Craft III Documentation

Duncan McDonald | 11/7/2021

**Delete the explanatory text and only include your answers.**

# Object-Oriented Programming Concepts

**Define the terms and explain how they were used in your project.**

## Encapsulation

Protecting data under a single class.

public class Puppet:Item

{

private string material1 = "";

private string material2 = "";

private string material3 = "";

private string rarity = "rare";

public string Material1 { get => material1; set => material1 = value; }

public string Material2 { get => material2; set => material2 = value; }

public string Material3 { get => material3; set => material3 = value; }

public string Rarity { get => rarity; set => rarity = value; }

}

I used encapsulation to protect the data exclusive to puppets.

## Inheritance (“is a”)

Classes deriving methods or properties from parent classes

public class Item

{

private string name = "";

private double value = 0;

public string Name { get => name; set => name = value; }

public double Value { get => value; set => this.value = value; }

}

public class Material:Item

{

}

Puppets and Materials both inherit from the item class.

## Polymorphism

Ability of a method or task to behave in multiple ways.

public void Assemble(int numberInput)

{

if (CanBuild(numberInput))

{

puppetinventory.Add(puppets[numberInput]);

materialinventory.Remove(puppets[numberInput].Material1);

materialinventory.Remove(puppets[numberInput].Material2);

materialinventory.Remove(puppets[numberInput].Material3);

int rarityModifier = RandomNumber.Next(1,10);

if (rarityModifier > 3)

{

puppets[numberInput].Rarity = "common";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.1f;

}

else if (rarityModifier == 2 || rarityModifier == 3)

{

puppets[numberInput].Rarity = "uncommon";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.2f;

}

else

{

puppets[numberInput].Rarity = "rare";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.5f;

}

Print($"You have built a {puppets[numberInput].Name}!");

Console.ReadKey();

Console.Clear();

}

else

{

Print($"You cannot build a {puppets[numberInput].Name}, you are missing required materials!");

Console.ReadKey();

Console.Clear();

}

}

This method is able to generate many different puppets based off a single choice from the user.

## Separation of Concern

Definition: Separation of components into siloed actions that can function independently.

public int BuyOption()

{

int numberInput;

Print($"Select the item you wish to purchase, enter your response as a number.\nCurrent Funds: ${player.funds}");

int index = 0;

foreach (Item i in supplier.supplies)

{

Print($"{index}: {i.Name}, ${i.Value}");

index++;

}

string response = ReadLine();

numberInput = Convert.ToInt32(response);

return numberInput;

}

public bool CanBuy(int numberInput)

{

if (player.funds >= supplier.supplies[numberInput].Value)

{

return true;

}

else

{

return false;

}

}

BuyOption and CanBuy work together to run purchases but each works independent of the other.

# Generalized Puppet Algorithm

public void Assemble(int numberInput)

{

if (CanBuild(numberInput))

{

puppetinventory.Add(puppets[numberInput]);

materialinventory.Remove(puppets[numberInput].Material1);

materialinventory.Remove(puppets[numberInput].Material2);

materialinventory.Remove(puppets[numberInput].Material3);

int rarityModifier = RandomNumber.Next(1,10);

if (rarityModifier > 3)

{

puppets[numberInput].Rarity = "common";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.1f;

}

else if (rarityModifier == 2 || rarityModifier == 3)

{

puppets[numberInput].Rarity = "uncommon";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.2f;

}

else

{

puppets[numberInput].Rarity = "rare";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.5f;

}

Print($"You have built a {puppets[numberInput].Name}!");

Console.ReadKey();

Console.Clear();

}

else

{

Print($"You cannot build a {puppets[numberInput].Name}, you are missing required materials!");

Console.ReadKey();

Console.Clear();

}

}

I used this method in my code to dynamically generate any selected puppet based of the player’s choice. This framework also helped me generate functionality in the purchasing sections.

# C# Skills

**Explain how the skills below were used in your project and include code excerpts.**

## Collection (e.g., a list, array, dictionary, etc.)

public List<Material> supplies = new List<Material>()

{

new Material{Name = "sock", Value = 5},

new Material{Name = "googlyeye", Value = 5},

new Material{Name = "pompom", Value = 5},

new Material{Name = "pipecleaner", Value = 5},

new Material{Name = "toothpick", Value = 5},

new Material{Name = "felt", Value = 5},

new Material{Name = "glitter", Value = 5},

new Material{Name = "hay", Value = 5},

new Material{Name = "vynil", Value = 5}

## };

## Enum

I did not find a use for an enum in this project.

## Exception

public int BuildChoice()

{

int index = 0;

Print("Which puppet would you like to make today?");

foreach (Puppet puppet in puppets)

{

Print($"{index} " + puppet.Name + Environment.NewLine);

index++;

}

try

{

string puppetResponse = Console.ReadLine();

puppetResponse = puppetResponse.ToLower();

int numberInput = Convert.ToInt32(puppetResponse);

return numberInput;

}

catch (Exception)

{

Print("Please input your response as a number.");

Console.ReadKey();

Print("Which puppet would you like to make today?");

index = 0;

foreach (Puppet puppet in puppets)

{

Print($"{index} " + puppet.Name + Environment.NewLine);

index++;

}

string puppetResponse = Console.ReadLine();

puppetResponse = puppetResponse.ToLower();

int numberInput = Convert.ToInt32(puppetResponse);

return numberInput;

}

}

Catching the exception here helps the program with stability and helps to avoid crashes.

