Craft Project Documentation

Meghan Patnode April 30th, 2024

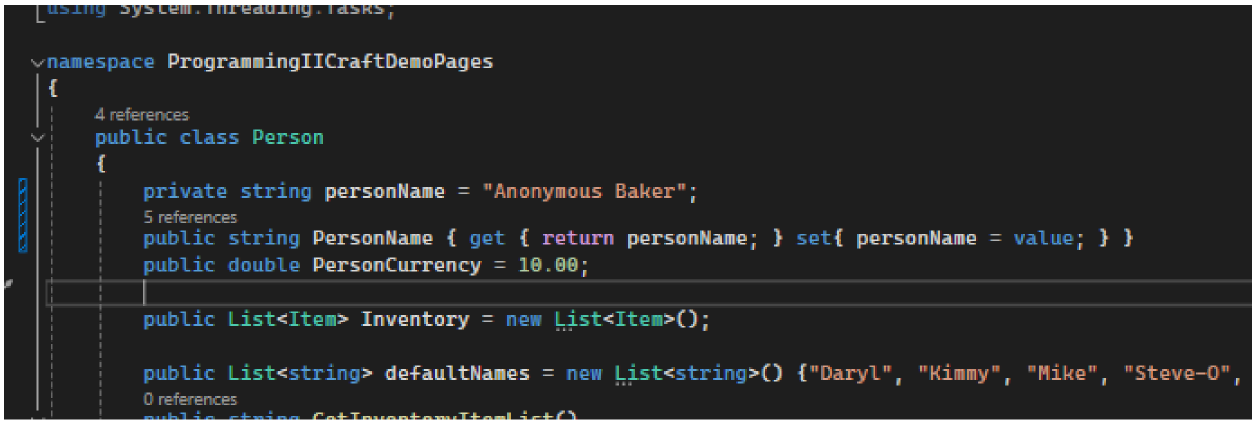
Object-Oriented Programming Concepts

Encapsulation

Definition:

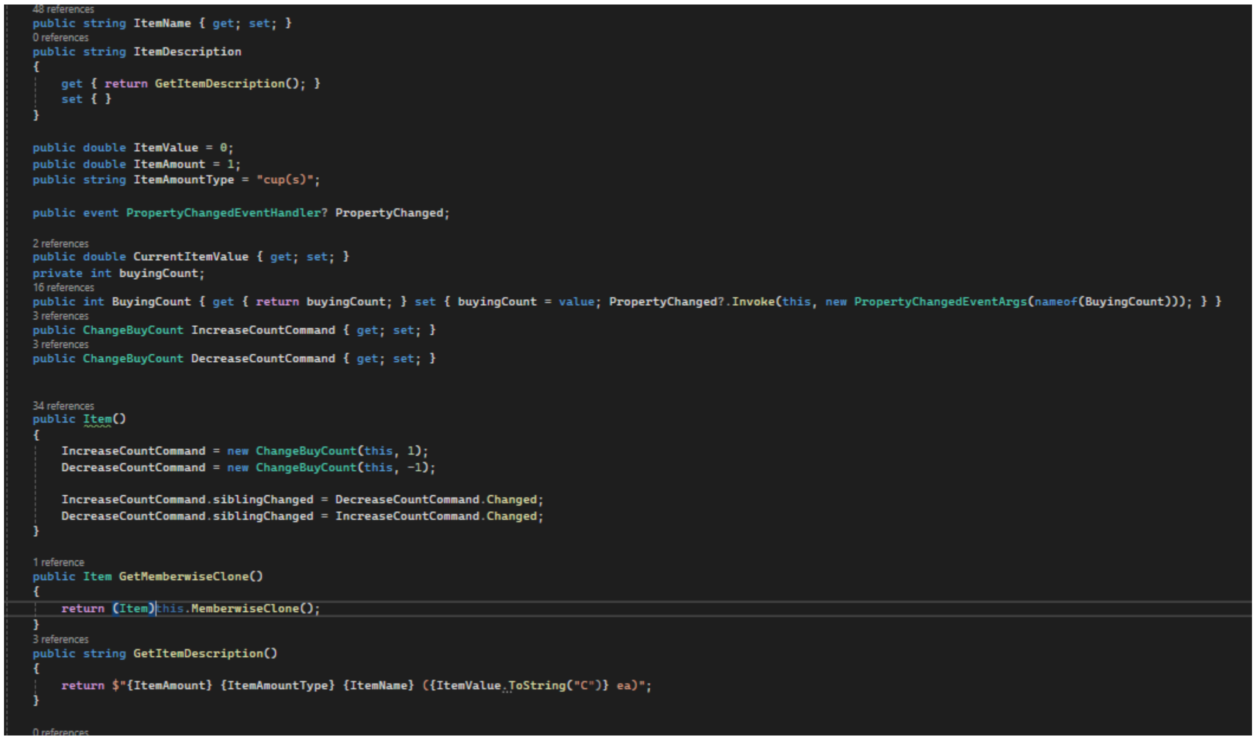
Encapsulation is the practice of containing data in one class, in order to prevent other classes from changing that data and to provide a level of abstraction that helps when/if fields are changed later in a project. This is done by creating a private field that stores the actual information being protected and a public property that uses getters and setters to access the information in the private field. It is best practice to keep fields private when possible, so that private information is protected, and values are not unintentionally changed.

Brief code excerpt(s) from your project:



```
using System.Threading.Tasks;

namespace ProgrammingIICraftDemoPages
{
    4 references
    public class Person
    {
        private string personName = "Anonymous Baker";
        5 references
        public string PersonName { get { return personName; } set{ personName = value; } }
        public double PersonCurrency = 10.00;

        public List<Item> Inventory = new List<Item>();

        public List<string> defaultNames = new List<string>() {"Daryl", "Kimmy", "Mike", "Steve-O",
        0 references
        public string GetInventoryItemList()
```

^ Person class. Encapsulation Image 1.



```
48 references
public string ItemName { get; set; }
0 references
public string ItemDescription
{
    get { return GetItemDescription(); }
    set { }
}

public double ItemValue = 0;
public double ItemAmount = 1;
public string ItemAmountType = "cup(s)";

public event PropertyChangedEventHandler? PropertyChanged;

2 references
public double CurrentItemValue { get; set; }
private int buyingCount;
16 references
public int BuyingCount { get { return buyingCount; } set { buyingCount = value; PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(BuyingCount))); } }
3 references
public ChangeBuyCount IncreaseCountCommand { get; set; }
3 references
public ChangeBuyCount DecreaseCountCommand { get; set; }


34 references
public Item()
{
    IncreaseCountCommand = new ChangeBuyCount(this, 1);
    DecreaseCountCommand = new ChangeBuyCount(this, -1);

    IncreaseCountCommand.siblingChanged = DecreaseCountCommand.Changed;
    DecreaseCountCommand.siblingChanged = IncreaseCountCommand.Changed;
}

1 reference
public Item GetMemberwiseClone()
{
    return (Item)this.MemberwiseClone();
}
3 references
public string GetItemDescription()
{
    return $"{ItemAmount} {ItemAmountType} {ItemName} ({ItemValue.ToString("C")} ea)";
}

0 references
```

^ Item class. Encapsulation Image 2.

Explain usage in your project:

In Image 1, Encapsulation is used to hide information from other classes so that the player's name remains only accessible through the public version with getters and setters. I found the player's/person's name to be best suited for this because it is only changed once by the start-up window (therefore, this makes it harder to accidentally change) and all other usages are read-only. Also, the person name is sensitive information given by the player.

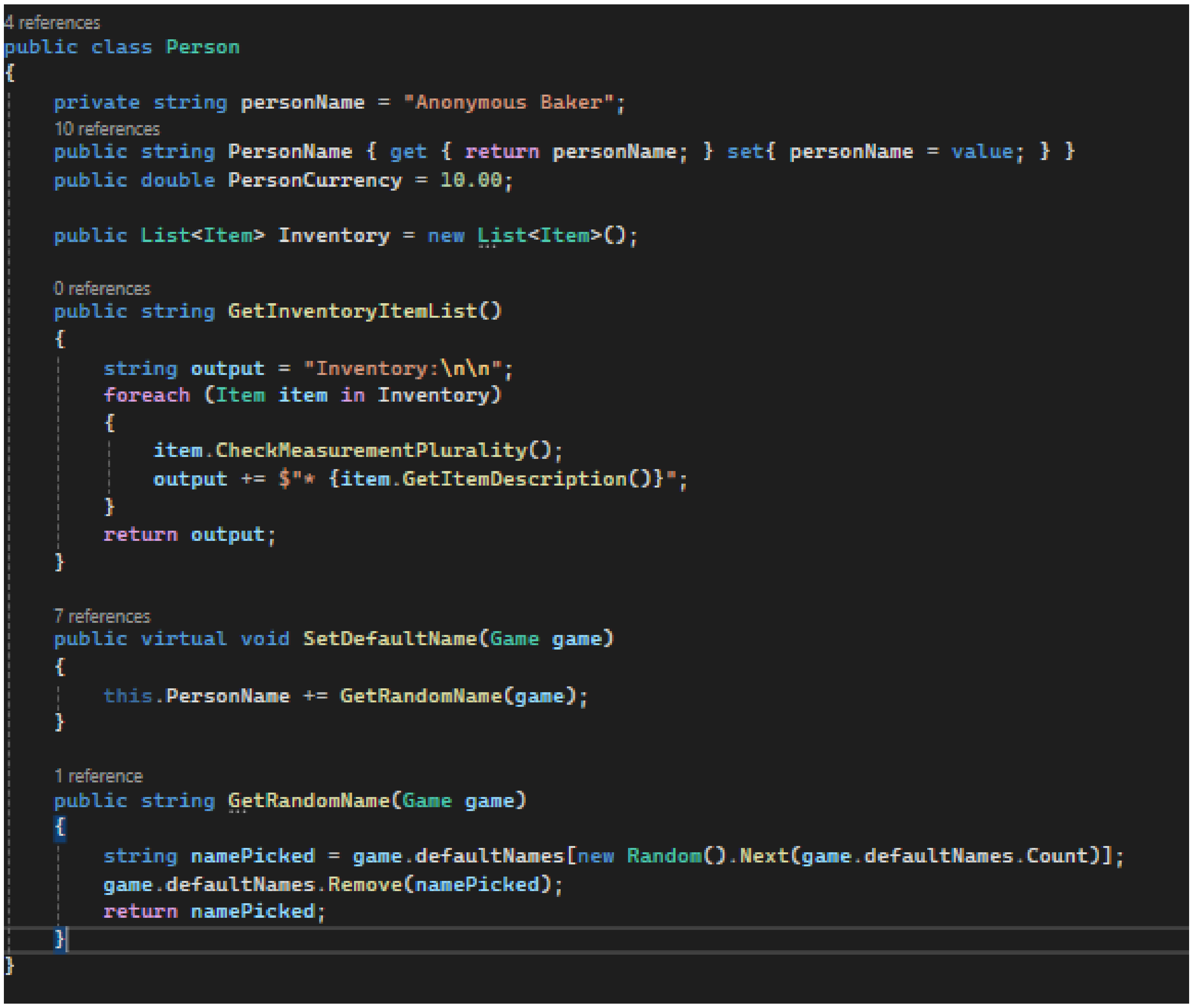
In Image 2, getters and setters are used for item description because the set allows access to all of the fields in item needed for listing the ItemDescription, in a read-only format. This makes the information pulled to create the ItemDescription (when writing the description) impossible to change from the class accessing it. Additionally, it simplifies the code so that item description is only written once.

Inheritance ("is a")

Definition:

Inheritance is used when a class "is" also another class. This is shown in a parent/child relationship. A child class inherits all of the aspects from a parent class, and all of its own new ones. This is good for when something is a more specific version of a blueprint. For example, a class for "Moby Dick" would be a child and a "book" class would be a parent.

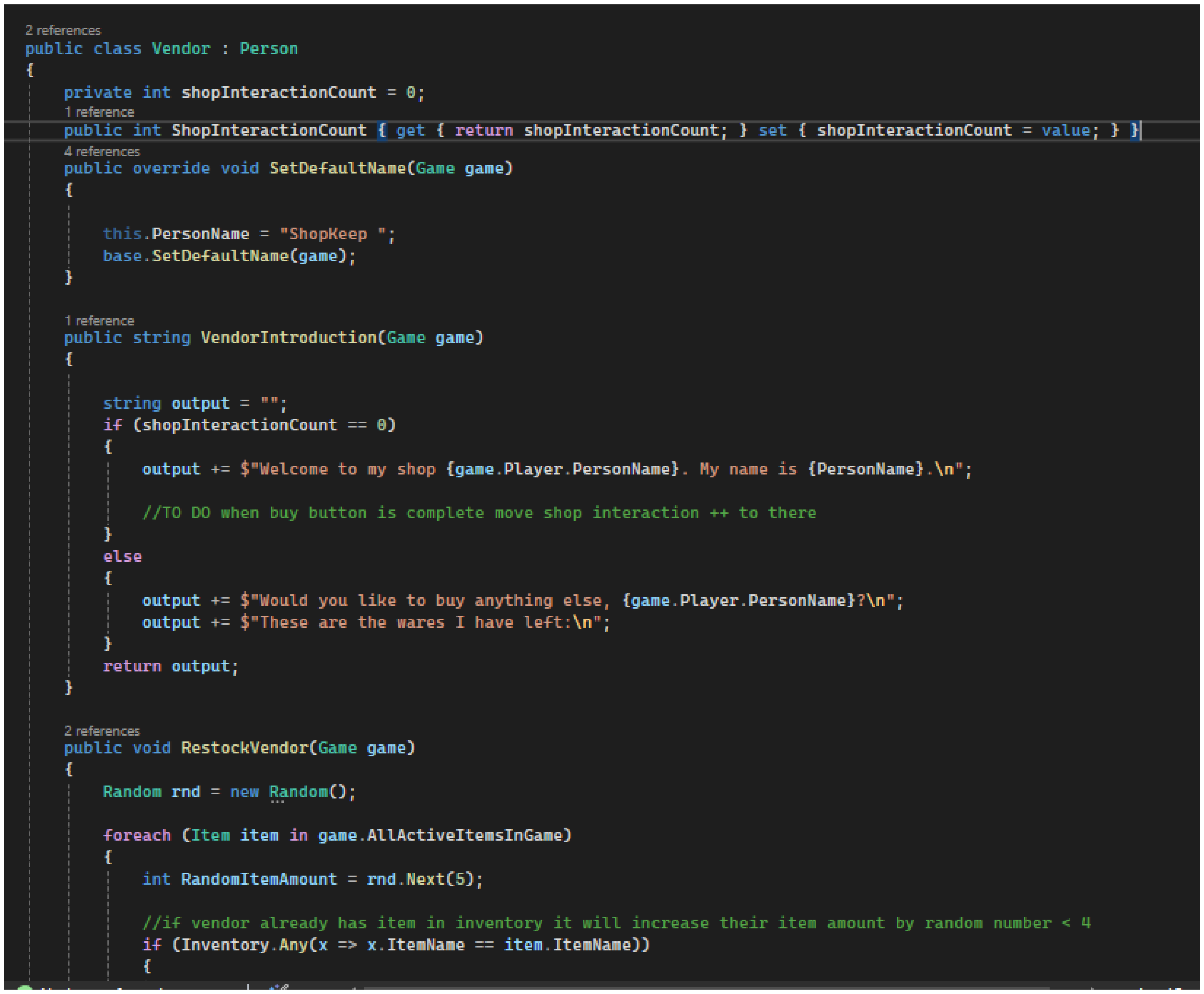
Brief code excerpt(s) from your project:



```
4 references
public class Person
{
    private string personName = "Anonymous Baker";
    10 references
    public string PersonName { get { return personName; } set{ personName = value; } }
    public double PersonCurrency = 10.00;

    public List<Item> Inventory = new List<Item>();

    0 references
    public string GetInventoryItemList()
    {
        string output = "Inventory:\n\n";
        foreach (Item item in Inventory)
        {
            item.CheckMeasurementPlurality();
            output += $"* {item.GetItemDescription()}";
        }
        return output;
    }

    7 references
    public virtual void SetDefaultName(Game game)
    {
        this.PersonName += GetRandomName(game);
    }

    1 reference
    public string GetRandomName(Game game)
    {
        string namePicked = game.defaultNames[new Random().Next(game.defaultNames.Count)];
        game.defaultNames.Remove(namePicked);
        return namePicked;
    }
}
```

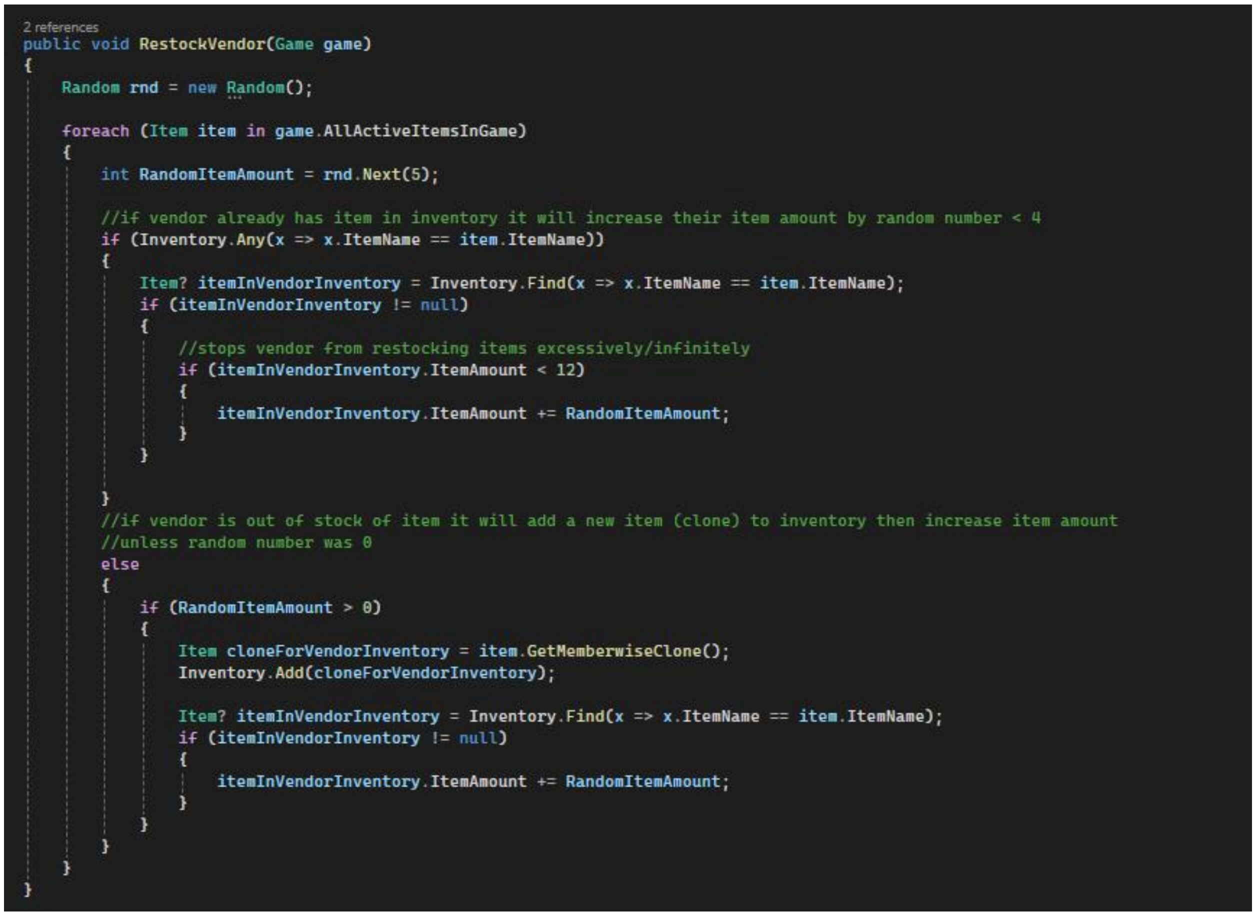
[1]^ *Person class that the Customer and Vendor classes both inherit from. Inheritance Image 1*

