XNA 4.0 RPG Tutorials

Part 30

Updating Weapons

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.

I wasn't going to modify weapons to use effects but in the end it makes sense to update them. It is going to break a few things so this is going to be a rather tedious tutorial. In the end it will be well worth the effort though.

The first step is to add in an override of the **ToString** method of **DamageEffectData**. Add this override of the **ToString** method to the **Virtual Method Region** of **DamageEffectData**.

```
public override string ToString()
{
    string toString = Name + ", " + DamageType.ToString() + ", ";
    toString += AttackType.ToString() + ", ";
    toString += DieType.ToString() + ", ";
    toString += NumberOfDice.ToString() + ", ";
    toString += Modifier.ToString();

    return toString;
}
```

Nothing hard there. I just create a local variable and build a string that represents the instance of **DamageEffectData** and then return it. The next step is to update the **WeaponData** class. You want to replace the old field related to damage with a **DamageEffectData** field. Then update the **ToString** method to use **DamageEffectData** now. Update the **WeaponData** class to the following.

Weapons will have a **DamageEffectData** associated with them because they cause damage. You will need to update the **WeaponData** and **Weapon** classes. I will start with the **WeaponData** class. Change that class to the following.

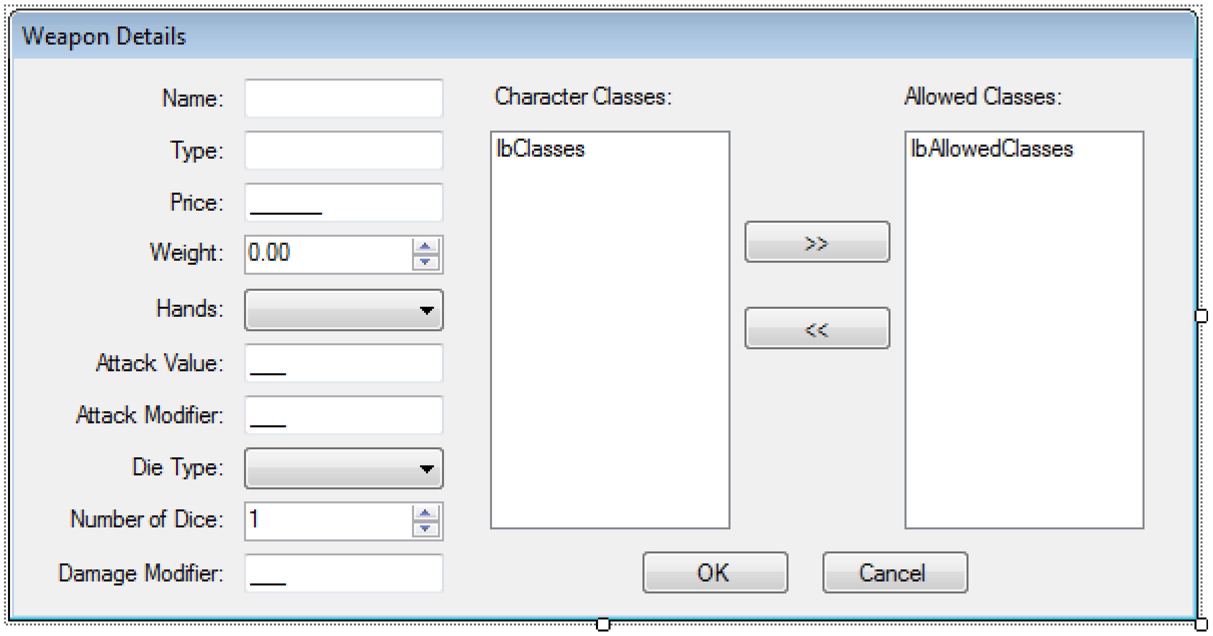
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RpgLibrary.EffectClasses;