## External data (read in)

public static List<Puppet> LoadXML(string filename)

{

List<Puppet> puppets = new List<Puppet>();

XmlDocument doc = new XmlDocument();

doc.Load(filename);

XmlNode root = doc.DocumentElement;

XmlNodeList puppetList = root.SelectNodes("/puppets/puppet");

doc.AppendChild(root);

foreach (XmlElement x in puppetList)

{

puppets.Add(new Puppet

{

Name = x.GetAttribute("name"),

Material1 = x.GetAttribute("material1"),

Material2 = x.GetAttribute("material2"),

Material3 = x.GetAttribute("material3"),

Value = Convert.ToInt32(x.GetAttribute("value"))

});

}

return puppets;

}

I loaded data for puppets and their recipes via XML

## Delegate(s)

class Display

{

public delegate void PrintOption(string message);

public static PrintOption Print;

public static void PrintCommandLine(string message)

{

Console.WriteLine(message);

}

}

I used a delegate to create a print delegate

## Interface(s)

I did not find a use for interfaces in this project

## One of these: event handler, LINQ, or XML

public static List<Puppet> LoadXML(string filename)

{

List<Puppet> puppets = new List<Puppet>();

XmlDocument doc = new XmlDocument();

doc.Load(filename);

XmlNode root = doc.DocumentElement;

XmlNodeList puppetList = root.SelectNodes("/puppets/puppet");

doc.AppendChild(root);

foreach (XmlElement x in puppetList)

{

puppets.Add(new Puppet

{

Name = x.GetAttribute("name"),

Material1 = x.GetAttribute("material1"),

Material2 = x.GetAttribute("material2"),

Material3 = x.GetAttribute("material3"),

Value = Convert.ToInt32(x.GetAttribute("value"))

});

}

return puppets;

}

I used XML to load in the data, this is helpful because it makes altering and editing data simple and easy. I can also edit data outside of visual studio without modifying code.

# Required Functionality

**Include code excerpts and explain how you completed the required functionality.**

## Supplier

public List<Material> supplies = new List<Material>()

{

new Material{Name = "sock", Value = 5},

new Material{Name = "googlyeye", Value = 5},

new Material{Name = "pompom", Value = 5},

new Material{Name = "pipecleaner", Value = 5},

new Material{Name = "toothpick", Value = 5},

new Material{Name = "felt", Value = 5},

new Material{Name = "glitter", Value = 5},

new Material{Name = "hay", Value = 5},

new Material{Name = "vynil", Value = 5}

};

public void Purchase()

{

int numberInput = BuyOption();

if (CanBuy(numberInput))

{

Print($"Congrats, you purchased a {supplier.supplies[numberInput].Name}");

ReadKey();

materialinventory.Add(supplier.supplies[numberInput].Name);

supplier.supplies.Remove(supplier.supplies[numberInput]);

player.funds = player.funds - supplier.supplies[numberInput].Value;

}

else

{

Print("You do not have the necessary funds.");

ReadKey();

}

}

The supplier has a list of available materials, the player can use their funds to trade for materials

## Customer

public void Sell()

{

int numberInput = SellOption();

Print($"Congratulations! You sold a {puppetinventory[numberInput].Name} for ${Math.Round(puppetinventory[numberInput].Value, 2)}.\nTotal Profit on puppet: ${Math.Round(puppetinventory[numberInput].Value - baseValue, 2)}");

//profit %

double profit = puppetinventory[numberInput].Value - baseValue;

double percentProfit = Math.Round((profit / puppetinventory[numberInput].Value) \* 100, 2);

Print($"\nProfit Margin: {percentProfit}%\n");

player.funds += puppetinventory[numberInput].Value;

ReadKey();

puppetinventory.Remove(puppetinventory[numberInput]);

}

The customer simply takes your unwanted puppets and provides funds. At this time, the customer does not store or keep any puppets.

## Profit Margin

public void Sell()

{

int numberInput = SellOption();

Print($"Congratulations! You sold a {puppetinventory[numberInput].Name} for ${Math.Round(puppetinventory[numberInput].Value, 2)}.\nTotal Profit on puppet: ${Math.Round(puppetinventory[numberInput].Value - baseValue, 2)}");

//profit %

double profit = puppetinventory[numberInput].Value - baseValue;

double percentProfit = Math.Round((profit / puppetinventory[numberInput].Value) \* 100, 2);

Print($"\nProfit Margin: {percentProfit}%\n");

player.funds += puppetinventory[numberInput].Value;

ReadKey();

puppetinventory.Remove(puppetinventory[numberInput]);

}

I calculated the profit margin by calculating profit/value \* 100, this shows the percent profit of the sale.

## Probability

int rarityModifier = RandomNumber.Next(1,10);

if (rarityModifier > 3)

{

puppets[numberInput].Rarity = "common";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.1f;

}

else if (rarityModifier == 2 || rarityModifier == 3)

{

puppets[numberInput].Rarity = "uncommon";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.2f;

}

else

{

puppets[numberInput].Rarity = "rare";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.5f;

}

I used a random number to create a simple probability if statement waterfall to assign rarity(magical component) based on percentages.

Magical component.

int rarityModifier = RandomNumber.Next(1,10);

if (rarityModifier > 3)

{

puppets[numberInput].Rarity = "common";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.1f;

}

else if (rarityModifier == 2 || rarityModifier == 3)

{

puppets[numberInput].Rarity = "uncommon";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.2f;

}

else

{

puppets[numberInput].Rarity = "rare";

puppets[numberInput].Value = puppets[numberInput].Value \* 1.5f;

}

The magic component of the puppets is their rarity level. Each puppet has a rarity that can be viewed in inventory. Rare/Magic puppets sell for more than normal puppets.

# UML Diagrams



# UML Diagram Explanation

My UML needs to be reworked along with the code. Currently, there are many methods and collections that reside in classes that they should not. I need to refactor selling and buying methods into the supplier and customer classes. Similarly, the build methods need to reside in the player class. I spent a lot of time working on functionality so the project is not organized as it should be.

# Credits

Select code aspects derived from in class examples.