Enhancement 1 Narrative

Ryan Reese

The Game:

The game I am working on involves a inventory system where the player has a limited number of resources. The system I had previously gave the player 2 types of resources that were used at different parts of a player's turn. After prototyping the game I decided I could go back and change this part of the game.

First the inventory. Due to the rushed nature of the game's prototyping, made in one week for a game jam, I had a few band aid fixes that caused problems in development. One was the fact that the inventory system did not manage the number of items you carried, the items carried that data. I plan on fixing this with a inventory slot middle man. Next was a change to combine the two types of resources into a single class and use properties to make differences between resources. This involved culling 4 classes that were duplicated and reformatting a UI element to handle the changes in a turn of combat.

Making the changes:

First was the changes to the inventory system they were straight forward:

```
1  //This class is meant to be a middle man between the inventoryManager and the ingredients
2
3  using System;
4
5  [Serializable]
   5 references
6  public class InventorySlot
7  {
8      /// <summary>
9      /// how much of the Ingredient this slot holds
10     /// </summary>
11     public int Amount;
12
13     /// <summary>
14     /// the ingredient that the slot holds
15     /// </summary>
16     public Ingredient Ingredient;
17
18     /// <summary>
19     /// Sets the slots amount to the given number
20     /// </summary>
21     /// <param name="newAmount">the number the amount will be set to</param>
       0 references
22     public void SetAmount(int newAmount)
23     {
24         Amount = newAmount;
25     }
26
27     /// <summary>
28     /// adds this number to the current amount
29     /// </summary>
30     /// <param name="amountChange">number to add to the amount</param>
       1 reference
31     public void AddAmount(int amountChange)
32     {
33         Amount += amountChange;
34     }
35
36 }
```

First this inventory slot was created, this would hold an ingredient (the resources mentioned earlier) and then hold the amount of that ingredient. It was also given a way to set the amount to any number and a way to add to is amount.

This required me to changed parts of the inventory manager:

```
Unity Script (1 asset reference) | 3 references
public class InventoryManager : MonoBehaviour
{
    public static InventoryManager Instance;

    public List<InventorySlot> IngredientsList;
```

The IngredientsList was changed from type List<Ingredient> to type List<InventorySlot> allowing the inventory slots to play their roles. I also made changes to the UI scripts:

```
Unity Script (1 asset reference) | 3 references
public class IngredientsPanel : MonoBehaviour
{
```

```
    1 reference
    public void SetSelectedSlot(InventorySlot slot)
    {
        _selectedSlot = slot;
    }
```

Within the IngredientsPanel script, a variable would track which ingredient the player had selected but with the inventory slot change the _selectedSlot variable had to be changed to a type InventorySlot and so did the respective methods.

```
0 references
public void AddIngredientToPotion()
{
    if (_selectedSlot == null)
    {
        ErrorMessaging.instance.ShowError("No Ingreident Selected");
        Debug.Log("No Ingredient Selected");
        return;
    }

    if (_selectedSlot.Amount == 0)
    {
        ErrorMessaging.instance.ShowError("No Ingreidents Left");
        Debug.Log("No Ingreidents Left");
        return;
    }

    if(_selectedSlot.Ingredient.APCost > GameManager.instance.Player.CurrentAP)
    {
        ErrorMessaging.instance.ShowError("Not enough AP");
        return;
    }
```

Additionally the parts of the script that referenced the Ingredient now needed to be adjusted to reference the Ingredient through the slot so these if statements to check the validity of the selected ingredient needed to get the ingredient from the _selectedSlot first.

The final UI element to change due to this change was the IngredientCard script, this script formats the buttons that allow the player to select his ingredients.