namespace RpgLibrary.ItemClasses
{
    public class WeaponData
    {
        public string Name;
        public string Type;
        public int Price;
        public float Weight;
        public bool Equipped;
        public Hands NumberHands;
        public int AttackValue;
        public int AttackModifier;
        public DamageEffectData DamageEffectData;
        public string[] AllowableClasses;

        public WeaponData()
        {
            DamageEffectData = new DamageEffectData();
        }

        public override string ToString()
        {
            string toString = Name + ", ";
            toString += Type + ", ";
            toString += Price.ToString() + ", ";
            toString += Weight.ToString() + ", ";
            toString += NumberHands.ToString() + ", ";
            toString += AttackValue.ToString() + ", ";
            toString += AttackModifier.ToString() + ", ";
            toString += DamageEffectData.ToString();

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

            return toString;
        }
    }
}
```

I replaced the **DamageValue** and **DamageModifier** fields with a **DamageEffectData** field that represents the damage that the weapon does. I also update the **ToString** method to write out the **DamageEffectData** field. I also added in a using statement for **EffectClasses**. The **Weapon** class took a little more work. Update the **Weapon** class to the following.

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

namespace RpgLibrary.ItemClasses
{
    public class Weapon : BaseItem
    {
        #region Field Region

        Hands hands;
        int attackValue;
        int attackModifier;
        DamageEffectData damageEffectData;

        #endregion

        #region Property Region

        public Hands NumberHands
        {
            get { return hands; }
            protected set { hands = value; }
        }

        public int AttackValue
        {
            get { return attackValue; }
            protected set { attackValue = value; }
        }

        public int AttackModifier
        {
            get { return attackModifier; }
            protected set { attackModifier = value; }
        }

        public DamageEffectData DamageEffect
        {
            get { return damageEffectData; }
            protected set { damageEffectData = value; }
        }

        #endregion

        #region Constructor Region

        public Weapon(
                string weaponName,
                string weaponType,
                int price,
                float weight,
                Hands hands,
                int attackValue,
                int attackModifier,
                DamageEffectData damageEffectData,
                params string[] allowableClasses)
            : base(weaponName, weaponType, price, weight, allowableClasses)
        {
            NumberHands = hands;
            AttackValue = attackValue;
            AttackModifier = attackModifier;
            DamageEffect = damageEffectData;
        }

        #endregion

        #region Abstract Method Region

        public override object Clone()
        {
            string[] allowedClasses = new string[allowableClasses.Count];

            for (int i = 0; i < allowableClasses.Count; i++)
                allowedClasses[i] = allowableClasses[i];

            Weapon weapon = new Weapon(
                Name,
                Type,
                Price,
                Weight,
                NumberHands,
                AttackValue,
                AttackModifier,
                DamageEffect,
                allowedClasses);

            return weapon;
        }

        public override string ToString()
        {
            string weaponString = base.ToString() + ", ";
            weaponString += NumberHands.ToString() + ", ";
            weaponString += AttackValue.ToString() + ", ";
            weaponString += AttackModifier.ToString() + ", ";
            weaponString += DamageEffect.ToString();

            foreach (string s in allowableClasses)
                weaponString += ", " + s;

            return weaponString();
        }

        #endregion
    }
}
```

Again there is a using statement for the **EffectClasses** namespace. I replaced the **damageValue** and **damageModifier** fields with a **DamageEffectData** field. I updated the properties for the new field as well. The constructor now takes a **DamageEffectData** field for damage instead of the two integer fields. It sets the fields with the values passed in. I updated the **Clone** and **ToString** methods as well to use the new field.

That ends up breaking a lot of things. It breaks the item editor and the weapons that were created. The best solution that I could come up with is to delete the weapons that were added and update the editor. Browse to the **Weapon** folder in the **EyesOfTheDragonContent** project. Select all of the entries, right click on them and select **Delete**. Next, right click the **RpgEditor** project and select **Set as StartUp Project**. You will have four errors, all of them in **FormWeaponDetails**. The first step is to redesign the form. I'm going to make our life a little easier though. A weapon only causes **Weapon** damage. The will also only attack **Health**. That leaves having to change the form so you can select the **DieType**, number of dice, and modifier. The finished form in the designer appears below.