1



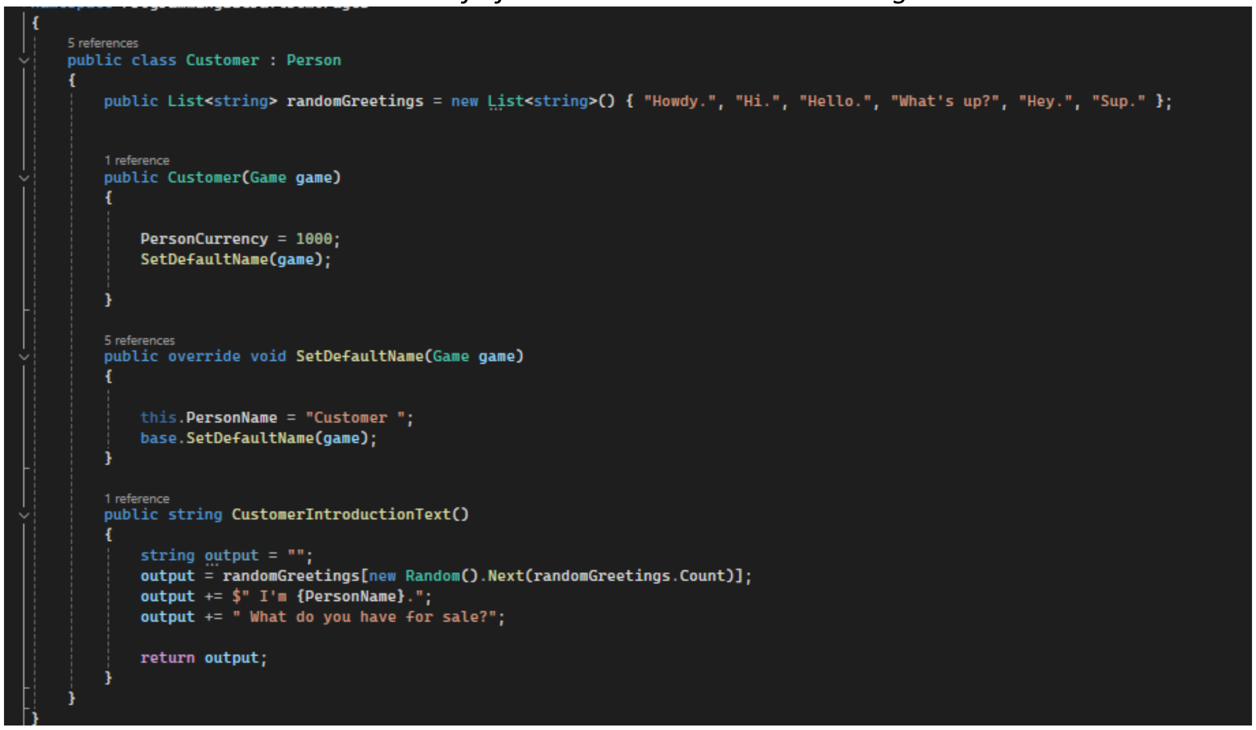
```
2 references
public class Vendor : Person
{
    private int shopInteractionCount = 0;
    1 reference
    public int ShopInteractionCount { get { return shopInteractionCount; } set { shopInteractionCount = value; } }
    4 references
    public override void SetDefaultName(Game game)
    {

        this.PersonName = "ShopKeep ";
        base.SetDefaultName(game);
    }

    1 reference
    public string VendorIntroduction(Game game)
    {

        string output = "";
        if (shopInteractionCount == 0)
        {
            output += $"Welcome to my shop {game.Player.PersonName}. My name is {PersonName}.\n";

            //TO DO when buy button is complete move shop interaction ++ to there
        }
        else
        {
            output += $"Would you like to buy anything else, {game.Player.PersonName}?\n";
            output += $"These are the wares I have left:\n";
        }
        return output;
    }

    2 references
    public void RestockVendor(Game game)
    {
        Random rnd = new Random();

        foreach (Item item in game.AllActiveItemsInGame)
        {
            int RandomItemAmount = rnd.Next(5);

            //if vendor already has item in inventory it will increase their item amount by random number < 4
            if (Inventory.Any(x => x.ItemName == item.ItemName))
            {
```

^ Top half of vendor class. Inheritance Image 2.



```
2 references
public void RestockVendor(Game game)
{
    Random rnd = new Random();

    foreach (Item item in game.AllActiveItemsInGame)
    {
        int RandomItemAmount = rnd.Next(5);

        //if vendor already has item in inventory it will increase their item amount by random number < 4
        if (Inventory.Any(x => x.ItemName == item.ItemName))
        {
            Item? itemInVendorInventory = Inventory.Find(x => x.ItemName == item.ItemName);
            if (itemInVendorInventory != null)
            {
                //stops vendor from restocking items excessively/infinitely
                if (itemInVendorInventory.ItemAmount < 12)
                {
                    itemInVendorInventory.ItemAmount += RandomItemAmount;
                }
            }

        }
        //if vendor is out of stock of item it will add a new item (clone) to inventory then increase item amount
        //unless random number was 0
        else
        {
            if (RandomItemAmount > 0)
            {
                Item cloneForVendorInventory = item.GetMemberwiseClone();
                Inventory.Add(cloneForVendorInventory);

                Item? itemInVendorInventory = Inventory.Find(x => x.ItemName == item.ItemName);
                if (itemInVendorInventory != null)
                {
                    itemInVendorInventory.ItemAmount += RandomItemAmount;
                }
            }
        }
    }
}
```

^ Second half of vendor class. Inheritance Image 3.



```
{
    5 references
    public class Customer : Person
    {
        public List<string> randomGreetings = new List<string>() { "Howdy.", "Hi.", "Hello.", "What's up?", "Hey.", "Sup." };


        1 reference
        public Customer(Game game)
        {

            PersonCurrency = 1000;
            SetDefaultName(game);

        }

        5 references
        public override void SetDefaultName(Game game)
        {

            this.PersonName = "Customer ";
            base.SetDefaultName(game);
        }

        1 reference
        public string CustomerIntroductionText()
        {
            string output = "";
            output = randomGreetings[new Random().Next(randomGreetings.Count)];
            output += $" I'm {PersonName}.";
            output += " What do you have for sale?";

            return output;
        }
    }
}
```

^ Customer class, child of person class. Inheritance Image 4.

Explain usage in your project:

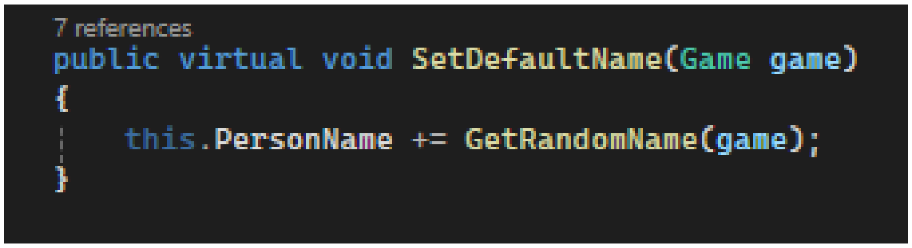
In Image 1, the parent class, Person is shown. Image 2-4 show the two classes that inherit from the Person class, Vendor and Customer. This made sense because both of these child classes needed the contents of the parent class, such as the name setup, but also needed unique functionality. For example, the vendor child class also contains the functionality for restocking the vendor's inventory, something neither the player or the customer would need (which are the other two instances in which the person class is used). This is especially useful with the customer class, because a new customer is generated each time the player needs to sell items. Therefore, if the customer class did not exist, things like the CustomerIntroductionText would need to exist in the Person class (not useful to other persons) and the variables like the customer's currency (which is set to be a number the player cannot possibly max out in sales, a unique need for the customer) would need to be set each time this customer is generated. Ultimately, the use of inheritance simplifies the code, allowing for more reusability.

Polymorphism

Definition:

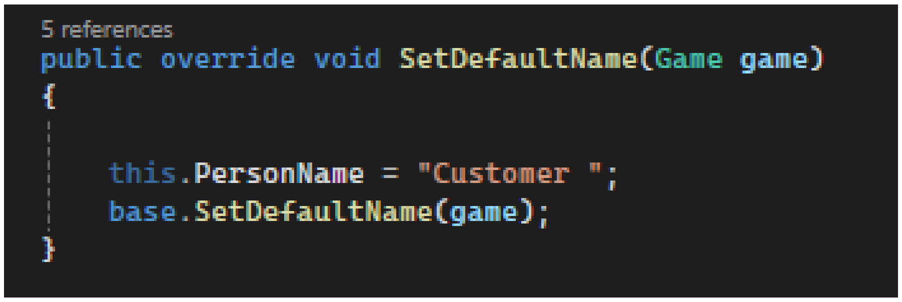
Polymorphism goes hand in hand with Inheritance. Something is polymorphic when it is inherited from a parent class, but utilizes an override method. A base or parent class can have a method with a virtual keyword, and a child class can have that same method with the override keyword. If the child is called, the override method will be used instead of the virtual one at runtime. This allows for classes to break uniformity in order to have specifics related to themselves, but still reuse what is needed from the parent class.

Brief code excerpt(s) from your project:



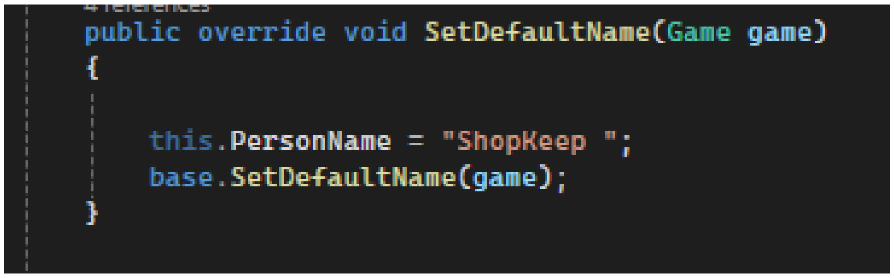
```
7 references
public virtual void SetDefaultName(Game game)
{
    this.PersonName += GetRandomName(game);
}
```

^ Virtual class inside person. Polymorphism Image 1.



```
5 references
public override void SetDefaultName(Game game)
{

    this.PersonName = "Customer ";
    base.SetDefaultName(game);
}
```

^ Customer class override. Polymorphism Image 2.



```
    public override void SetDefaultName(Game game)
    {

        this.PersonName = "ShopKeep ";
        base.SetDefaultName(game);
    }
```

^ Vendor class override. Polymorphism Image 3.

Explain usage in your project:

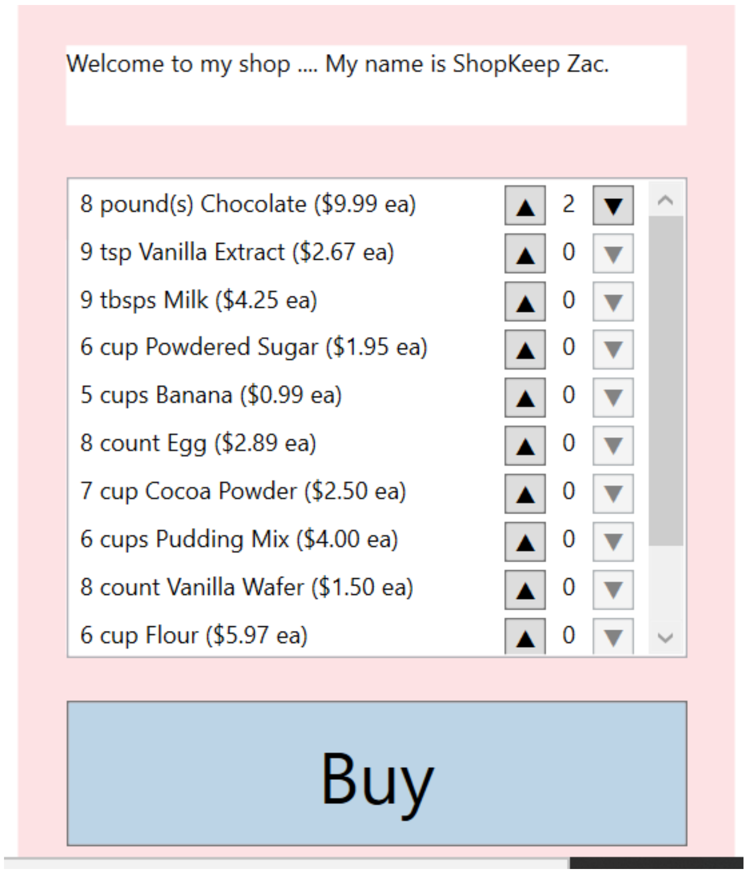
In Image 1, the virtual method inside the Person class is used to set the name of the Person to one of the default names provided from a list, this is used for almost all persons. However, the usage of the override inside the Vendor and Customer class allows the prefix of either Shopkeep or Customer to be added. Then, the normal method is from the person class is run. This way, the Vendor and Customer class can reuse the code needed from the person class (pulling and setting a random name) but still include the functionality specific to the child class, without creating an entirely new function.

Separation of Concern

Definition:

Separation of Concern (SoC) is a design principle in which importance is placed on separating each part of the code (In C# typically classes) into distinct pieces that each individually address one singular concern of the software. Breaking code into parts like this allows for easier time in development and leads to more reusable code. Additionally, it makes it easier to troubleshoot issues within the code (i.e. if you are experiencing an issue, there’s only one location that is able to cause said issue).

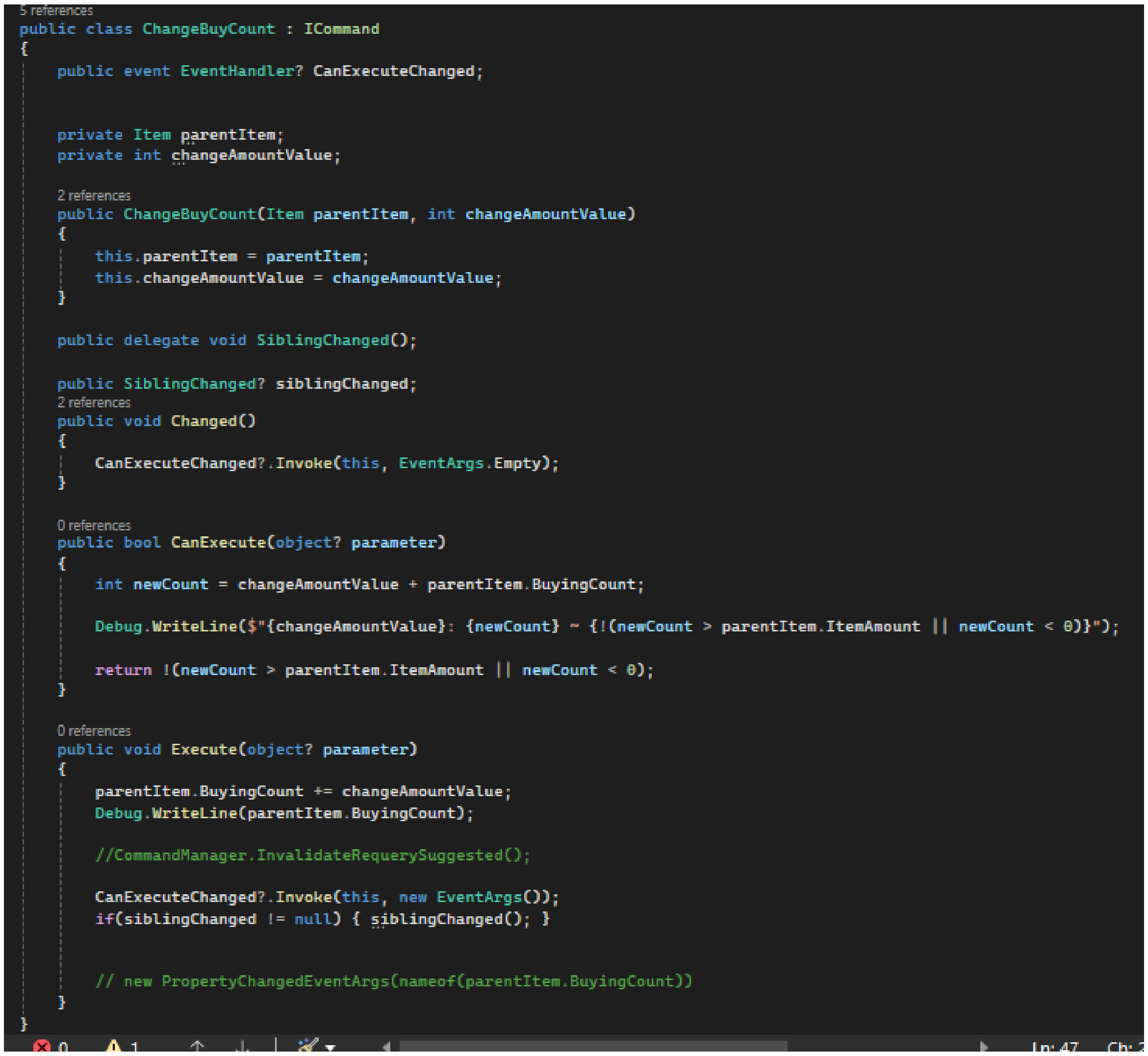
Brief code excerpt(s) from your project:



Welcome to my shop .... My name is ShopKeep Zac.