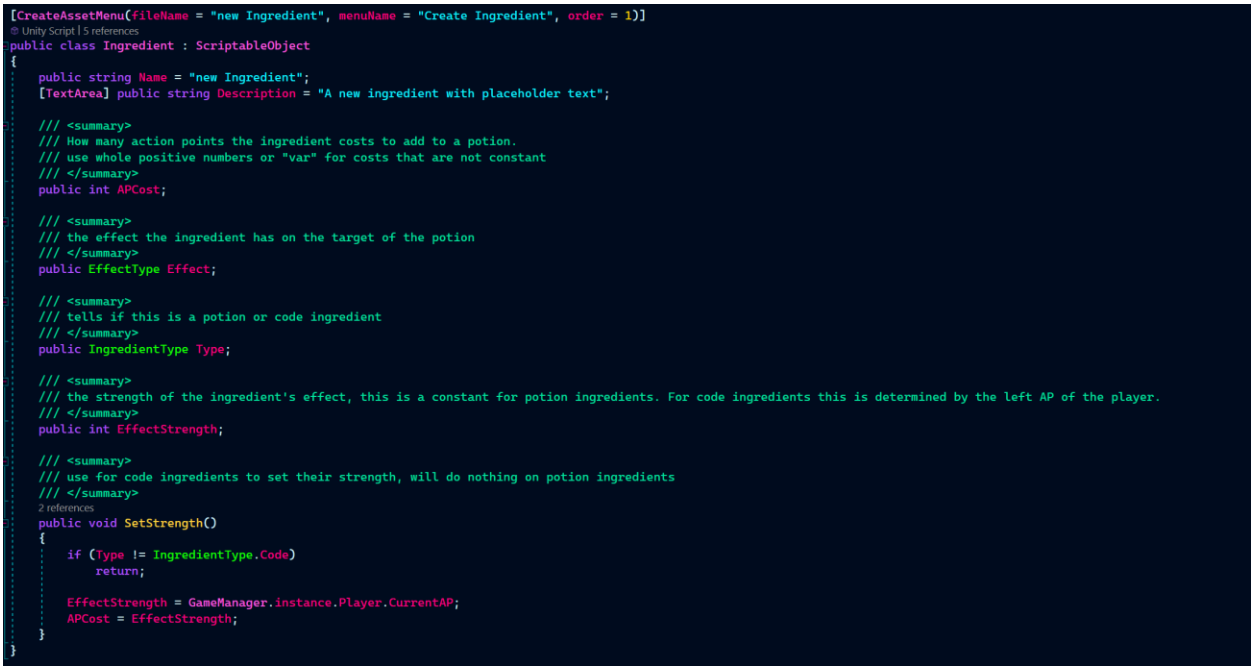
```
Unity Script (7 asset references) | 2 references
public class IngredientCard : MonoBehaviour
{
```

Originally this class would hold an ingredient type to pull the data from but with the new InventorySlot system I had to change that:

```
public void Initialize(IngredientCardData data, InventorySlot slot)
{
    _slot = slot;
    _name.text = slot.Ingredient.Name;
    _amount.text = slot.Amount.ToString();

    // Sets the numbers for an ingredients cost and strength if needed
    slot.Ingredient.SetStrength();

    _apCost.text = slot.Ingredient.APCost.ToString();

    if(slot.Ingredient.Type == IngredientDataOptions.IngredientType.Code)
    {
        _apCost.text = GameManager.instance.Player.CurrentAP.ToString();
    }

    if(slot.Amount == 0)
    {
        _button.interactable = false;
    }
}
```

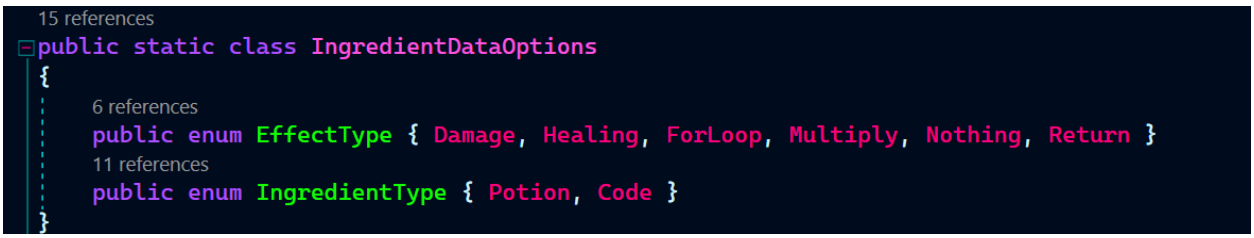
The Initialize function correctly formats and binds the inventorySlot to a button the player can press. The data variable is the data used to format the size, colors, and fonts the button uses while the slot variable is the Inventory slot that the button represents. Originally slot was the ingredient instead and had to be replaced due to the introduction of the InventorySlot.

The next change I will be going over is the change to the way ingredients are separated. I originally tried to create two classes for ingredients a potion ingredient type and a code ingredient type. Instead I merged them into a single ingredient type:



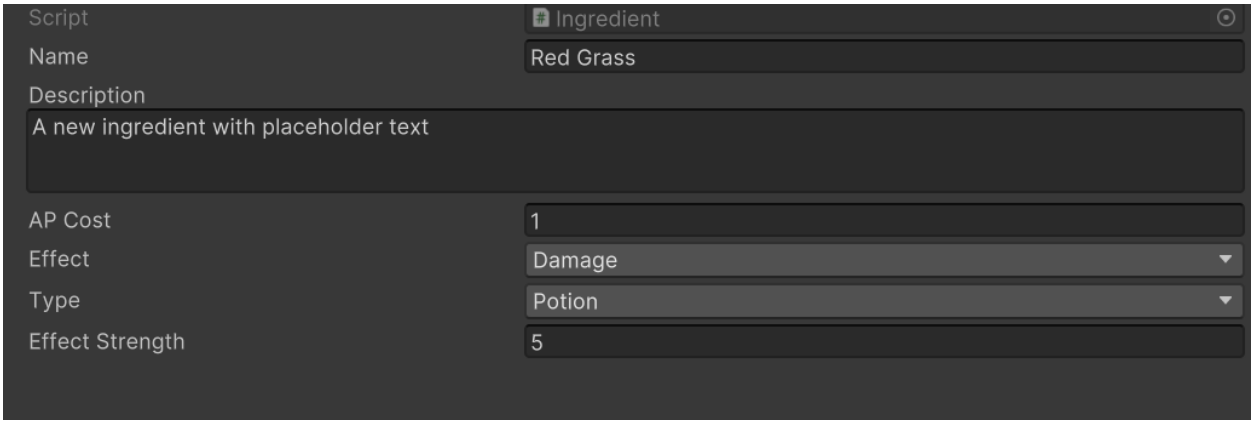
```
[CreateAssetMenu(fileName = "new Ingredient", menuName = "Create Ingredient", order = 1)]
Unity Script | 5 references
public class Ingredient : ScriptableObject
{
    public string Name = "new Ingredient";
    [TextArea] public string Description = "A new ingredient with placeholder text";

    /// <summary>
    /// How many action points the ingredient costs to add to a potion.
    /// use whole positive numbers or "var" for costs that are not constant
    /// </summary>
    public int APCost;

    /// <summary>
    /// the effect the ingredient has on the target of the potion
    /// </summary>
    public EffectType Effect;

    /// <summary>
    /// tells if this is a potion or code ingredient
    /// </summary>
    public IngredientType Type;

    /// <summary>
    /// the strength of the ingredient's effect, this is a constant for potion ingredients. For code ingredients this is determined by the left AP of the player.
    /// </summary>
    public int EffectStrength;

    /// <summary>
    /// use for code ingredients to set their strength, will do nothing on potion ingredients
    /// </summary>
    2 references
    public void SetStrength()
    {
        if (Type != IngredientType.Code)
            return;

        EffectStrength = GameManager.instance.Player.CurrentAP;
        APCost = EffectStrength;
    }
}
```

Compared to the ingredients before the only new thing here is the IngredientType type variable, this Is used to create the separation that the two classes had before. Also this class is a scriptable object as well, this allows me to make the creation of more ingredients and the storage of their data easier in Unity. Finally you'll notice the IngredientType and EffectType data types, these are created with a static class call IngredientDataOptions:



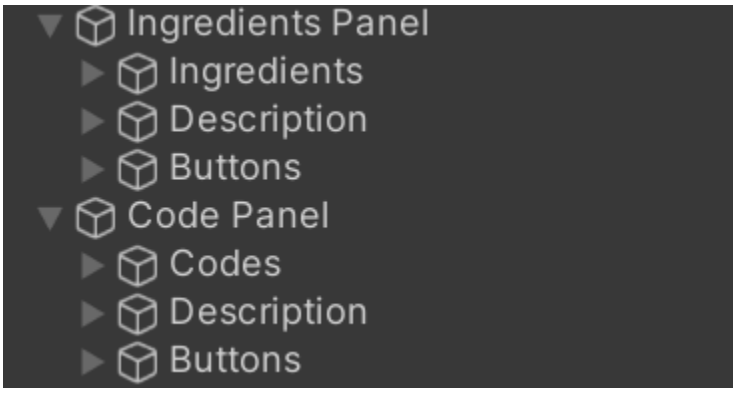
```
15 references
public static class IngredientDataOptions
{
    6 references
    public enum EffectType { Damage, Healing, ForLoop, Multiply, Nothing, Return }
    11 references
    public enum IngredientType { Potion, Code }
}
```

These enums are used got storing the possible types of ingredients and their possible effects. I did this so in Unity you select the types in a drop down menu instead of using strings to type them in:



This is the menu you get when modifying an ingredient, The effect and type areas have a drop down to make it easier for designing. Now I don't have to open up my IDE to check for the wording I used for these variables.