First, make your form bigger to allow for the new controls. I deleted the **Damage Value:** label and the **mtbDamageValue** masked text box. I deleted the **Damage Modifier:** label and **mtbDamageModifier** as well. Under the **Attack Modifier:** label I dragged a new label and set its **Text** property to **Die Type:**. Beside that I dragged a combo box. Under the **Die Type:** label I dragged on and positioned a label and set its **Text** property to **Number Of Dice**. I dragged a numeric up down control. Under those I dragged a label and set its **Text** property to **Damage Modifier:**. I dragged a masked text box beside that label and under the numeric up down. I then resized the form back to the desired size. I also moved the **OK** and **Cancel** buttons closer to the bottom of the form and resized the list boxes. The properties I set for the new controls are in the following tables.

Combo Box	
Property	**Value**
(Name)	cboDieType
DropDownStyle	DropDownList
Location	115, 194
Size	100, 21
TabIndex	17

Numeric Up Down	
Property	**Value**
(Name)	nudDice
Location	115, 221
Minimum	1
Size	100, 20
TabIndex	18

Masked Text Box	
Property	**Value**
(Name)	mtbDamageModifier
Location	115, 247
Mask	0
Size	100, 20
TabIndex	19

The next step will be to add the code for the form. Bring up the code for **FormWeaponDetails** and update it to the following.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using RpgLibrary.ItemClasses;
using RpgLibrary;

namespace RpgEditor
{
    public partial class FormWeaponDetails : Form
    {
        #region Field Region
        WeaponData weapon = null;

        #endregion

        #region Property Region

        public WeaponData Weapon
        {
            get { return weapon; }
            set { weapon = value; }
        }

        #endregion

        #region Constructor Region

        public FormWeaponDetails()
        {
            InitializeComponent();

            this.Load += new EventHandler(FormWeaponDetails_Load);
            this.FormClosing += new FormClosingEventHandler(FormWeaponDetails_FormClosing);

            btnMoveAllowed.Click += new EventHandler(btnMoveAllowed_Click);
            btnRemoveAllowed.Click += new EventHandler(btnRemoveAllowed_Click);
            btnOK.Click += new EventHandler(btnOK_Click);
            btnCancel.Click += new EventHandler(btnCancel_Click);
        }

        #endregion

        #region Event Handler Region

        void FormWeaponDetails_Load(object sender, EventArgs e)
        {
            foreach (string s in FormDetails.EntityDataManager.EntityData.Keys)
                lbClasses.Items.Add(s);

            foreach (Hands location in Enum.GetValues(typeof(Hands)))
                cboHands.Items.Add(location);

            foreach (DieType die in Enum.GetValues(typeof(DieType)))
                cboDieType.Items.Add(die);

            cboHands.SelectedIndex = 0;
            cboDieType.SelectedIndex = 0;
            cboDieType.SelectedValue = Enum.GetName(typeof(DieType), DieType.D4);

            if (weapon != null)
            {
                tbName.Text = weapon.Name;
                tbType.Text = weapon.Type;
                mtbPrice.Text = weapon.Price.ToString();
                nudWeight.Value = (decimal)weapon.Weight;
                cboHands.SelectedIndex = (int)weapon.NumberHands;
                mtbAttackValue.Text = weapon.AttackValue.ToString();
                mtbAttackModifier.Text = weapon.AttackModifier.ToString();

                for (int i = 0; i < cboDieType.Items.Count; i++)
                {
                    if (cboDieType.Items[i].ToString() ==
weapon.DamageEffectData.DieType.ToString())
                    {
                        cboDieType.SelectedIndex = i;
                        cboDieType.SelectedValue = cboDieType.Items[i];
                        break;
                    }
                }

                nudDice.Value = weapon.DamageEffectData.NumberOfDice;
                mtbDamageModifier.Text = weapon.DamageEffectData.Modifier.ToString();