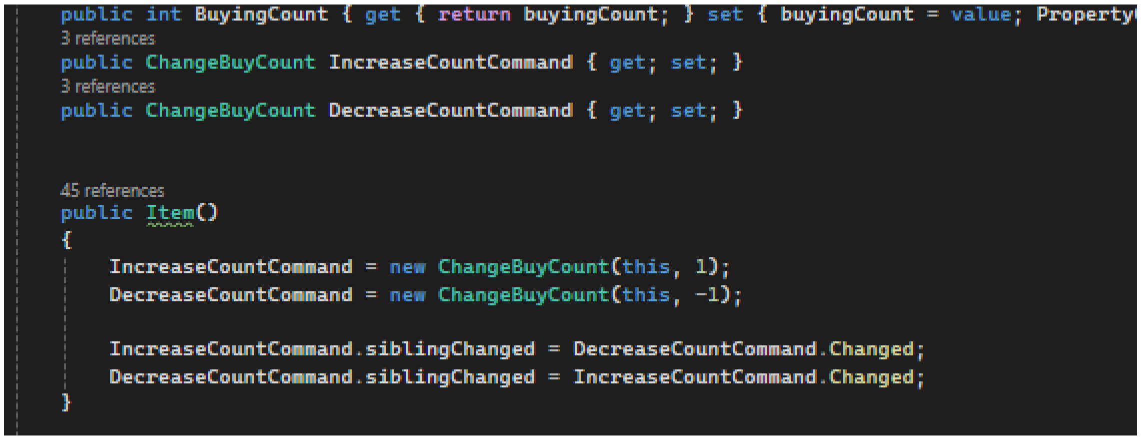
```
8 pound(s) Chocolate ($9.99 ea)      ▲ 2 ▼
9 tsp Vanilla Extract ($2.67 ea)     ▲ 0 ▼
9 tbsps Milk ($4.25 ea)              ▲ 0 ▼
6 cup Powdered Sugar ($1.95 ea)      ▲ 0 ▼
5 cups Banana ($0.99 ea)             ▲ 0 ▼
8 count Egg ($2.89 ea)               ▲ 0 ▼
7 cup Cocoa Powder ($2.50 ea)        ▲ 0 ▼
6 cups Pudding Mix ($4.00 ea)        ▲ 0 ▼
8 count Vanilla Wafer ($1.50 ea)     ▲ 0 ▼
6 cup Flour ($5.97 ea)               ▲ 0 ▼
```

^ Trade menu in final product, increment/decrement buttons explained below shown on right side. SoC Image 1



```
5 references
public class ChangeBuyCount : ICommand
{
    public event EventHandler? CanExecuteChanged;


    private Item parentItem;
    private int changeAmountValue;

    2 references
    public ChangeBuyCount(Item parentItem, int changeAmountValue)
    {
        this.parentItem = parentItem;
        this.changeAmountValue = changeAmountValue;
    }

    public delegate void SiblingChanged();

    public SiblingChanged? siblingChanged;
    2 references
    public void Changed()
    {
        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }

    0 references
    public bool CanExecute(object? parameter)
    {
        int newCount = changeAmountValue + parentItem.BuyingCount;

        Debug.WriteLine($"{changeAmountValue}: {newCount} ~ {!(newCount > parentItem.ItemAmount || newCount < 0)}");

        return !(newCount > parentItem.ItemAmount || newCount < 0);
    }

    0 references
    public void Execute(object? parameter)
    {
        parentItem.BuyingCount += changeAmountValue;
        Debug.WriteLine(parentItem.BuyingCount);

        //CommandManager.InvalidateRequerySuggested();

        CanExecuteChanged?.Invoke(this, new EventArgs());
        if(siblingChanged != null) { siblingChanged(); }


        // new PropertyChangedEventArgs(nameof(parentItem.BuyingCount))
    }
}
```

^ ChangeBuyCount contained in the Item script. SoC Image 2.



```
    public int BuyingCount { get { return buyingCount; } set { buyingCount = value; Property
    3 references
    public ChangeBuyCount IncreaseCountCommand { get; set; }
    3 references
    public ChangeBuyCount DecreaseCountCommand { get; set; }


    45 references
    public Item()
    {
        IncreaseCountCommand = new ChangeBuyCount(this, 1);
        DecreaseCountCommand = new ChangeBuyCount(this, -1);

        IncreaseCountCommand.siblingChanged = DecreaseCountCommand.Changed;
        DecreaseCountCommand.siblingChanged = IncreaseCountCommand.Changed;
    }
```

^ ChangeBuyCount called in Item class, which is binded to the buttons in the XAML. SoC Image 3.

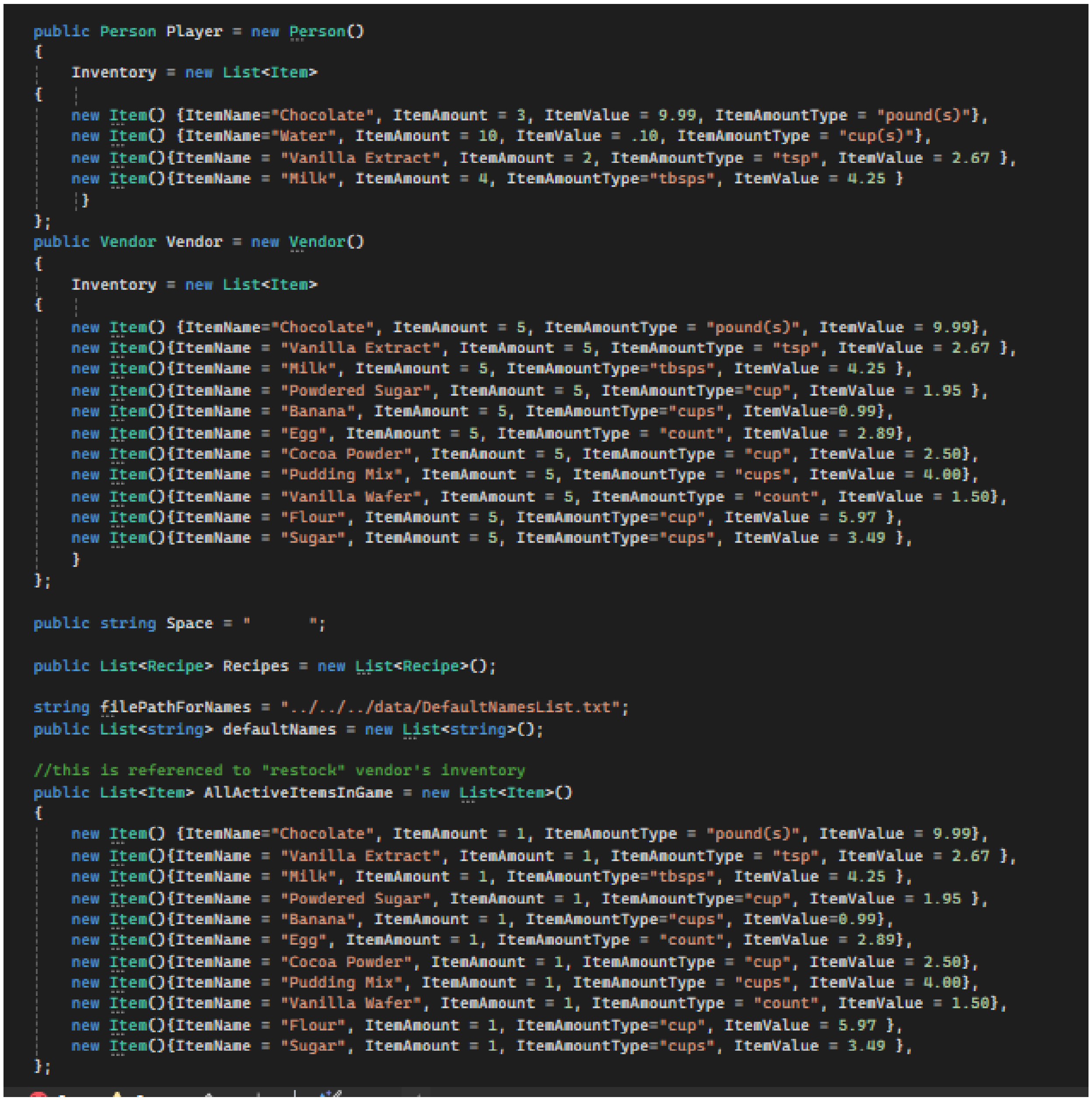
Explain usage in your project:

The functionality of the increase and decrease buttons on the Trade.XAML page is separated from the XAML.cs file so that the functionality for incrementing/decrementing the buying count of the item is contained entirely inside the item itself. This way, the actual XAML file for the Trade page is only concerned with the visuals of the software, whereas the item class is focused on the item's functionality and logic.

C# Programming Skills

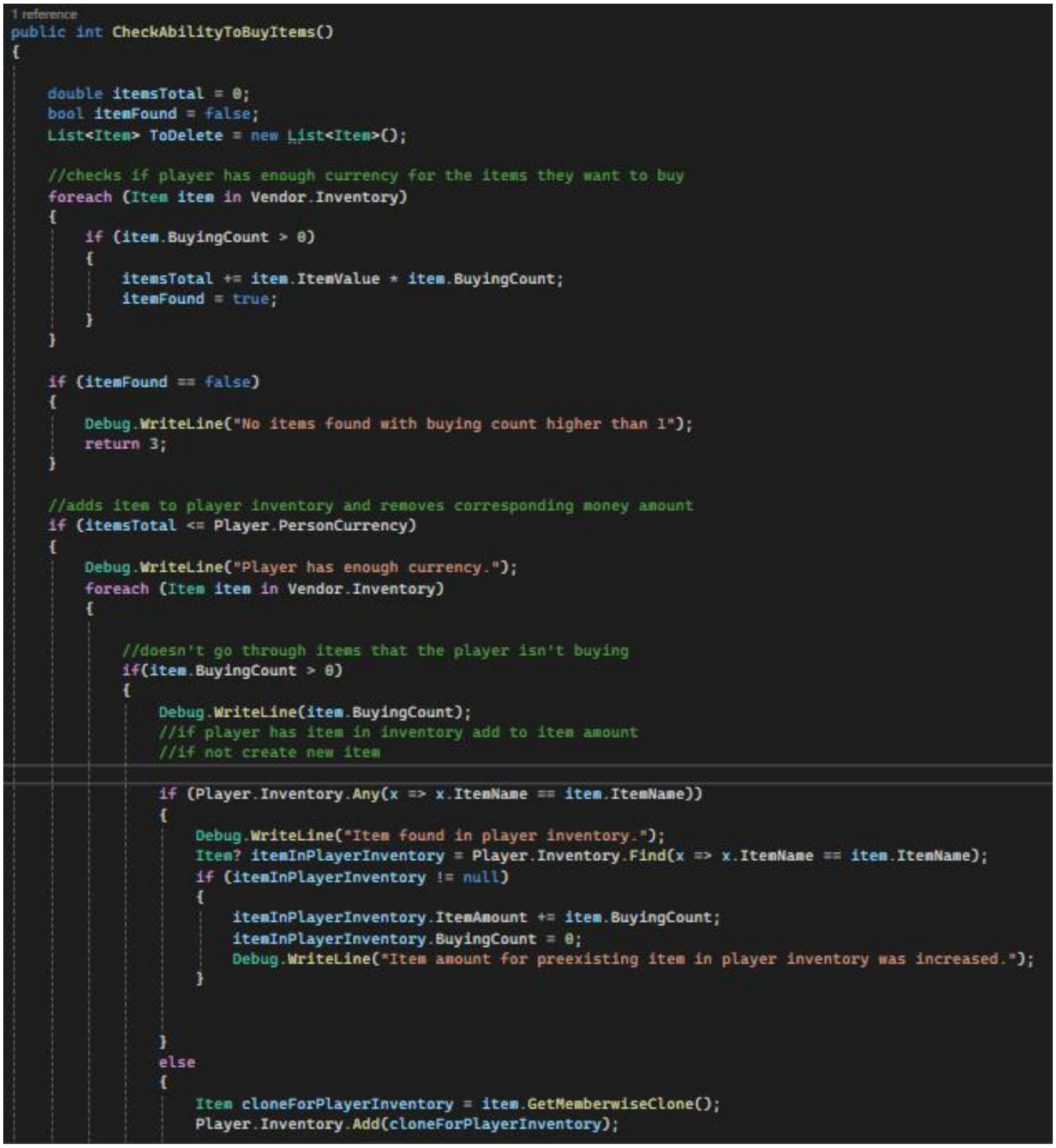
Collection (e.g., a list, array, dictionary, etc.)

Brief code excerpt(s) from your project:



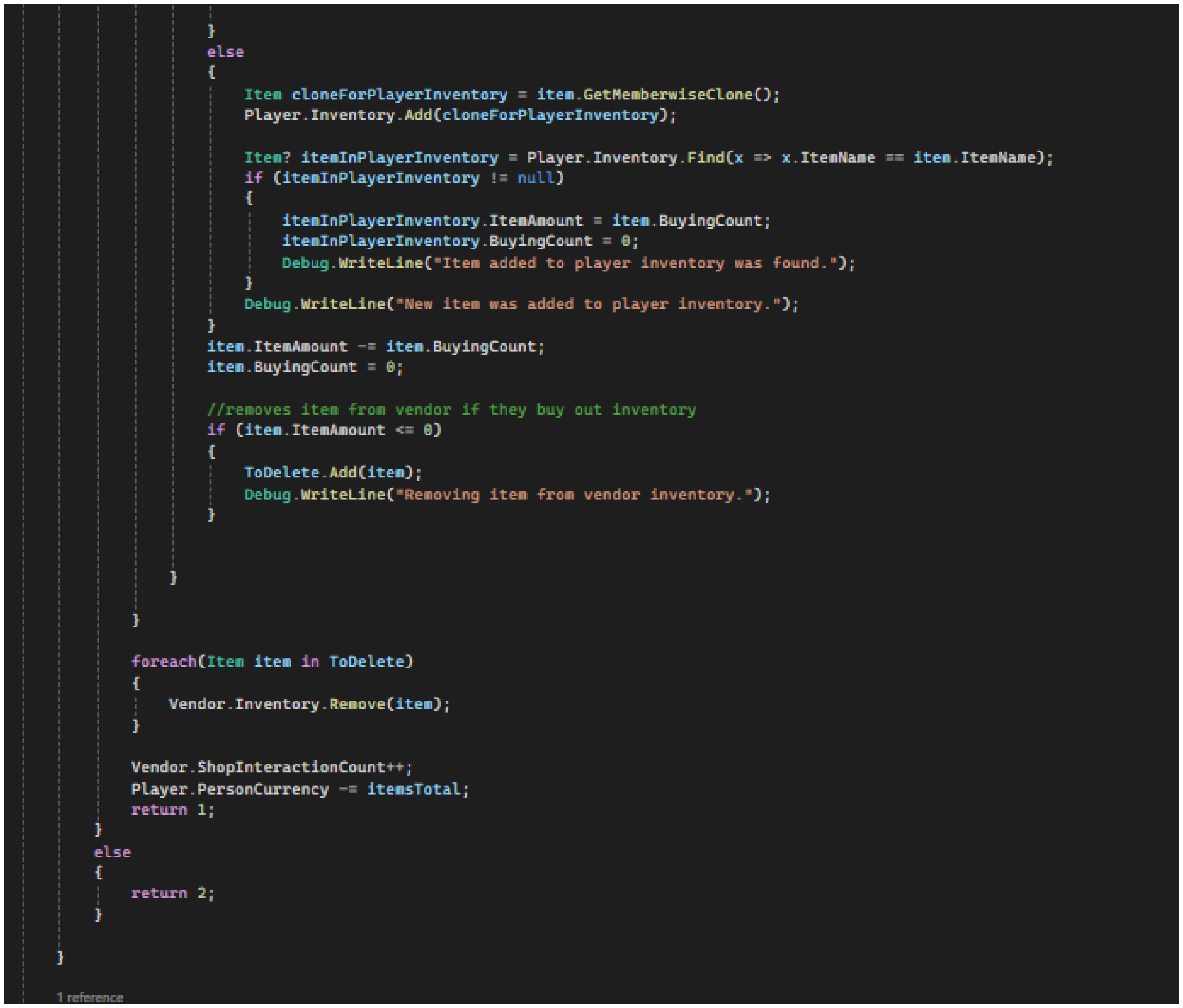
```
public Person Player = new Person()
{
    Inventory = new List<Item>
{
    new Item() {ItemName="Chocolate", ItemAmount = 3, ItemValue = 9.99, ItemAmountType = "pound(s)"},
    new Item() {ItemName="Water", ItemAmount = 10, ItemValue = .10, ItemAmountType = "cup(s)"},
    new Item(){ItemName = "Vanilla Extract", ItemAmount = 2, ItemAmountType = "tsp", ItemValue = 2.67 },
    new Item(){ItemName = "Milk", ItemAmount = 4, ItemAmountType="tbsps", ItemValue = 4.25 }
    }
};
public Vendor Vendor = new Vendor()
{
    Inventory = new List<Item>
{
    new Item() {ItemName="Chocolate", ItemAmount = 5, ItemAmountType = "pound(s)", ItemValue = 9.99},
    new Item(){ItemName = "Vanilla Extract", ItemAmount = 5, ItemAmountType = "tsp", ItemValue = 2.67 },
    new Item(){ItemName = "Milk", ItemAmount = 5, ItemAmountType="tbsps", ItemValue = 4.25 },
    new Item(){ItemName = "Powdered Sugar", ItemAmount = 5, ItemAmountType="cup", ItemValue = 1.95 },
    new Item(){ItemName = "Banana", ItemAmount = 5, ItemAmountType="cups", ItemValue=0.99},
    new Item(){ItemName = "Egg", ItemAmount = 5, ItemAmountType = "count", ItemValue = 2.89},
    new Item(){ItemName = "Cocoa Powder", ItemAmount = 5, ItemAmountType = "cup", ItemValue = 2.50},
    new Item(){ItemName = "Pudding Mix", ItemAmount = 5, ItemAmountType = "cups", ItemValue = 4.00},
    new Item(){ItemName = "Vanilla Wafer", ItemAmount = 5, ItemAmountType = "count", ItemValue = 1.50},
    new Item(){ItemName = "Flour", ItemAmount = 5, ItemAmountType="cup", ItemValue = 5.97 },
    new Item(){ItemName = "Sugar", ItemAmount = 5, ItemAmountType="cups", ItemValue = 3.49 },
    }
};

public string Space = "      ";

public List<Recipe> Recipes = new List<Recipe>();

string filePathForNames = "../../../data/DefaultNamesList.txt";
public List<string> defaultNames = new List<string>();

//this is referenced to "restock" vendor's inventory
public List<Item> AllActiveItemsInGame = new List<Item>()
{
    new Item() {ItemName="Chocolate", ItemAmount = 1, ItemAmountType = "pound(s)", ItemValue = 9.99},
    new Item(){ItemName = "Vanilla Extract", ItemAmount = 1, ItemAmountType = "tsp", ItemValue = 2.67 },
    new Item(){ItemName = "Milk", ItemAmount = 1, ItemAmountType="tbsps", ItemValue = 4.25 },
    new Item(){ItemName = "Powdered Sugar", ItemAmount = 1, ItemAmountType="cup", ItemValue = 1.95 },
    new Item(){ItemName = "Banana", ItemAmount = 1, ItemAmountType="cups", ItemValue=0.99},
    new Item(){ItemName = "Egg", ItemAmount = 1, ItemAmountType = "count", ItemValue = 2.89},
    new Item(){ItemName = "Cocoa Powder", ItemAmount = 1, ItemAmountType = "cup", ItemValue = 2.50},
    new Item(){ItemName = "Pudding Mix", ItemAmount = 1, ItemAmountType = "cups", ItemValue = 4.00},
    new Item(){ItemName = "Vanilla Wafer", ItemAmount = 1, ItemAmountType = "count", ItemValue = 1.50},
    new Item(){ItemName = "Flour", ItemAmount = 1, ItemAmountType="cup", ItemValue = 5.97 },
    new Item(){ItemName = "Sugar", ItemAmount = 1, ItemAmountType="cups", ItemValue = 3.49 },
};
```

