I then append the move as a string. After serializing the moves, I return the string builder as a string.

FromString is static and takes the value to be parsed. It returns a ShadowMonster object. The first step is to create a ShadowMonster object to set the properties on. Next, you need to grab the type that we can use reflection on. Now, we split the string on commas. Following that, you loop over all of the parts.

Inside that loop is where we parse the parts. We split each part on the equals sign. If the length of that is less than two or the second part is null or white space, we move to the next part.

Next, we get a PropertyInfo object using the name of the part. If it is not null, it does not contain Moves, and it is not Health, I parse it as an integer or a string. If it is Health, I parse it as a Point the same way I did Mana in the Moves class.

If the part is not null and contains Moves, I set the value to a new List<MoveData>. I then split the moves into separate moves on the ampersand. I then loop over each of those parts. The part is passed to the FromString of the Move class. The Monster object is then returned.

The last thing I'm going to do is create a shadow monster, serialize it to a string, and deserialize it back. I will do that in the LoadContent method of the Desktop class. Replace that method with the following code.

```
protected override void LoadContent()
{
    _spriteBatch = new SpriteBatch(GraphicsDevice);
    Services.AddService(typeof(SpriteBatch), _spriteBatch);

    _conversationManager = new(this);
    _conversationManager.LoadConverstions(this);
    _conversationManager.WriteConversations();

    Components.Add(_conversationManager);

    _conversationState = new(this);
    _playState = new(this);
    _titleState = new(this);
    _mainMenuState = new(this);
    _newGameState = new(this);

    _manager.PushState(_titleState);

    Move smash = new()
    {
        Name = "Smash",
        Range = new(1, 6),
        Mana = new(40, 40),
        Target = TargetType.Enemy,
        TargetAttribute = TargetAttribute.Health,
        IsTemporary = false,
        Elements = 0
    };

    string m = smash.ToString();

    smash = Move.FromString(m);

    List<MoveData> moves = new()
    {
        smash
    };

    Move bash = new()
    {
        Name = "Bash",
        Range = new(2, 8),
        Mana = new(30, 30),
        Target = TargetType.Enemy,
        TargetAttribute = TargetAttribute.Health,
        IsTemporary = false,
        Elements = 0
    };

    ShadowMonster monster = new()
    {
        Name = "Goblin",
        Moves = moves,
        Elements = 0,
        Health = new(25, 25),
    };

    monster.LockedMoves.Add(bash);
    monster.UnlockedMoves.Add(smash);

    m = monster.ToString();

    monster = ShadowMonster.FromString(m);
}
```

After creating the game states, I create a new move, Smash, that is appropriate for a goblin. I set some of the attributes of the move. I then serialize it to a string and deserialize it back again. I then add it to a List<MoveData>. I create a second move, Bash, that also sounds fitting for a goblin. Next, I create a test monster. I add Smash to the unlocked moves and Bash to the locked moves. I serialize the monster to a string and then reverse the process.

You can build and run now, but nothing has changed visibly. I will tackle that in the next tutorial, as I'm not going to dive any further in this tutorial. I think I've fed you more than enough for one day. I encourage you to keep visiting my blog for the latest news on my tutorials.

Good luck with your game programming adventures.
Cynthia