                foreach (string s in weapon.AllowableClasses)
                {
                    if (lbClasses.Items.Contains(s))
                        lbClasses.Items.Remove(s);

                    lbAllowedClasses.Items.Add(s);
                }
            }
        }

        void FormWeaponDetails_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (e.CloseReason == CloseReason.UserClosing)
            {
                e.Cancel = true;
            }
        }

        void btnMoveAllowed_Click(object sender, EventArgs e)
        {
            if (lbClasses.SelectedItem != null)
            {
                lbAllowedClasses.Items.Add(lbClasses.SelectedItem);
                lbClasses.Items.RemoveAt(lbClasses.SelectedIndex);
            }
        }

        void btnRemoveAllowed_Click(object sender, EventArgs e)
        {
            if (lbAllowedClasses.SelectedItem != null)
            {
                lbClasses.Items.Add(lbAllowedClasses.SelectedItem);
                lbAllowedClasses.Items.RemoveAt(lbAllowedClasses.SelectedIndex);
            }
        }

        void btnOK_Click(object sender, EventArgs e)
        {
            int price = 0;
            float weight = 0f;
            int attVal = 0;
            int attMod = 0;
            int damMod = 0;

            if (string.IsNullOrEmpty(tbName.Text))
            {
                MessageBox.Show("You must enter a name for the item.");
                return;
            }

            if (!int.TryParse(mtbPrice.Text, out price))
            {
                MessageBox.Show("Price must be an integer value.");
                return;
            }

            weight = (float)nudWeight.Value;

            if (!int.TryParse(mtbAttackValue.Text, out attVal))
            {
                MessageBox.Show("Attack value must be an interger value.");
                return;
            }

            if (!int.TryParse(mtbAttackModifier.Text, out attMod))
            {
                MessageBox.Show("Attack modifier must be an interger value.");
                return;
            }

            if (!int.TryParse(mtbDamageModifier.Text, out damMod))
            {
                MessageBox.Show("Damage modifier must be an integer value.");
                return;
            }

            List<string> allowedClasses = new List<string>();

            foreach (object o in lbAllowedClasses.Items)
                allowedClasses.Add(o.ToString());

            weapon = new WeaponData();

            weapon.Name = tbName.Text;
            weapon.Type = tbType.Text;
            weapon.Price = price;
            weapon.Weight = weight;
            weapon.NumberHands = (Hands)cboHands.SelectedIndex;
            weapon.AttackValue = attVal;
            weapon.AttackModifier = attMod;
            weapon.AllowableClasses = allowedClasses.ToArray();

            weapon.DamageEffectData.Name = tbName.Text;
            weapon.DamageEffectData.AttackType = RpgLibrary.EffectClasses.AttackType.Health;
            weapon.DamageEffectData.DamageType = RpgLibrary.EffectClasses.DamageType.Weapon;

            weapon.DamageEffectData.DieType = (DieType)Enum.Parse(
                typeof(DieType),
                cboDieType.Items[cboDieType.SelectedIndex].ToString());

            weapon.DamageEffectData.NumberOfDice = (int)nudDice.Value;
            weapon.DamageEffectData.Modifier = damMod;

            this.FormClosing -= FormWeaponDetails_FormClosing;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            weapon = null;
            this.FormClosing -= FormWeaponDetails_FormClosing;
            this.Close();
        }