^ Vendor, Player, and ActiveItems Lists. Collection Image 1.



```
1 reference
public int CheckAbilityToBuyItems()
{

    double itemsTotal = 0;
    bool itemFound = false;
    List<Item> ToDelete = new List<Item>();

    //checks if player has enough currency for the items they want to buy
    foreach (Item item in Vendor.Inventory)
    {
        if (item.BuyingCount > 0)
        {
            itemsTotal += item.ItemValue * item.BuyingCount;
            itemFound = true;
        }
    }

    if (itemFound == false)
    {
        Debug.WriteLine("No items found with buying count higher than 1");
        return 3;
    }

    //adds item to player inventory and removes corresponding money amount
    if (itemsTotal <= Player.PersonCurrency)
    {
        Debug.WriteLine("Player has enough currency.");
        foreach (Item item in Vendor.Inventory)
        {

            //doesn't go through items that the player isn't buying
            if(item.BuyingCount > 0)
            {
                Debug.WriteLine(item.BuyingCount);
                //if player has item in inventory add to item amount
                //if not create new item

                if (Player.Inventory.Any(x => x.ItemName == item.ItemName))
                {
                    Debug.WriteLine("Item found in player inventory.");
                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == item.ItemName);
                    if (itemInPlayerInventory != null)
                    {
                        itemInPlayerInventory.ItemAmount += item.BuyingCount;
                        itemInPlayerInventory.BuyingCount = 0;
                        Debug.WriteLine("Item amount for preexisting item in player inventory was increased.");
                    }


                }
                else
                {
                    Item cloneForPlayerInventory = item.GetMemberwiseClone();
                    Player.Inventory.Add(cloneForPlayerInventory);
```

^ Collection usage in CheckAbilityToBuyItems Method (First Half). Collection Image 2.



```
                }
                else
                {
                    Item cloneForPlayerInventory = item.GetMemberwiseClone();
                    Player.Inventory.Add(cloneForPlayerInventory);

                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == item.ItemName);
                    if (itemInPlayerInventory != null)
                    {
                        itemInPlayerInventory.ItemAmount = item.BuyingCount;
                        itemInPlayerInventory.BuyingCount = 0;
                        Debug.WriteLine("Item added to player inventory was found.");
                    }
                    Debug.WriteLine("New item was added to player inventory.");
                }
                item.ItemAmount -= item.BuyingCount;
                item.BuyingCount = 0;

                //removes item from vendor if they buy out inventory
                if (item.ItemAmount <= 0)
                {
                    ToDelete.Add(item);
                    Debug.WriteLine("Removing item from vendor inventory.");
                }


            }

        }

        foreach(Item item in ToDelete)
        {
            Vendor.Inventory.Remove(item);
        }

        Vendor.ShopInteractionCount++;
        Player.PersonCurrency -= itemsTotal;
        return 1;
    }
    else
    {
        return 2;
    }

}

1 reference
```

^ Collection usage in CheckAbilityToBuyItems Method (Second Half). Collection Image 3.

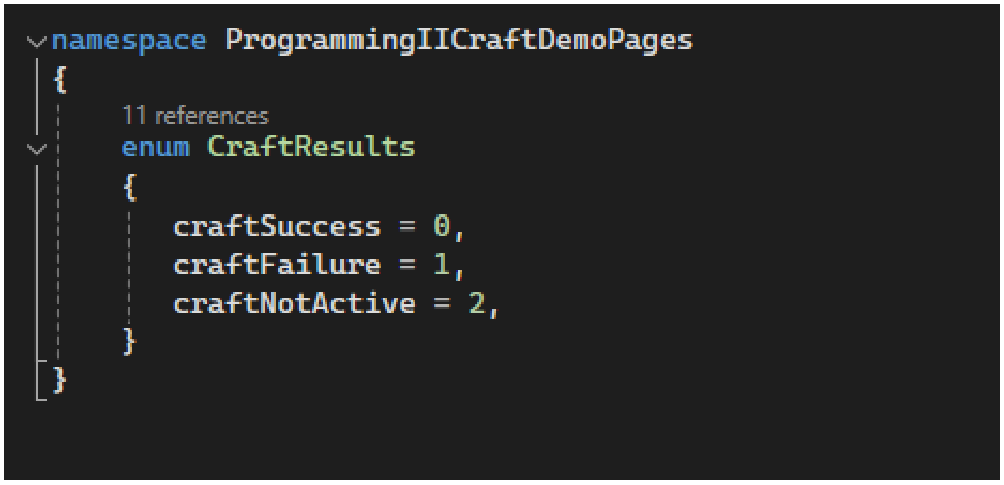
Explain usage in your project:

One example of collections in my project is how Lists are used throughout to keep track of the Vendor and Player's inventory, helping facilitate the distribution of items. In Image 1, three lists are instantiated with the correct items corresponding. The first two are for the Vendor and Player. The third list is to keep track of all possible items the vendor would need to sell at some point in order for the player to continue creating recipes. When the Vendor restocks, it goes through this list to check if it has enough stock of each item inside of it, and if it doesn't, it creates a clone for its own inventory.

Image 2 and Image 3 are examples of how these lists are used to transfer items from the vendor's inventory into the player's. By using a collection for this, all items with their own individual data can be maintained and sorted, depending on whose inventory they belong to.

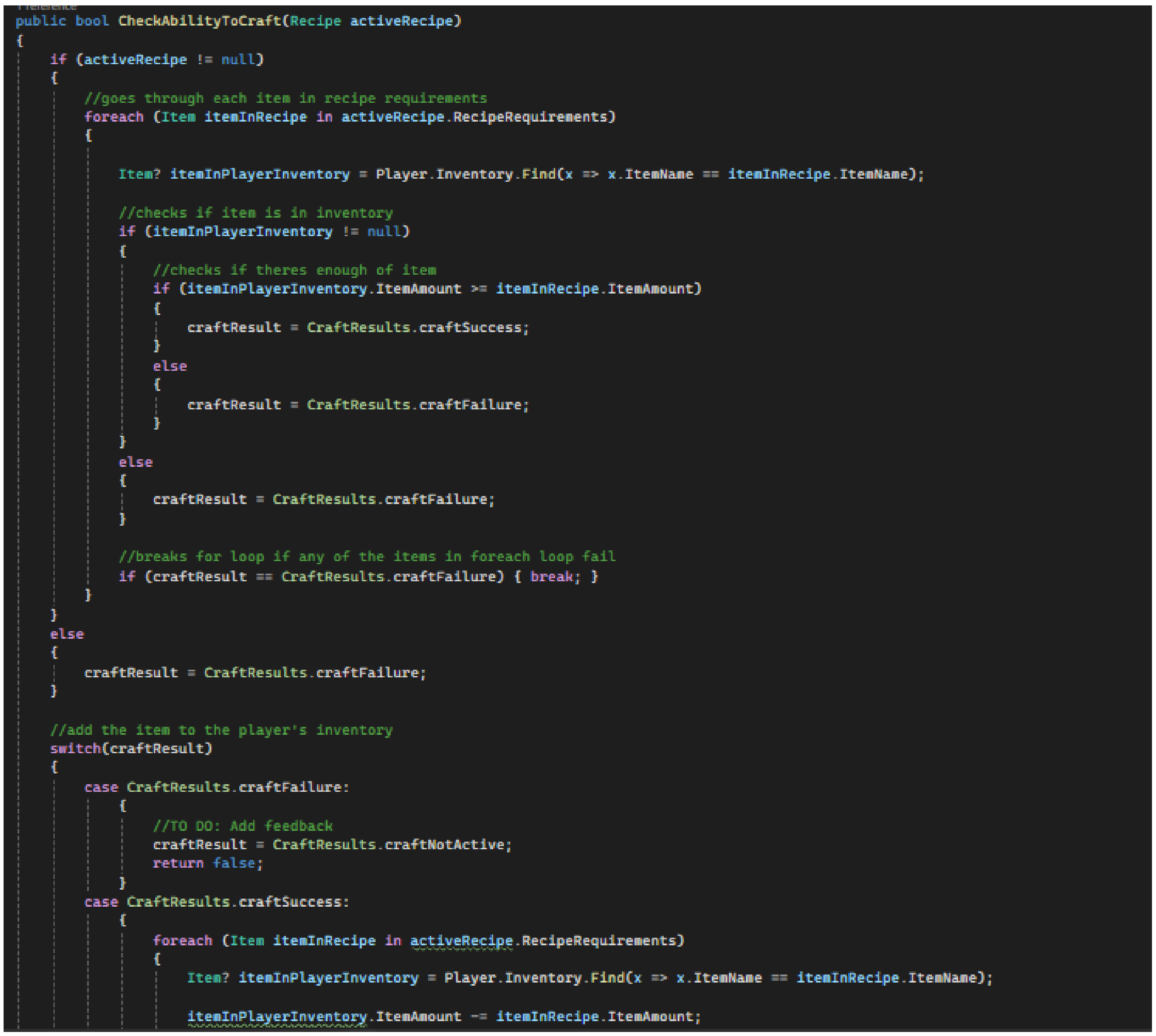
Enum

Brief code excerpt(s) from your project:



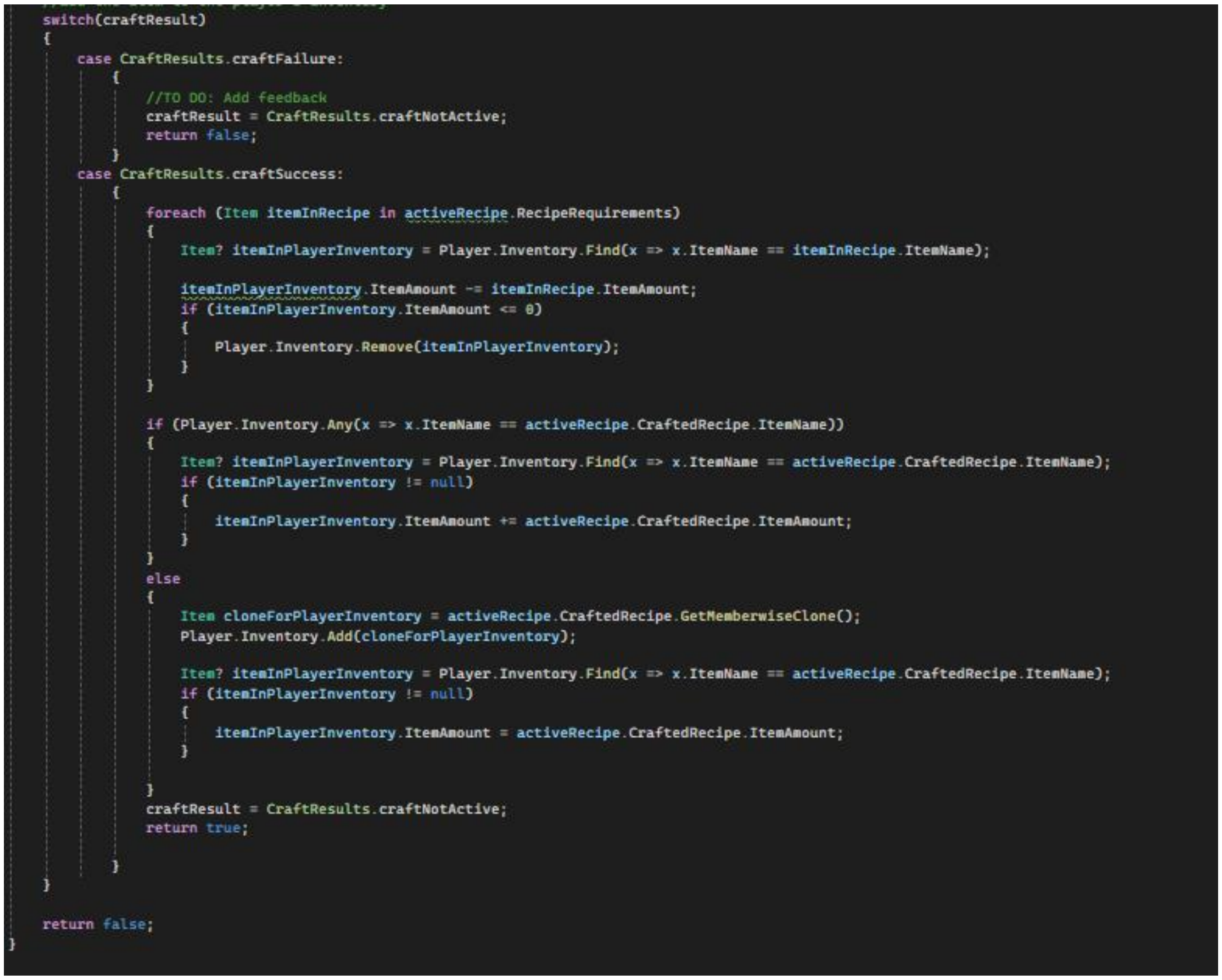
```
namespace ProgrammingIICraftDemoPages
{
    11 references
    enum CraftResults
    {
        craftSuccess = 0,
        craftFailure = 1,
        craftNotActive = 2,
    }
}
```

^ Enum used in project. Enum Image 1.



```
public bool CheckAbilityToCraft(Recipe activeRecipe)
{
    if (activeRecipe != null)
    {
        //goes through each item in recipe requirements
        foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
        {

            Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

            //checks if item is in inventory
            if (itemInPlayerInventory != null)
            {
                //checks if theres enough of item
                if (itemInPlayerInventory.ItemAmount >= itemInRecipe.ItemAmount)
                {
                    craftResult = CraftResults.craftSuccess;
                }
                else
                {
                    craftResult = CraftResults.craftFailure;
                }
            }
            else
            {
                craftResult = CraftResults.craftFailure;
            }

            //breaks for loop if any of the items in foreach loop fail
            if (craftResult == CraftResults.craftFailure) { break; }
        }
    }
    else
    {
        craftResult = CraftResults.craftFailure;
    }

    //add the item to the player's inventory
    switch(craftResult)
    {
        case CraftResults.craftFailure:
            {
                //TO DO: Add feedback
                craftResult = CraftResults.craftNotActive;
                return false;
            }
        case CraftResults.craftSuccess:
            {
                foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
                {
                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

                    itemInPlayerInventory.ItemAmount -= itemInRecipe.ItemAmount;
```

^CheckAbilityToCraft method's usage of enum (First Half). Enum Image 2.



```
    switch(craftResult)
    {
        case CraftResults.craftFailure:
            {
                //TO DO: Add feedback
                craftResult = CraftResults.craftNotActive;
                return false;
            }
        case CraftResults.craftSuccess:
            {
                foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
                {
                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

                    itemInPlayerInventory.ItemAmount -= itemInRecipe.ItemAmount;
                    if (itemInPlayerInventory.ItemAmount <= 0)
                    {
                        Player.Inventory.Remove(itemInPlayerInventory);
                    }
                }

                if (Player.Inventory.Any(x => x.ItemName == activeRecipe.CraftedRecipe.ItemName))
                {
                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == activeRecipe.CraftedRecipe.ItemName);
                    if (itemInPlayerInventory != null)
                    {
                        itemInPlayerInventory.ItemAmount += activeRecipe.CraftedRecipe.ItemAmount;
                    }
                }
                else
                {
                    Item cloneForPlayerInventory = activeRecipe.CraftedRecipe.GetMemberwiseClone();
                    Player.Inventory.Add(cloneForPlayerInventory);

                    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == activeRecipe.CraftedRecipe.ItemName);
                    if (itemInPlayerInventory != null)
                    {
                        itemInPlayerInventory.ItemAmount = activeRecipe.CraftedRecipe.ItemAmount;
                    }

                }
                craftResult = CraftResults.craftNotActive;
                return true;

            }
    }

    return false;
}
```

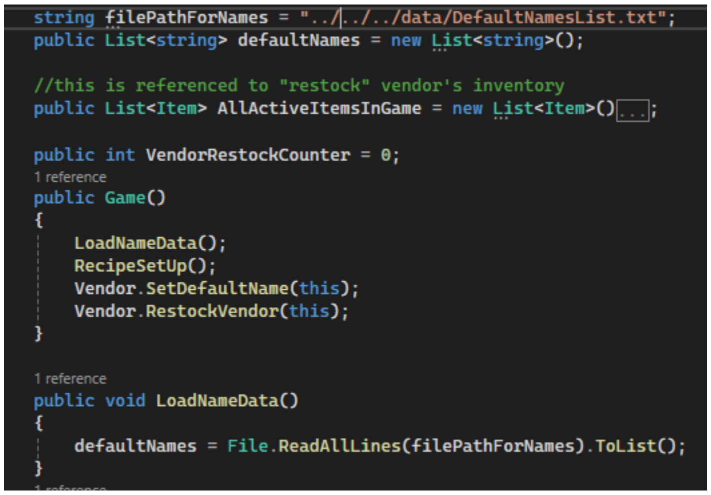
^ CheckAbilityToCraft method's usage of enum (Second Half). Enum Image 3.

Explain usage in your project:

The enum is used to change the value of craftResult depending on whether or not crafting was successful. This allows the switch later used in the script (Image 3), to reference this variable and switch cases accordingly. This makes it easier to check the success because the value of craftResults is always going to be 1 of 3 options.

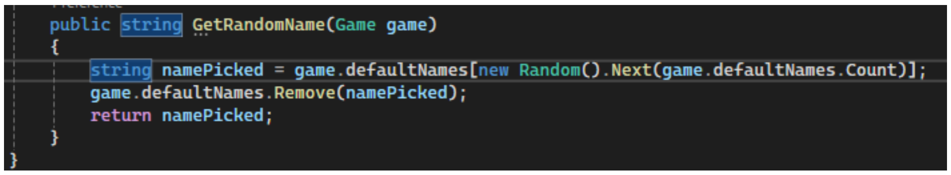
External data (read in)

Brief code excerpt(s) from your project:



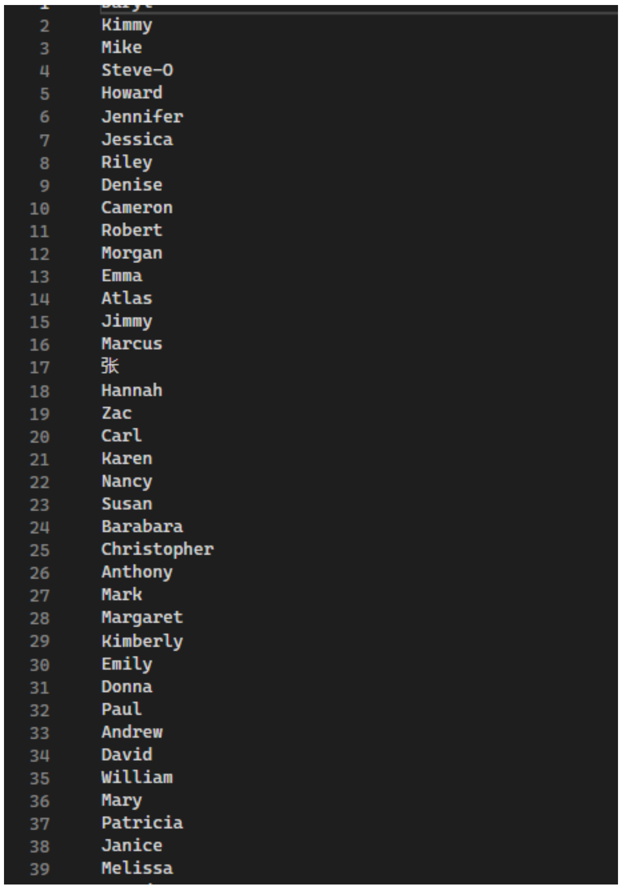
```
string filePathForNames = "../../../data/DefaultNamesList.txt";
public List<string> defaultNames = new List<string>();

//this is referenced to "restock" vendor's inventory
public List<Item> AllActiveItemsInGame = new List<Item>()...;

public int VendorRestockCounter = 0;
1 reference
public Game()
{
    LoadNameData();
    RecipeSetUp();
    Vendor.SetDefaultName(this);
    Vendor.RestockVendor(this);
}

1 reference
public void LoadNameData()
{
    defaultNames = File.ReadAllLines(filePathForNames).ToList();
}
```

^ The DefaultNamesList.txt file being saved in a variable, then loaded in inside LoadNameData(). External Data Image 1.



```
    public string GetRandomName(Game game)
    {
        string namePicked = game.defaultNames[new Random().Next(game.defaultNames.Count)];
        game.defaultNames.Remove(namePicked);
        return namePicked;
    }
}
```

^ Method that sets a default name for a person getting a random name from read in list. External Data Image 2.



```
Kimmy
Mike
Steve-O
Howard
Jennifer
Jessica
Riley
Denise
Cameron
Robert
Morgan
Emma
Atlas
Jimmy
Marcus
张
Hannah
Zac
Carl
Karen
Nancy
Susan
Barabara
Christopher
Anthony
Mark
Margaret
Kimberly
Emily
Donna
Paul
Andrew
David
William
Mary
Patricia
Janice
Melissa
```

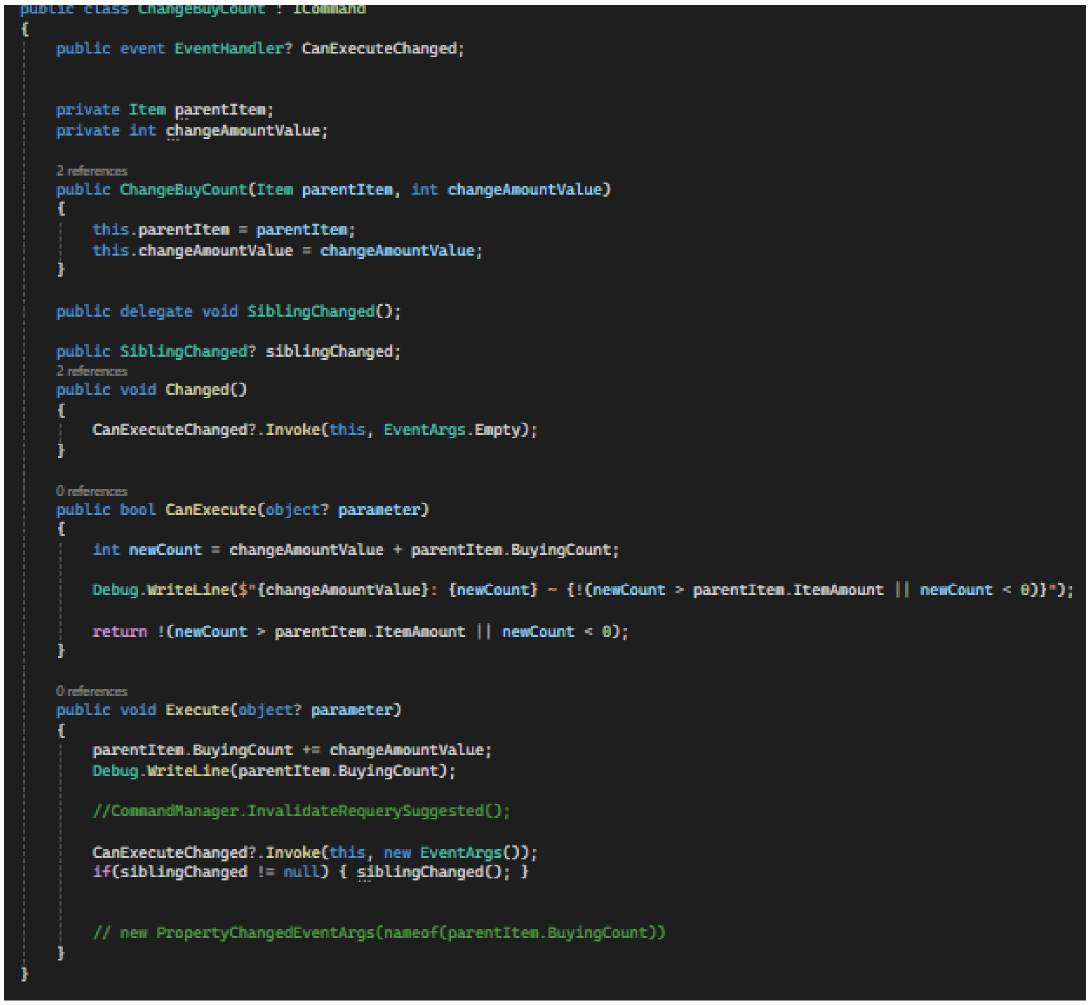
^ Text file read in to project. Image External Data 3.

Explain usage in your project:

External Data is read in for my project during set up for the list of default names (used for Customers and the Vendor). This usage makes sense because the list of default names is over 100 names long. The reason for this length is because a name is removed from this list once it is assigned so that each default person has a unique name. So, an exorbitant amount of names prevents the player from ever actually depleting the list. Therefore, it makes more sense for this string to be loaded in from a separate text file, rather than cluttering up the code with an extremely long list of strings.

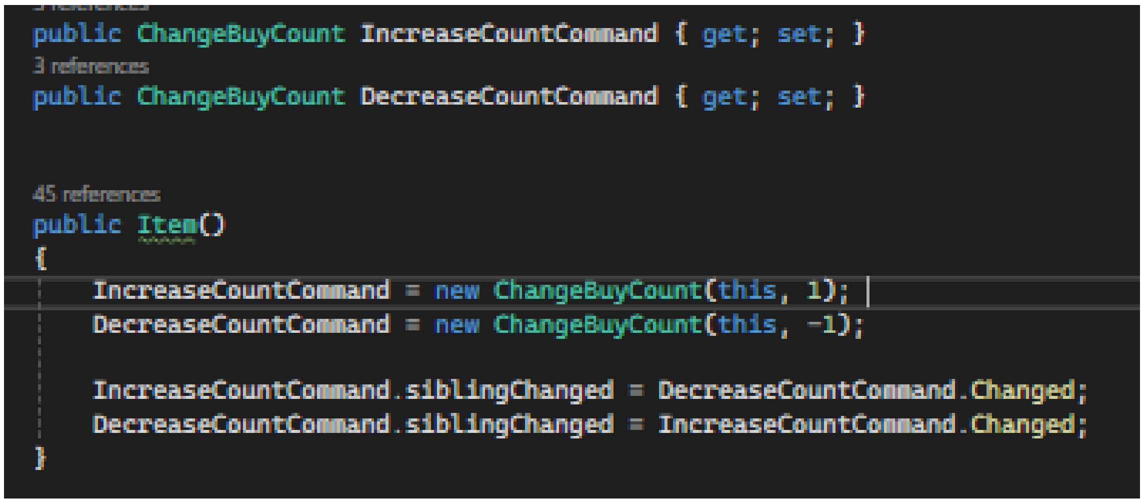
Delegate(s)

Brief code excerpt(s) from your project:



```
public class ChangeBuyCount : ICommand
{
    public event EventHandler? CanExecuteChanged;


    private Item parentItem;
    private int changeAmountValue;

    2 references
    public ChangeBuyCount(Item parentItem, int changeAmountValue)
    {
        this.parentItem = parentItem;
        this.changeAmountValue = changeAmountValue;
    }

    public delegate void SiblingChanged();

    public SiblingChanged? siblingChanged;
    2 references
    public void Changed()
    {
        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }

    0 references
    public bool CanExecute(object? parameter)
    {
        int newCount = changeAmountValue + parentItem.BuyingCount;

        Debug.WriteLine($"{changeAmountValue}: {newCount} ~ {!(newCount > parentItem.ItemAmount || newCount < 0)}");

        return !(newCount > parentItem.ItemAmount || newCount < 0);
    }

    0 references
    public void Execute(object? parameter)
    {
        parentItem.BuyingCount += changeAmountValue;
        Debug.WriteLine(parentItem.BuyingCount);

        //CommandManager.InvalidateRequerySuggested();

        CanExecuteChanged?.Invoke(this, new EventArgs());
        if(siblingChanged != null) { siblingChanged(); }


        // new PropertyChangedEventArgs(nameof(parentItem.BuyingCount))
    }
}
```

^ ChangeBuyCount class using a delegate with SiblingChanged to check if the sibling (decrement button if the clicked increase button sets the buying count to <= 1) needs to be updated. Delegate Image 1.



```
public ChangeBuyCount IncreaseCountCommand { get; set; }
3 references
public ChangeBuyCount DecreaseCountCommand { get; set; }


45 references
public Item()
{
    IncreaseCountCommand = new ChangeBuyCount(this, 1);
    DecreaseCountCommand = new ChangeBuyCount(this, -1);

    IncreaseCountCommand.siblingChanged = DecreaseCountCommand.Changed;
    DecreaseCountCommand.siblingChanged = IncreaseCountCommand.Changed;
}
```

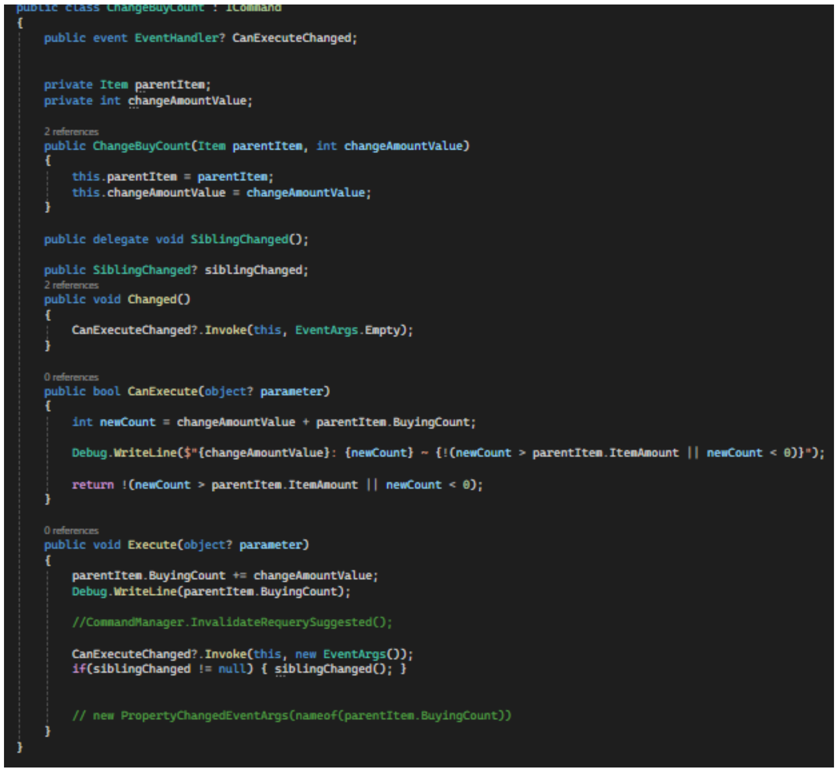
^ Item class calling delegate function to update increment and decrement buttons.

Explain usage in your project:

Delegates let you pass and store functions like they are a variable. Increment needed to be able to store decrement's update UI function, so it would be able to call it at the same time it own UI was updated (and vice versa).

Interface(s)

Brief code excerpt(s) from your project:



```
public class ChangeBuyCount : ICommand
{
    public event EventHandler? CanExecuteChanged;


    private Item parentItem;
    private int changeAmountValue;

    2 references
    public ChangeBuyCount(Item parentItem, int changeAmountValue)
    {
        this.parentItem = parentItem;
        this.changeAmountValue = changeAmountValue;
    }

    public delegate void SiblingChanged();

    public SiblingChanged? siblingChanged;
    2 references
    public void Changed()
    {
        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }

    0 references
    public bool CanExecute(object? parameter)
    {
        int newCount = changeAmountValue + parentItem.BuyingCount;

        Debug.WriteLine($"{changeAmountValue}: {newCount} ~ {!(newCount > parentItem.ItemAmount || newCount < 0)}");

        return !(newCount > parentItem.ItemAmount || newCount < 0);
    }

    0 references
    public void Execute(object? parameter)
    {
        parentItem.BuyingCount += changeAmountValue;
        Debug.WriteLine(parentItem.BuyingCount);

        //CommandManager.InvalidateRequerySuggested();

        CanExecuteChanged?.Invoke(this, new EventArgs());
        if(siblingChanged != null) { siblingChanged(); }


        // new PropertyChangedEventArgs(nameof(parentItem.BuyingCount))
    }
}
```

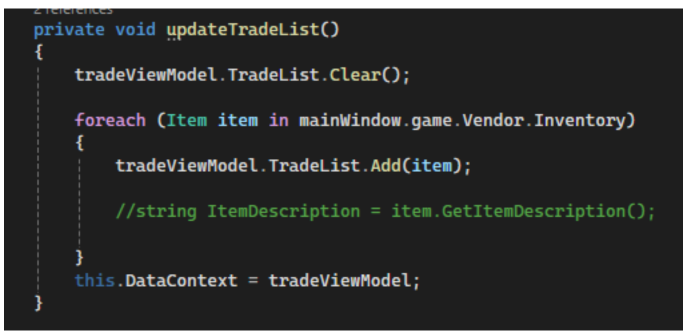
^ Class which implements ICommand interface. Interface Image 1.

Explain usage in your project:

The interface I used in my project is the ICommand, which is a built in interface in WPF. This is helpful for the increment and decrement buttons because of the usage of the CanExecute function and CanExecuteChanged. This way, the buttons can be updated accordingly (for example, if the buying count gets to the exact amount the vendor has in stock, the increment button should be disabled). By implementing an interface for these buttons, as opposed to using button click, the buttons use more complex logic, through the required parameters.

One of these: LINQ or XML

Brief code excerpt(s) from your project:



```
2 references
private void updateTradeList()
{
    tradeViewModel.TradeList.Clear();

    foreach (Item item in mainWindow.game.Vendor.Inventory)
    {
        tradeViewModel.TradeList.Add(item);

        //string ItemDescription = item.GetItemDescription();

    }
    this.DataContext = tradeViewModel;
}
```

^ LINQ Image 1.