        #endregion
    }
}
```

Most of the code is the same but enough of it changed that I wanted to give you the code for it all. The first changes were in the **FormWeaponDetails_Load** method. The first new code is that I add all of the values in the **DieType** enumeration to **cboDieType**. What is different is that you can't just cast **DieType** to an integer for the **SelectedIndex** property of **cboDieType** because **DieType** has values associated with each member of the enumeration. For example, 4 is associated with **D4** instead of 0. For that reason I set the **SelectedValue** property of **cboDieType** to the name of **DieType.D4**. I get the name using the **GetName** method passing in the type and the value.

The next change is in the if statement where I check to see if **weapon** is not null. The first change is a for loop that loops through all of the items in **cboDieType**. I check to see if the **ToString** value of the item is equal to the **ToString** value of the **DieType** field of **DamageEffectData** of **weapon**. I set **SelectedIndex** to **i** and **SelectedValue** to **cboDieType.Items[i]**. I also break out of the loop. I then set the **Value** property of **nudDice** to the **NumberOfDice** field of **DamageEffectData**. I also set the **Text** property of **mtbDamageModifier** to the **Modifier** field.

I also had to modify **btnOK_Click** because **WeaponData** changed as well as the controls on the form. I of course removed everything that had to do with damage to use **DamageEffectData**. The verifying of values on controls is like before. The new part is creating a new **WeaponData** object. You have to set the fields of **DamageEffectData**. I set the **Name** to be the **Text** property of **tbName** as weapons will be unique so the **DamageEffectData** associated with weapons will also be unique by name. For now I'm assuming that the **AttackType** will be **AttackType.Health** and that the **DamageType** will be **DamageType.Weapon**. You could easily have controls on the form to select these. The hard part was setting the **DieType** field. I used the **Enum.Parse** method which parses a string to be the associated value of an enumeration. For the type you pass in **typeof(DieType)** and for the string I passed in the item at **SelectedIndex** and the **ToString** method. I then set the **NumberOfDice** and **Modifier** fields using the **Value** property of **nudDice** and **damMod** variable respectively.

You could have different types of weapon damages like piercing, slashing and crushing and different armors have different resistances to the types of damages. A lot of pen and paper RPGs follow this route and I've seen computer RPGs follow that route as well. I'm not going to go into that depth in the tutorials but it is certainly possible to do it. What I've done doesn't allow for enchanted weapons and armor, like a flaming sword. I may in another tutorial add in enchanted weapons and armor. I would be doing that by creating classes that inherit from **Weapon** and **Armor** rather than changing the existing classes.

If you build and run the editor now things will work. You will be able to create weapons, save them and read them back in like before. I think the last thing I'm going to cover in this tutorial is adding in updated data for weapons. I will use a table like I did before.

Weapons

Name	Type	Price	Weight	Hands	Attack Value	Attack Modifier	Die Type	# of Dice	Damage Modifier	Allowed Classes
Club	Crushing	8	10	One	4	0	6	1	0	Fighter Rogue Priest
Mace	Crushing	16	12	One	6	0	8	1	0	Fighter Rogue Priest
Flail	Crushing	20	14	One	8	0	10	1	0	Fighter Priest
Apprentice Staff	Magic	20	5	Two	6	0	6	1	0	Wizard
Acolyte Staff	Magic	40	8	Two	8	0	8	1	0	Wizard
Dagger	Piercing	10	3	One	4	0	6	1	0	Fighter Rogue
Short Sword	Piercing	20	10	One	6	0	8	1	0	Fighter Rogue
Long Sword	Slashing	40	15	One	10	0	10	1	0	Fighter Rogue
Broad Sword	Slashing	60	18	One	12	0	12	1	0	Fighter Rogue
Great Sword	Slashing	80	25	Two	12	0	8	2	0	Fighter

Halberd	Slashing	100	30	Two	16	0	20	1	0	Fighter
War Axe	Slashing	20	15	One	10	0	10	1	0	Fighter Rogue
Battle Axe	Slashing	50	25	Two	12	0	8	2	0	Fighter

The last thing that you want to do for this tutorial is to add the new data to the content project for the game. Open up the directory where your **Game** folder is in Windows Explorer. Now drag the **Game** folder from Windows Explorer onto the **EyesOfTheDragonContent** project to update the content for the game. If you drill down to the **Weapon** folder from the root folder **Game** you will find all of the weapons. As well, right click the **EyesOfTheDragon** project and select **Set as StartUp Project**.

I think I'm going to end this tutorial here. I wanted to expand weapons to use the new **DamageEffect** and **DamageEffectData** classes. 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