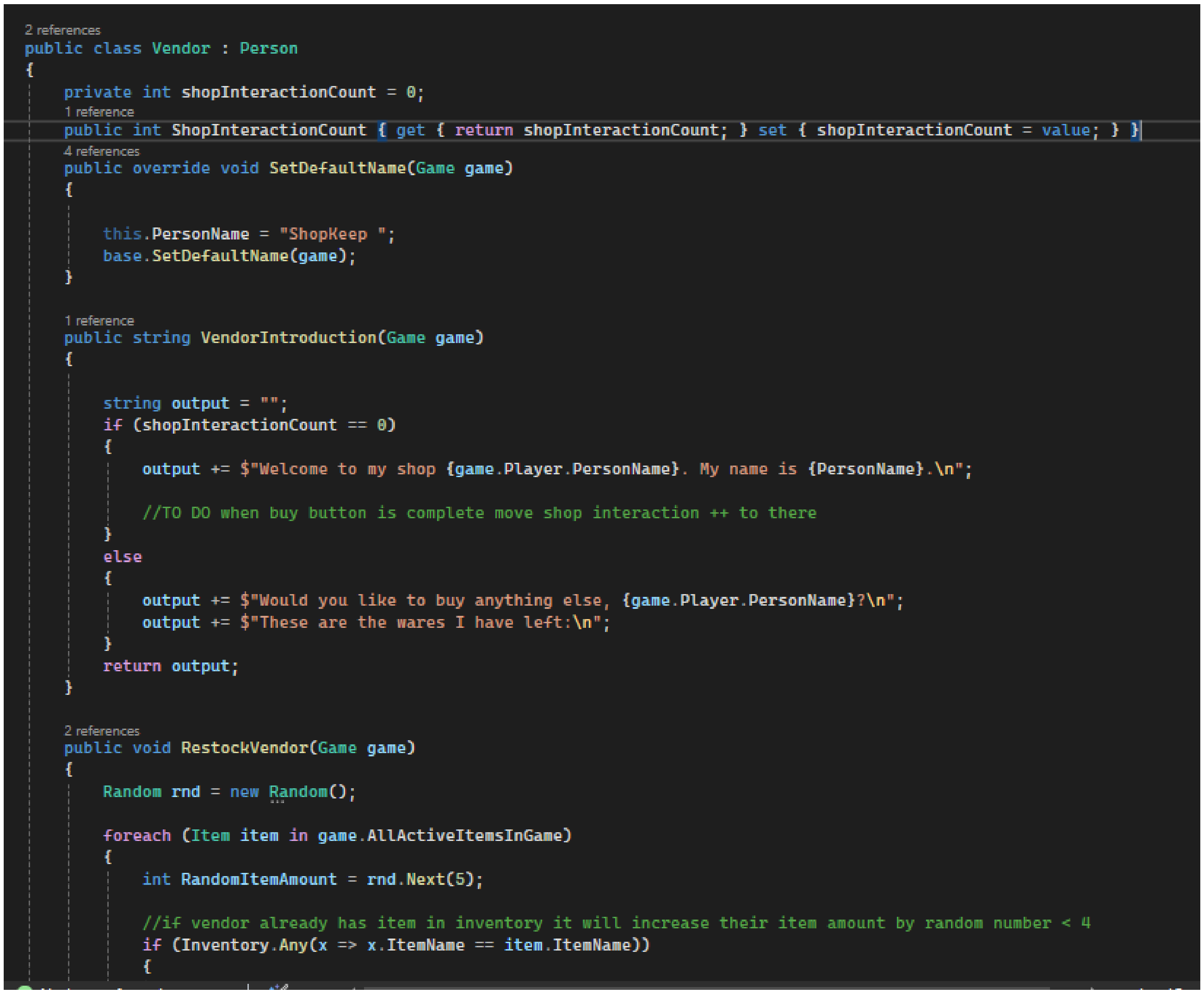
Explain usage in your project:

The UpdateTradeList function is using a query expression to then cycle through the data in the vendor's inventory. This is done whenever a change is made to adjust the view accordingly.

Required Functionality

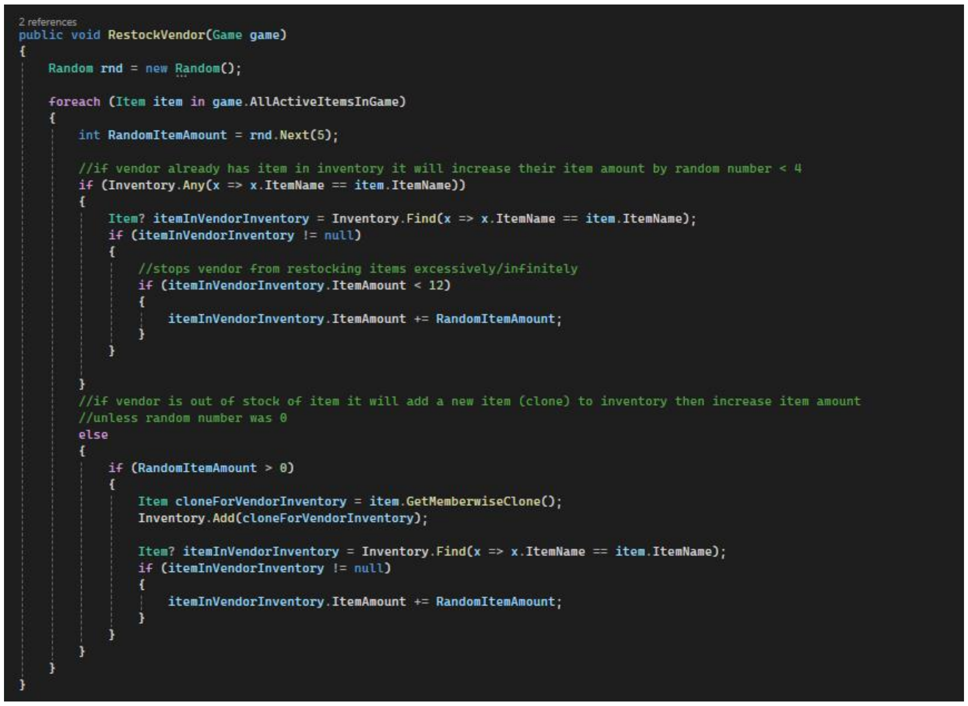
Supplier

Brief code excerpt(s) from your project:



```
2 references
public class Vendor : Person
{
    private int shopInteractionCount = 0;
    1 reference
    public int ShopInteractionCount { get { return shopInteractionCount; } set { shopInteractionCount = value; } }
    4 references
    public override void SetDefaultName(Game game)
    {

        this.PersonName = "ShopKeep ";
        base.SetDefaultName(game);
    }

    1 reference
    public string VendorIntroduction(Game game)
    {

        string output = "";
        if (shopInteractionCount == 0)
        {
            output += $"Welcome to my shop {game.Player.PersonName}. My name is {PersonName}.\n";

            //TO DO when buy button is complete move shop interaction ++ to there
        }
        else
        {
            output += $"Would you like to buy anything else, {game.Player.PersonName}?\n";
            output += $"These are the wares I have left:\n";
        }
        return output;
    }

    2 references
    public void RestockVendor(Game game)
    {
        Random rnd = new Random();

        foreach (Item item in game.AllActiveItemsInGame)
        {
            int RandomItemAmount = rnd.Next(5);

            //if vendor already has item in inventory it will increase their item amount by random number < 4
            if (Inventory.Any(x => x.ItemName == item.ItemName))
            {
```

^ Vendor Class (First Half). Supplier Image 1.



```
2 references
public void RestockVendor(Game game)
{
    Random rnd = new Random();

    foreach (Item item in game.AllActiveItemsInGame)
    {
        int RandomItemAmount = rnd.Next(5);

        //if vendor already has item in inventory it will increase their item amount by random number < 4
        if (Inventory.Any(x => x.ItemName == item.ItemName))
        {
            Item? itemInVendorInventory = Inventory.Find(x => x.ItemName == item.ItemName);
            if (itemInVendorInventory != null)
            {
                //stops vendor from restocking items excessively/infinitely
                if (itemInVendorInventory.ItemAmount < 12)
                {
                    itemInVendorInventory.ItemAmount += RandomItemAmount;
                }
            }

        }
        //if vendor is out of stock of item it will add a new item (clone) to inventory then increase item amount
        //unless random number was 0
        else
        {
            if (RandomItemAmount > 0)
            {
                Item cloneForVendorInventory = item.GetMemberwiseClone();
                Inventory.Add(cloneForVendorInventory);

                Item? itemInVendorInventory = Inventory.Find(x => x.ItemName == item.ItemName);
                if (itemInVendorInventory != null)
                {
                    itemInVendorInventory.ItemAmount += RandomItemAmount;
                }
            }
        }
    }
}
```

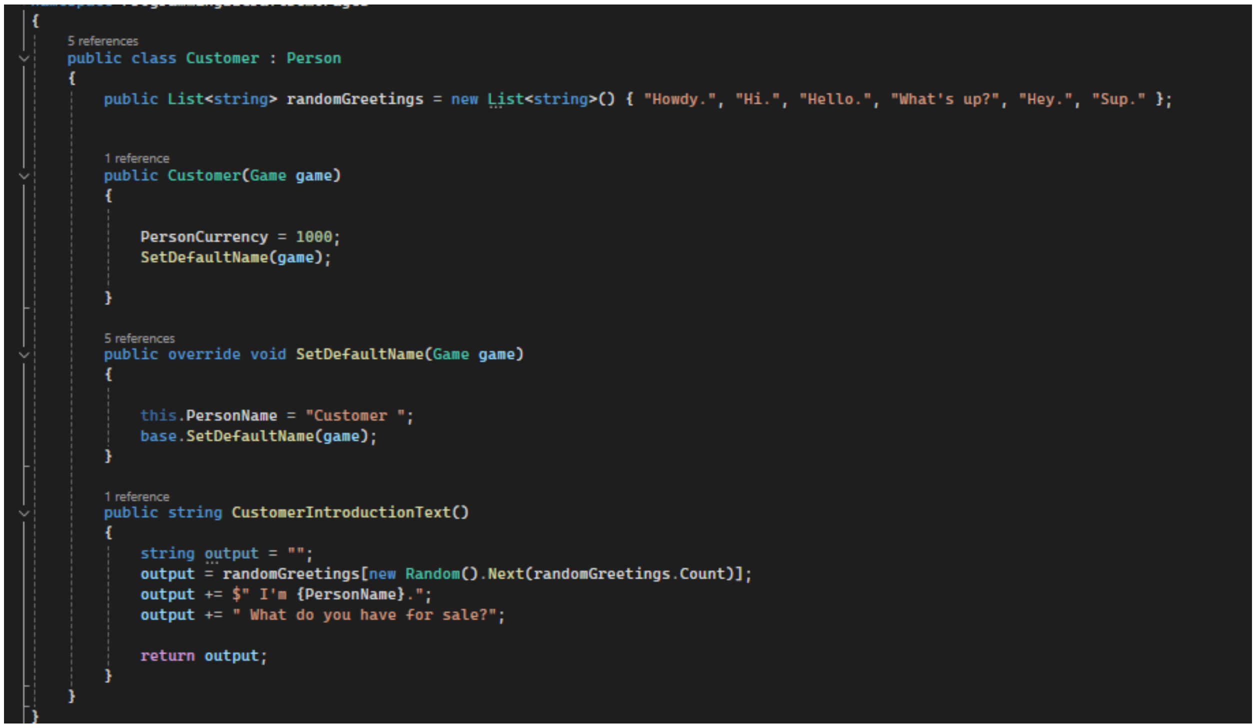
^ Vendor Class (Second Half). Supplier Image 2.

Explain usage in your project:

The vendor class is used to handle the supplier and is a child of person. The vendor introduction generates a string to talk to the player, asking them if they would like to buy ingredients. This function is called in the XAML that creates the display for the Buy Window. This class also contains the code to restock the vendor's inventory, which is to be called periodically when the player buys enough items. This character's inventory facilitates the items that the player can buy, including those items' prices.

Customer

Brief code excerpt(s) from your project:



```
{
    public class Customer : Person
    {
        public List<string> randomGreetings = new List<string>() { "Howdy.", "Hi.", "Hello.", "What's up?", "Hey.", "Sup." };


        public Customer(Game game)
        {

            PersonCurrency = 1000;
            SetDefaultName(game);

        }

        public override void SetDefaultName(Game game)
        {

            this.PersonName = "Customer ";
            base.SetDefaultName(game);
        }

        public string CustomerIntroductionText()
        {
            string output = "";
            output = randomGreetings[new Random().Next(randomGreetings.Count)];
            output += $" I'm {PersonName}.";
            output += " What do you have for sale?";

            return output;
        }
    }
}
```

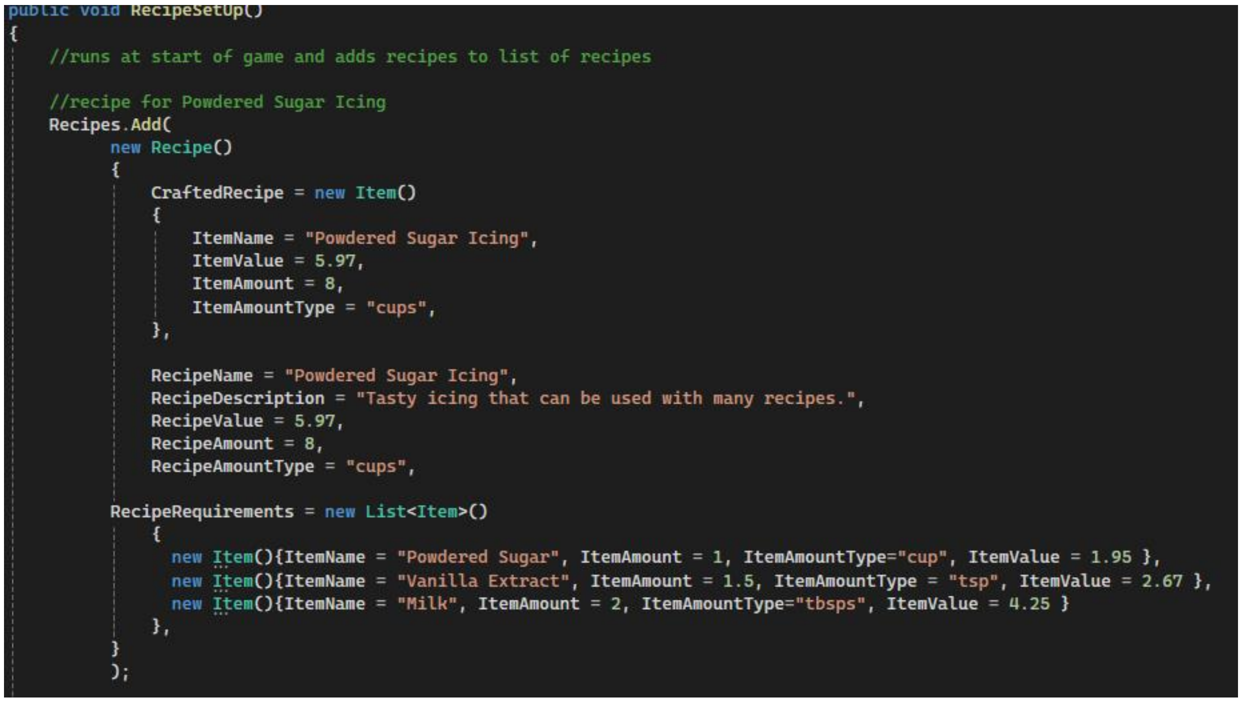
^ Customer class, child of Person. Customer Image 1.

Explain usage in your project:

This class is a child of person and is used to generate a new customer each time the player opens the Sell Window. By creating this class it automatically sets the funds of the customer, allows an override function for the naming of the customer, and handles the randomly generated introduction of the customer, which is printed to the XAML.

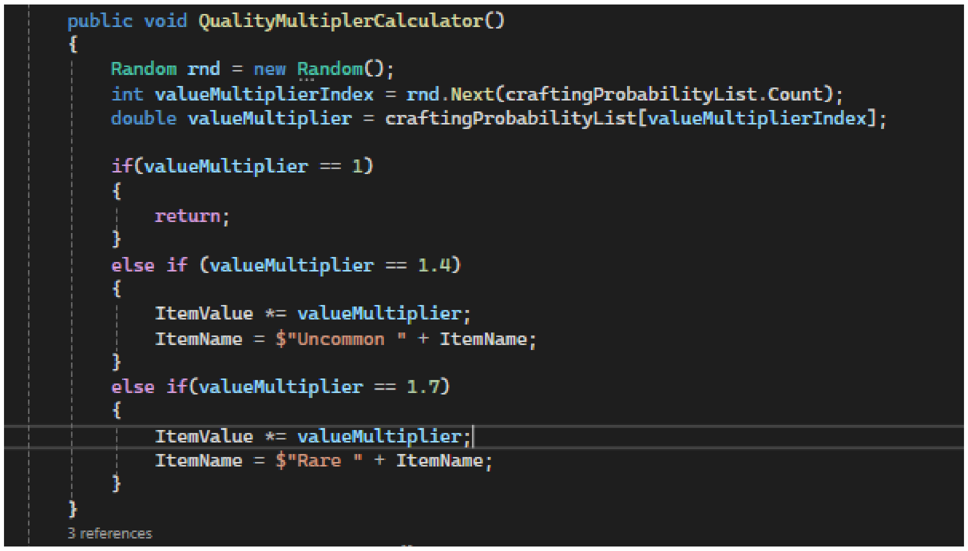
Profit Margin

Brief code excerpt(s) from your project:



```
public void RecipeSetUp()
{
    //runs at start of game and adds recipes to list of recipes

    //recipe for Powdered Sugar Icing
    Recipes.Add(
          new Recipe()
          {
              CraftedRecipe = new Item()
              {
                  ItemName = "Powdered Sugar Icing",
                  ItemValue = 5.97,
                  ItemAmount = 8,
                  ItemAmountType = "cups",
              },

              RecipeName = "Powdered Sugar Icing",
              RecipeDescription = "Tasty icing that can be used with many recipes.",
              RecipeValue = 5.97,
              RecipeAmount = 8,
              RecipeAmountType = "cups",

          RecipeRequirements = new List<Item>()
              {
                new Item(){ItemName = "Powdered Sugar", ItemAmount = 1, ItemAmountType="cup", ItemValue = 1.95 },
                new Item(){ItemName = "Vanilla Extract", ItemAmount = 1.5, ItemAmountType = "tsp", ItemValue = 2.67 },
                new Item(){ItemName = "Milk", ItemAmount = 2, ItemAmountType="tbsps", ItemValue = 4.25 }
              },
          }
          );
```

^ Recipe for Powdered Sugar Icing showing ingredient values and resulting item value. Profit Margin Image 1.



```
public void QualityMultiplerCalculator()
{
    Random rnd = new Random();
    int valueMultiplierIndex = rnd.Next(craftingProbabilityList.Count);
    double valueMultiplier = craftingProbabilityList[valueMultiplierIndex];

    if(valueMultiplier == 1)
    {
        return;
    }
    else if (valueMultiplier == 1.4)
    {
        ItemValue *= valueMultiplier;
        ItemName = $"Uncommon " + ItemName;
    }
    else if(valueMultiplier == 1.7)
    {
        ItemValue *= valueMultiplier;
        ItemName = $"Rare " + ItemName;
    }
}
3 references
```

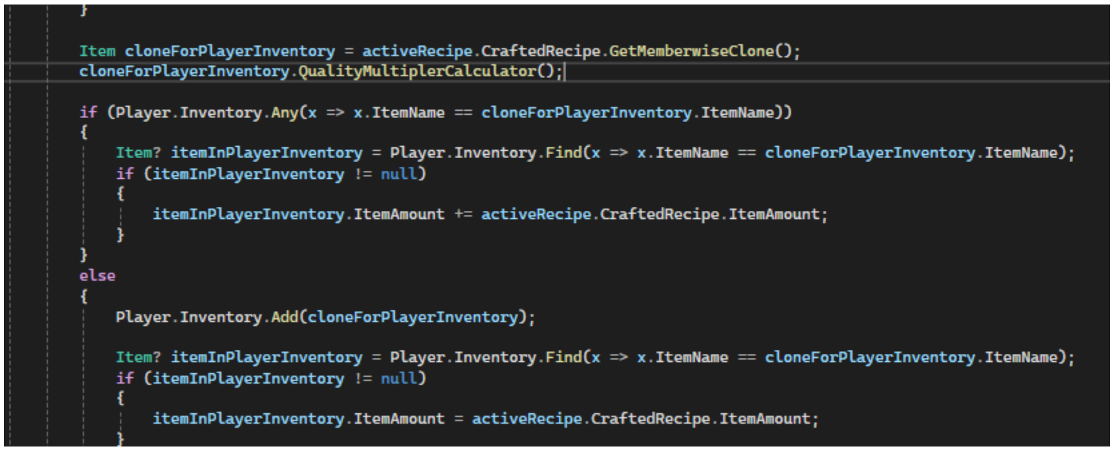
^ The function on the item class that adds the chance of Rare or Uncommon Quality. Profit Margin 2.

Explain usage in your project:

Each recipe's finished product is worth more than the combined value of all the ingredients. Additionally, when an item is crafted it has a chance to be Uncommon or Rare, further increasing the sell value of the item.

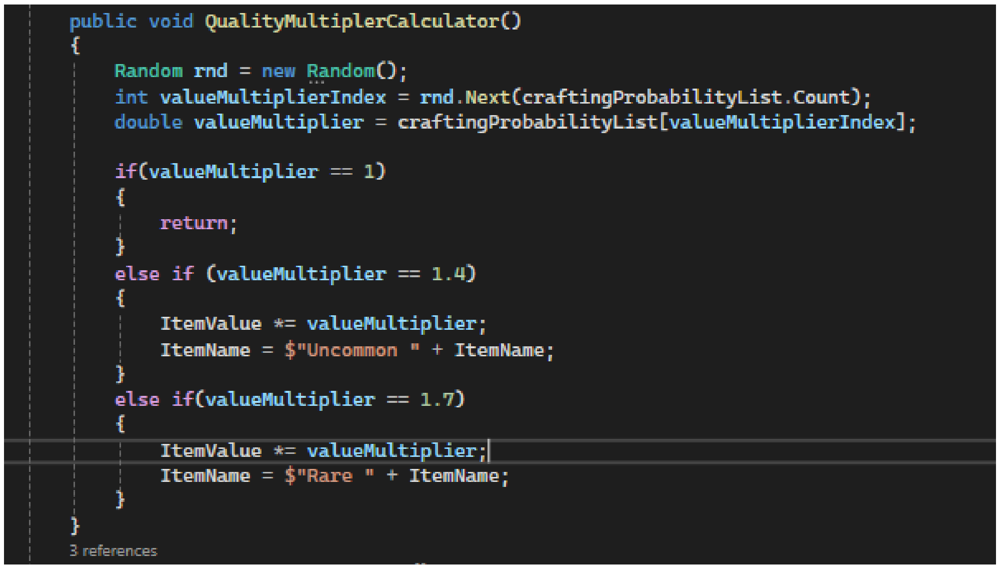
Probability

Brief code excerpt(s) from your project:



```
}

Item cloneForPlayerInventory = activeRecipe.CraftedRecipe.GetMemberwiseClone();
cloneForPlayerInventory.QualityMultiplerCalculator();

if (Player.Inventory.Any(x => x.ItemName == cloneForPlayerInventory.ItemName))
{
    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == cloneForPlayerInventory.ItemName);
    if (itemInPlayerInventory != null)
    {
        itemInPlayerInventory.ItemAmount += activeRecipe.CraftedRecipe.ItemAmount;
    }
}
else
{
    Player.Inventory.Add(cloneForPlayerInventory);

    Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == cloneForPlayerInventory.ItemName);
    if (itemInPlayerInventory != null)
    {
        itemInPlayerInventory.ItemAmount = activeRecipe.CraftedRecipe.ItemAmount;
    }
```

^ Code that runs the function to randomly (10% chance of Rare, 20% chance of Uncommon, 70% chance of regular item) add a multiplier to the item's quality. Then, checks to see if the player has one of exact quality or whether a new item needs to be added. Probability Image 1.



```
public void QualityMultiplerCalculator()
{
    Random rnd = new Random();
    int valueMultiplierIndex = rnd.Next(craftingProbabilityList.Count);
    double valueMultiplier = craftingProbabilityList[valueMultiplierIndex];

    if(valueMultiplier == 1)
    {
        return;
    }
    else if (valueMultiplier == 1.4)
    {
        ItemValue *= valueMultiplier;
        ItemName = $"Uncommon " + ItemName;
    }
    else if(valueMultiplier == 1.7)
    {
        ItemValue *= valueMultiplier;
        ItemName = $"Rare " + ItemName;
    }
}
3 references
```

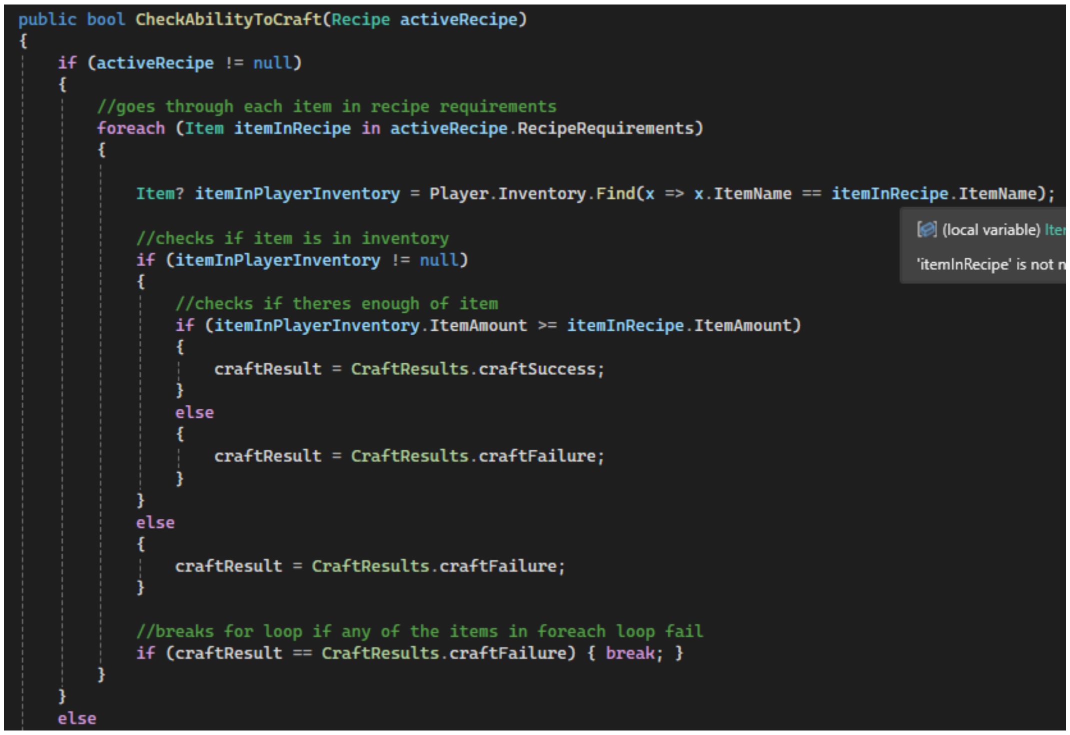
^ Method inside Item class that is used to add a random quality multiplier and name change to the item, called after it has been crafted.

Explain usage in your project:

When the player crafts an item the method QualityMultiplierCalculator() pictured in image 2 is used to get a quality multiplier for the item. The craftingProbabilityList contains doubles 1(x7), 1.4(x2), and 1.7. Therefore, when random is called on this list there is a 70% chance that the item will be regular quality, a 20% chance that the item will be of Uncommon quality, and a 10% chance that the item will be rare. Then the item's value will be multiplied by the double pulled randomly from the list. This function is called right after the item is crafted in Image 1.

Generalized Craft Algorithm

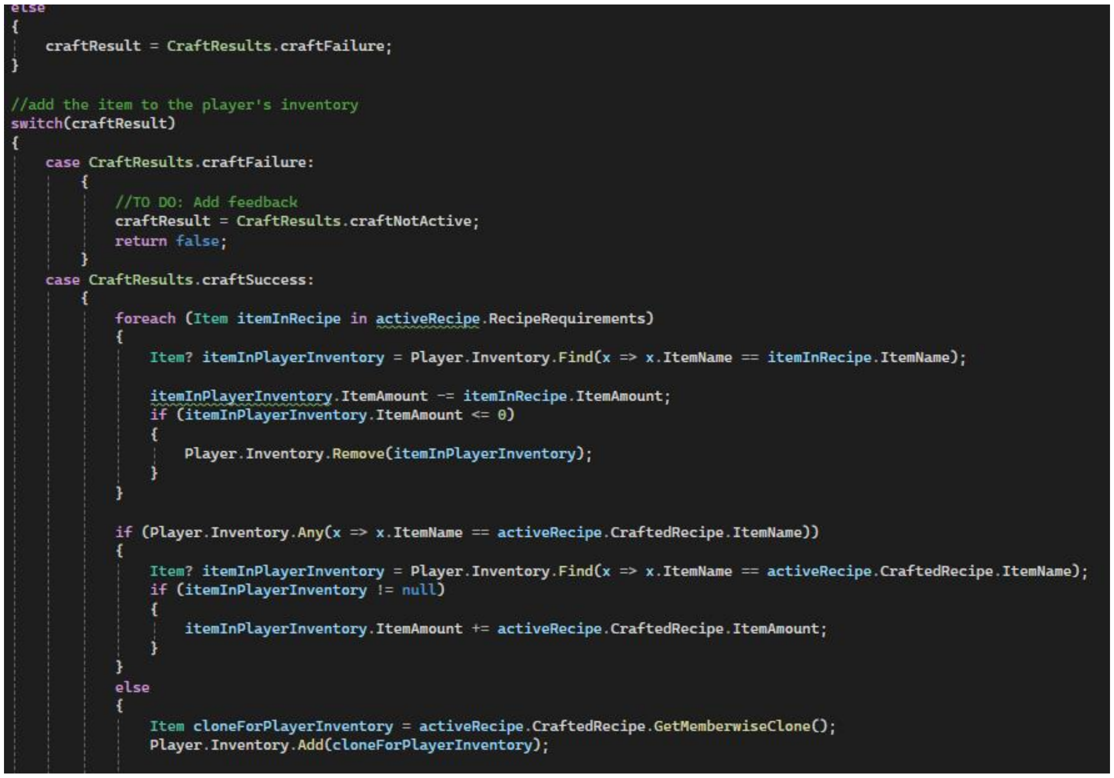
Brief code excerpt(s) from your project:



```
public bool CheckAbilityToCraft(Recipe activeRecipe)
{
    if (activeRecipe != null)
    {
        //goes through each item in recipe requirements
        foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
        {

            Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

            //checks if item is in inventory
            if (itemInPlayerInventory != null)
            {
                //checks if theres enough of item
                if (itemInPlayerInventory.ItemAmount >= itemInRecipe.ItemAmount)
                {
                    craftResult = CraftResults.craftSuccess;
                }
                else
                {
                    craftResult = CraftResults.craftFailure;
                }
            }
            else
            {
                craftResult = CraftResults.craftFailure;
            }

            //breaks for loop if any of the items in foreach loop fail
            if (craftResult == CraftResults.craftFailure) { break; }
        }
    }
    else
```

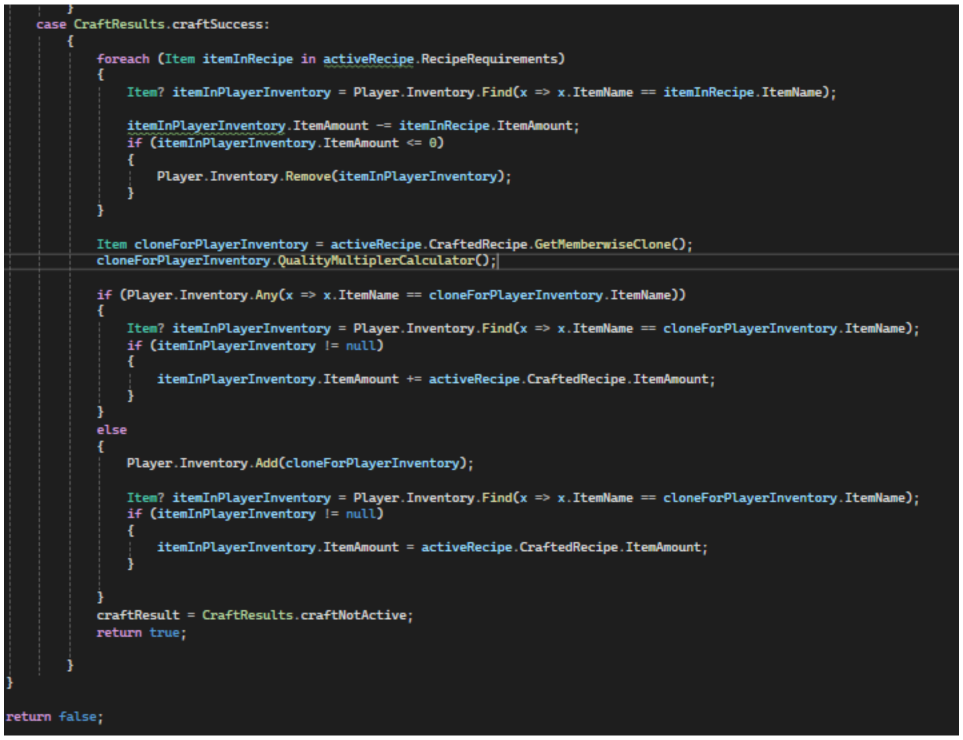
```
(local variable) Ite
'itemInRecipe' is not n
```

^ algorithm passes in whatever the player selected in the window on the Craft Page, then goes through each ingredient to check if the player has the correct quantity in their inventory, if they do the craftResult is enum is set to craftSuccess. If they don't the enum is set to craftFailure. Generalized Craft Algorithm Image 1.



```
else
{
    craftResult = CraftResults.craftFailure;
}

//add the item to the player's inventory
switch(craftResult)
{
    case CraftResults.craftFailure:
        {
            //TO DO: Add feedback
            craftResult = CraftResults.craftNotActive;
            return false;
        }
    case CraftResults.craftSuccess:
        {
            foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
            {
                Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

                itemInPlayerInventory.ItemAmount -= itemInRecipe.ItemAmount;
                if (itemInPlayerInventory.ItemAmount <= 0)
                {
                    Player.Inventory.Remove(itemInPlayerInventory);
                }
            }

            if (Player.Inventory.Any(x => x.ItemName == activeRecipe.CraftedRecipe.ItemName))
            {
                Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == activeRecipe.CraftedRecipe.ItemName);
                if (itemInPlayerInventory != null)
                {
                    itemInPlayerInventory.ItemAmount += activeRecipe.CraftedRecipe.ItemAmount;
                }
            }
            else
            {
                Item cloneForPlayerInventory = activeRecipe.CraftedRecipe.GetMemberwiseClone();
                Player.Inventory.Add(cloneForPlayerInventory);
```

^ Continuation of craft algorithm. Based off the setting of the enum, the switch case either results in a failure to craft the item or adds the item to the player's inventory. Generalized Crafting Algorithm Image 2.



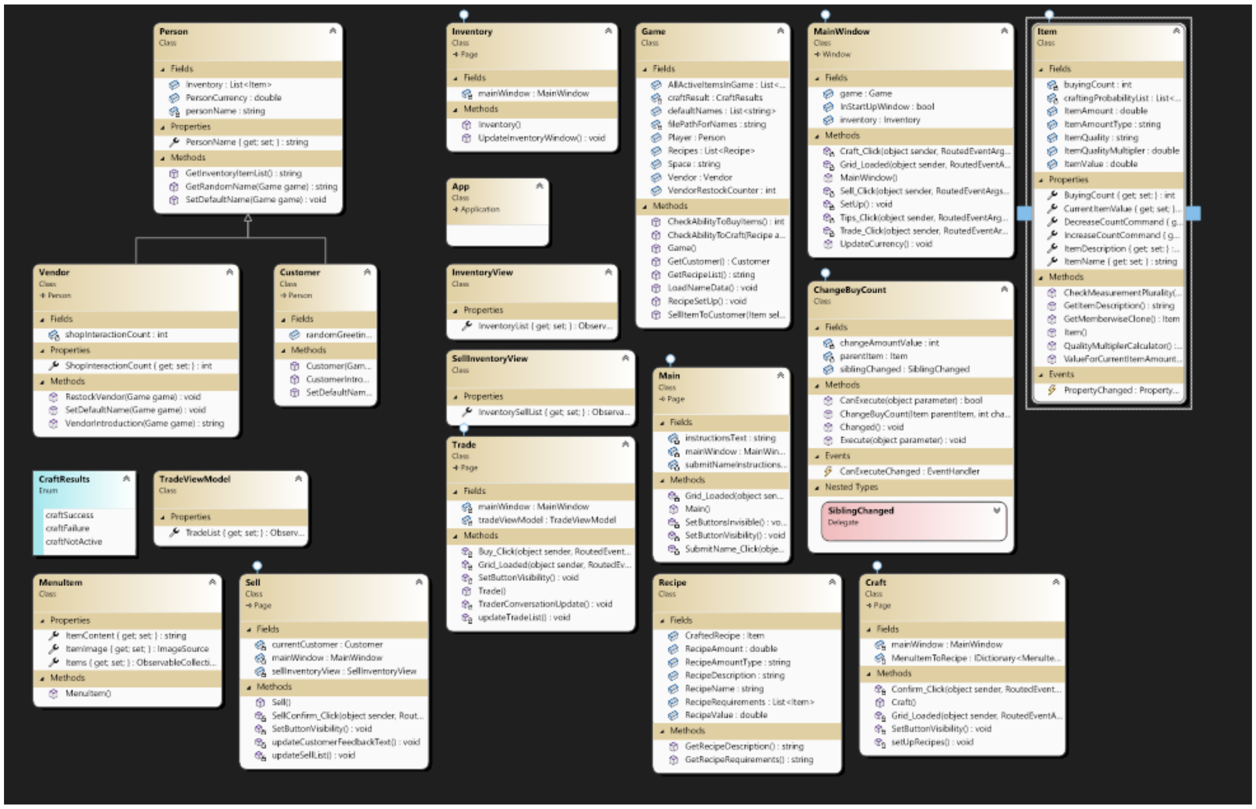
```
        }
    case CraftResults.craftSuccess:
        {
            foreach (Item itemInRecipe in activeRecipe.RecipeRequirements)
            {
                Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == itemInRecipe.ItemName);

                itemInPlayerInventory.ItemAmount -= itemInRecipe.ItemAmount;
                if (itemInPlayerInventory.ItemAmount <= 0)
                {
                    Player.Inventory.Remove(itemInPlayerInventory);
                }
            }

            Item cloneForPlayerInventory = activeRecipe.CraftedRecipe.GetMemberwiseClone();
            cloneForPlayerInventory.QualityMultiplerCalculator();

            if (Player.Inventory.Any(x => x.ItemName == cloneForPlayerInventory.ItemName))
            {
                Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == cloneForPlayerInventory.ItemName);
                if (itemInPlayerInventory != null)
                {
                    itemInPlayerInventory.ItemAmount += activeRecipe.CraftedRecipe.ItemAmount;
                }
            }
            else
            {
                Player.Inventory.Add(cloneForPlayerInventory);

                Item? itemInPlayerInventory = Player.Inventory.Find(x => x.ItemName == cloneForPlayerInventory.ItemName);
                if (itemInPlayerInventory != null)
                {
                    itemInPlayerInventory.ItemAmount = activeRecipe.CraftedRecipe.ItemAmount;
                }

            }
            craftResult = CraftResults.craftNotActive;
            return true;

        }
}

return false;
```

^ This continuation of the crafting algorithm calls the multiplier function on the item to set a quality and corresponding value for it. Then, checks to see if it should combine the item count with a preexisting item in the player's inventory or if it should add a new clone item to the player's inventory. Generalized Crafting Algorithm Image 3.

Explain usage in your project:

This method passes in whatever recipe the player has selected from the menu, therefore the variable can be used to abstractly craft. Every recipe will have ingredients on it and a finished item corresponding to the recipe (to clone and add to the player's inventory). Therefore, by calling the variables on the items, instead of specific values, this code can be reused for every single recipe. This is the only code used to craft items.

UML Diagrams


Person
Class
Fields
Inventory : List<Item>
PersonCurrency : double
personName : string
Properties
PersonName { get; set; } : string
Methods
GetInventoryItemList() : string
GetRandomName(Game game) : string
SetDefaultName(Game game) : void
Vendor
Class
Person
Fields
shopInteractionCount : int
Properties
ShopInteractionCount { get; set; } : int
Methods
RestockVendor(Game game) : void
SetDefaultName(Game game) : void
VendorIntroduction(Game game) : string
Customer
Class
Person
Fields
randomGreetin...
Methods
Customer(Gam...
CustomerIntro...
SetDefaultNam...
CraftResults
Enum
craftSuccess
craftFailure
craftNotActive
TradeViewModel
Class
Properties
TradeList { get; set; } : Observ...
MenuItem
Class
Properties
ItemContent { get; set; } : string
ItemImage { get; set; } : ImageSource
Items { get; set; } : ObservableCollecti...
Methods
MenuItem()
Sell
Class
Page
Fields
currentCustomer : Customer
mainWindow : MainWindow
sellInventoryView : SellInventoryView
Methods
Sell()
SellConfirm_Click(object sender, Rout...
SetButtonVisibility() : void
updateCustomerFeedbackText() : void
updateSellList() : void
Inventory
Class
Page
Fields
mainWindow : MainWindow
Methods
Inventory()
UpdateInventoryWindow() : void
App
Class
Application
InventoryView
Class
Properties
InventoryList { get; set; } : Observ...
SellInventoryView
Class
Properties
InventorySellList { get; set; } : Observa...
Trade
Class
Page
Fields
mainWindow : MainWindow
tradeViewModel : TradeViewModel
Methods
Buy_Click(object sender, RoutedEvent...
Grid_Loaded(object sender, RoutedEv...
SetButtonVisibility() : void
Trade()
TraderConversationUpdate() : void
updateTradeList() : void
Game
Class
Fields
AllActiveItemsInGame : List<...
craftResult : CraftResults
defaultNames : List<string>
filePathForNames : string
Player : Person
Recipes : List<Recipe>
Space : string
Vendor : Vendor
VendorRestockCounter : int
Methods
CheckAbilityToBuyItems() : int
CheckAbilityToCraft(Recipe a...
Game()
GetCustomer() : Customer
GetRecipeList() : string
LoadNameData() : void
RecipeSetUp() : void
SellItemToCustomer(Item sel...
Main
Class
Page
Fields
instructionsText : string
mainWindow : MainWin...
submitNameInstructions...
Methods
Grid_Loaded(object sen...
Main()
SetButtonsInvisible() : vo...
SetButtonVisibility() : void
SubmitName_Click(obje...
Recipe
Class
Fields
CraftedRecipe : Item
RecipeAmount : double
RecipeAmountType : string
RecipeDescription : string
RecipeName : string
RecipeRequirements : List<Item>
RecipeValue : double
Methods
GetRecipeDescription() : string
GetRecipeRequirements() : string
MainWindow
Class
Window
Fields
game : Game
InStartUpWindow : bool
inventory : Inventory
Methods
Craft_Click(object sender, RoutedEventArg...
Grid_Loaded(object sender, RoutedEventA...
MainWindow()
Sell_Click(object sender, RoutedEventArgs...
SetUp() : void
Tips_Click(object sender, RoutedEventArg...
Trade_Click(object sender, RoutedEventAr...
UpdateCurrency() : void
ChangeBuyCount
Class
Fields
changeAmountValue : int
parentItem : Item
siblingChanged : SiblingChanged
Methods
CanExecute(object parameter) : bool
ChangeBuyCount(Item parentItem, int cha...
Changed() : void
Execute(object parameter) : void
Events
CanExecuteChanged : EventHandler
Nested Types
SiblingChanged
Delegate
Craft
Class
Page
Fields
mainWindow : MainWindow
MenuItemToRecipe : IDictionary<MenuIte...
Methods
Confirm_Click(object sender, RoutedEvent...
Craft()
Grid_Loaded(object sender, RoutedEventA...
SetButtonVisibility() : void
setUpRecipes() : void
Item
Class
Fields
buyingCount : int
craftingProbabilityList : List<...
ItemAmount : double
ItemAmountType : string
ItemQuality : string
ItemQualityMultipler : double
ItemValue : double
Properties
BuyingCount { get; set; } : int
CurrentItemValue { get; set; }...
DecreaseCountCommand { g...
IncreaseCountCommand { g...
ItemDescription { get; set; } :...
ItemName { get; set; } : string
Methods
CheckMeasurementPlurality(...
GetItemDescription() : string
GetMemberwiseClone() : Item
Item()
QualityMultiplerCalculator() :...
ValueForCurrentItemAmount...
Events
PropertyChanged : Property...

UML Diagram Explanation

The relationship of inheritance used in this project is the parent Person class to the children Vendor and Customer classes. By doing this, the code for the person can be reused for all three, because all three require these more generalized components. Also, the Vendor and Customer can override virtual methods inside the Person class in order to add whatever corresponding unique components they need. In this case, the virtual method used is the SetDefaultName() method, which for customers is overridden to Customer plus their name (from the base function) and for the Vendor is overridden to Vendor plus their name (from the base function).

All of the code inside my XAML is used to handle what pages and buttons are visible, and what text is printed to the screen. This way, all of the code in these pages address only one concern, the visuals printed to the screen for the player. The Game class handles the functionality and logic of the game. The XAML calls methods within the game class, which change data based on the players selection in the window. Then, send that data back, for example, returning a string, for the XAML CS to print to the screen. Other classes like Item and Person, each have their own separate methods that game calls upon accordingly or store data for algorithms like the Buy and Craft functionality in game (Person sets up the variables each person contains, Item handles the functionality of the incrementation/decrementation of what the item the player is buying).

Playtesting

The most successful portion of my project that was revealed during playtesting was the organization of the user interface. Players found the interface to be easily intuitive. Specifically, the broad range of different WPF tools to organize information; such as the TreeView used in the Craft Menu to allow the player to expand the ingredients of each recipe when desired. Although it is important to acknowledge, that this could be biased due to the fact that play testers not only were familiar with the project, but have created versions themselves – therefore, it is unsurprising that these playtesters would know exactly what their intended goal was upon start. So, despite positive feedback a detailed instructions, menu was added after playtesting.

While play testers didn't comment on this in their feedback, one issue I observed was that sometimes the player would get stuck in a situation without any funds to continue. They would not have enough money to buy new ingredients and would not have enough ingredients to craft something to sell. As a result of this behavior, I added a Tips button despite it not being in the functionality requirements. This allowed the player to hit the button to increase their money, like a clicker game. However, because this button was not the intended way for players to earn money the button only increases the player's funds by 0.01$, to encourage players to use it only when they are completely cornered.

Another issue that was found was that because my code was set to remove items that had been structured to check if an item had a quantity greater than or equal to 1, or if the quantity was less than 0, and then remove the item accordingly. This was done so that items that had reached a quantity of 0 would be automatically removed from the player's inventory instead of creating clutter. This code had an issue in that it did not account for half cups. Items that had a decimal in their quantity would allow the player to just continue using it. If the vendor had been bought out of something on a decimal, the player could continue buying into the negatives. This was fixed by simply adding another if else statement that would take into account if the item was less than 1 but more than 0.

Test Session (April 22nd)

1) Christopher Leon

2) Leo Richnofsky

Most Successful Project Aspects

- Aesthetics – Players enjoyed the colors that differentiated the sections, and the stylization of information.
- Organization – Players enjoyed the organization of information, for example, that things like the recipes were formatted in a drop down menu so that they could be expanded and collapsed.
- Ease of repeated action – I found the inclusion of extra buttons (for example, arrows allowing the player to increment what they wanted to buy instead of clicking confirm multiple times) to be successful in contributing to ease of use.

Biggest Project Issues

- My project needs to be further completed. At the time of playtesting, the buy and sell window was not completely functional.
- My buy and sell options may need to be separate windows in order for their functionality to be clearer. As of now, it is not intuitive that you can interact with the player inventory on the sidebar and then hit sell on the main window.
- My project needs an instruction window. I think because the players had the context of the assignment uses the application was easier, but I do not think that a player completely new to the functionality would have the same success.

How did you use what you learned to improve your application?

After playtesting I separated my buy and sell windows, making the Customer and Vendor two separate entities. This made the software easier to understand as it was difficult for players to understand that the inventory menu (which was nothing more than display in other windows) would need to be selected in order to sell an item. Separating these also allowed me to make the program more engaging by giving the Vendor and Customer separate dialogue and names. Making this change also made my code easier to organize, because the code for the trade window was no longer trying to both use the code for selling and use the code for buying.

Another bug ran into during playtesting was the fact that changes made to items bought from the vendor would affect the same type of item in the vendor's inventory. For example, if the player bought an item from the vendor, the amount they had would change the vendor’s stock quantity (they buy 6, so the player’s inventory count changes to 6, but also changes the vendors inventory count to 6), and if they sold the item to a customer, it would deplete the vendor’s stock as well. This issue spawned from my own lack of understanding how adding an object (or anything for that matter, but it’s much less catastrophic with something like a string) to a collection would add the exact same reference, rather than creating a duplicate. This was fixed by creating a member wise clone of the item and adding that clone to the player’s inventory. My learning of what a member wise clone is, is further elaborated on in the Research section.

Credits

The base structure of the file was used from Janell Baxter’s Programming 2 ProgrammingIICraftDemoPages. Otherwise, all code used in the project was my own. However, many sources were referenced when creating said code. These sources include:

Bill Wagner. n.d. “C# Docs - Get Started, Tutorials, Reference.” Learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/csharp/.

Adegeo. 2022. “Getting Started - WPF .NET Framework.” Learn.microsoft.com. March 5, 2022. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/getting-started/?view=netframeworkdesktop-4.8.

Adegeo. 2023. “XAML Overview - WPF .NET.” Learn.microsoft.com. June 2, 2023. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-8.0.

w3schools. 2022. "C# Tutorial (c Sharp)." Www.w3schools.com. 2022. https://www.w3schools.com/cs/index.php.

"Shallow Copy and Deep Copy in C#." 2019. GeeksforGeeks. January 19, 2019. https://www.geeksforgeeks.org/shallow-copy-and-deep-copy-in-c-sharp/.

"Deep and Shallow Copy in C#." n.d. Www.partech.nl. Accessed May 6, 2024. https://www.partech.nl/en/publications/2021/11/deep-and-shallow-copy-in-c-sharp.

Specific pages of these sources are further elaborated on/included in the research portion below.

Research

When playtesting my project one bug noticed was that if the player had bought an item from the vendor, further action on that item would effect the remainder of the vendor's stock/inventory. For example, if the Player bought X item from the Vendor, then used that item to craft something else and depleting their inventory of said item, upon opening the Buy Window, the Vendor would be out of stock of that item. Another case of this was shown when the player bought an item from the Vendor, the Vendor's stock would immediately decrease incorrectly (The Vendor would have a 10 count of an item, the player would buy 4, and because the player's item count would be 4, the Vendor's stock would be 4 instead of 6). The reason for this was because I was using add on the finished item (CraftedRecipe variable), which was adding the exact same reference of the item in the vendor's inventory to the player's inventory. What I needed to be using was a memberwise clone of the item. When learning this I referenced this Microsoft Learn page:

dotnet-bot. n.d. "Object.MemberwiseClone Method (System)." Learn.microsoft.com. Accessed May 6, 2024. https://learn.microsoft.com/en-us/dotnet/api/system.object.memberwiseclone?view=net-8.0.

The memberwise clone creates a shallow copy of the CraftedRecipe variable associated with the Item in the vendor's inventory. This clone contains the ItemValue that is multiplied by the rarity multiplier in the Item Class, which uses probability to add a quality level to each item crafted. The article "Shallow Copy and Deep Copy in C#" cited in the Credits section is where I learned the difference between a shallow copy and a deep copy.

A shallow copy is used for creating a new object with the value types of a preexisting object into the new object. However, for reference types the memory will still be pointed to the same location. Therefore, if changes are made to one reference type it will affect the other. For this use, a shallow copy worked. Conversely, a deep copy does not share any objects with the original it's copied from.

Another topic I researched throughout creating this project was different WPF tools. For the Craft Page I used a TreeView to display the recipes. This way the ingredients could be organized neatly. As this was my first time using WPF this led to further research on how to use this tool.

Adegeo. 2023. "TreeView Overview - WPF .NET Framework." Learn.microsoft.com. February 6, 2023. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/controls/treeview-overview?view=netframeworkdesktop-4.8.

This led me into looking into data binding because this made sense for formatting the information. This way, instead of repeating code, the text in the TreeView could directly bind to the corresponding Recipe object's values (RecipeRequirements).

Adegeo. 2023. "Data Binding Overview - WPF .NET." Learn.microsoft.com. September 2, 2023. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-8.0.

I also did further research into the ICommand interface that goes with XAML. Originally, I was using the Button_Click feature of WPF, but I found this wasn't complex enough for what I was trying to do. By attaching the ICommand interface, I used the EventHandler CanExecuteChanged, CanExecute, and Execute. This made sense for the increment and decrement buttons because at certain points they would no longer be able to execute. For example, if the player sets the buying count equal to the stock of the Vendor, the increment button should disable. The following webpages were referenced:

Dotnet-bot. n.d. "ICommand Interface (System.Windows.Input)." Learn.microsoft.com. Accessed May 6, 2024. https://learn.microsoft.com/en-us/dotnet/api/system.windows.input.icommand?view=net-8.0.

Palkar, Rikam. n.d. "ICommand Interface in MVVM." Www.c-Sharpcorner.com. Accessed May 6, 2024. https://www.c-sharpcorner.com/article/icommand-interface-in-mvvm/.

Instructor Provided Research

KptnCook.com. "3 Ingredient Chocolate Cake Gluten Free and Dairy Free." KptnCook. https://blog.kptncook.com/2015/11/05/3-ingredient-chocolate-cake (accessed February 3, 2022).

Tablespoon.com. "Powdered Sugar Icing." tbsp. https://www.tablespoon.com/recipes/powdered-sugar-icing/7b63e81d-a088-473a-8bca-74bb1641fc8d (accessed February 3, 2022).

WebMD.com. "Health Benefits of Cacao Powder." WebMD. https://www.webmd.com/diet/health-benefits-cacao-powder (accessed February 3, 2022).

WebMD.com. "Health Benefits of Eggs." WebMD. https://www.webmd.com/diet/ss/slideshow-eggs-health-benefits (accessed February 3, 2022).