The next big thing this change influenced was the way the UI worked. The separation between potion ingredient and code ingredient classes forced me to create two different UI menus, two that functioned identically:



The scripts for them were deleted by here you can see their structure was the same, a menu that shows a list of ingredients that the player has access to, a description of the ingredient they had selected, and finally the buttons to add the ingredient or move on to the next step. The only differences were the ingredients panel handles the ingredients that use the PotionIngredients class and the code panel handles the ingredients handled the ingredients that used the CodeIngredients class and the ingredients panel had a button under the Buttons category that the code panel did not have. After Merging the ingredient types together I had a mission on my hands to merge the panels together and have them correctly show both types of ingredients separately. The biggest thing I had to change was how the panel would list out ingredients:

```
Unity Script (1 asset reference) | 3 references
public class IngredientsPanel : MonoBehaviour
{
```

```
3 references
public void ListOutIngredients(IngredientDataOptions.IngredientType type)
{
    foreach (var c in _cards)
    {
        Destroy(c.gameObject);
    }

    _cards.Clear();

    if (type == IngredientDataOptions.IngredientType.Potion)
    {
        _throwButton.gameObject.SetActive(false);
        _addIngredientButton.gameObject.SetActive(true);
        _codeButton.gameObject.SetActive(true);
    }
    else
    {
        _throwButton.gameObject.SetActive(true);
        _addIngredientButton.gameObject.SetActive(false);
        _codeButton.gameObject.SetActive(false);
    }

    foreach (var i in InventoryManager.Instance.IngredientsList)
    {
        if (i.Ingredient.Type != type)
            continue;

        var card = Instantiate(_prefab, _selectionParent);
        card.Initialize(_data, i);
        _cards.Add(card);
    }
}
```

This ListOutIngredients method belongs to the IngredientsPanel class and is used to List out the ingredients by a type. This is how I created the divide between the code ingredients and the potion ingredients that classes once did. When listing ingredients the panel only lists out the ingredients that match the type given. The foreach loop at the bottom of the method is responsible for that. There is also that if else statement in the middle of the method, before I mentioned the fact that the potion ingredient panel had an extra button that the code ingredient panel did not have. This if else statement is a fix for that, now the panel will hide buttons depending on if it is showing the potion ingredients or code ingredients.

Here is the menu for potion ingredients ^

Here is the menu for code ingredients ^

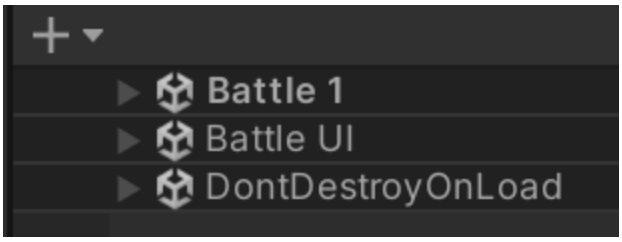
these both use the same script and you see the difference with the buttons on the right, meaning my script is working.

Final part I want to touch on is a change I made to how the UI works. This doesn't relate to the main goal of this enhancement but it is on the UI topic. Before my game was made up of 5 scenes of battle:

This is the list of all the battle scenes within the game ^

Each of these held the data for each battle and the game would move between these to progress the game. The problem with this was my UI, each scene had its own copy of the UI and to change the UI I had to change it in each scene. This meant each change had to occur 5 times. But now my UI has its own scene:

Which is loaded in when a fight begins:



This is a list of the scenes that exist during gameplay ^

To make this work the Battle UI scene binds itself to parts of the battle scene:

```
public class PlayerActionUI : MonoBehaviour
{
    [SerializeField] Image _brewPanel;
    [SerializeField] Image _ifPanel;
    [SerializeField] IngredientsPanel _ingredientsPanel;


    Unity Message | 0 references
    void Awake()
    {
        BattleTurnManager.instance.OnPlayerTurnStart += PlayerTurnStart;
    }
```

Above is the PlayerActionUI class, it is in charge of the UI the player interacts with on their turn. In the Awake method I set up a listener for the OnPlayerTurnStart action, this way when the player's turn starts the PlayerActionUI will respond accordingly. I also had to do this with health bars:

```
Unity Script (12 asset references) | 0 references
public class HealthBar : MonoBehaviour
{
    [SerializeField] Image _filler;
    [SerializeField] Fighter _owner;
    [SerializeField] string _id;

    Unity Message | 0 references
    private void OnEnable()
    {
        if (_id == "Player")
            _owner = GameManager.instance.Player;
        else if(_id == "Enemy")
            _owner = GameManager.instance.Enemy;

        _owner.OnHealthChanged += UpdateHealth;
    }
```

Here we see the same pattern, the difference here is deciding who the health bar belongs to, so I set up an _id string to handle that. These where the only two changes that needed to be made to this to save me time on bouncing between scenes to change UI 5 times as much as I need to.

What I learned:

Briefly describe the artifact. What is it? When was it created?

The artifact was a game jam game created in a week. The game got 2nd place in the game jam.

Justify the inclusion of the artifact in your ePortfolio. Why did you select this item? What specific components of the artifact showcase your skills and abilities in software development? How was the artifact improved?

The artifact is a good example of rapid prototyping since it was made in one week to show off a concept. The continuation of this artifact will show my abilities to work with and improve the prototype that I made. In this enhancement the artifact's code and UI are now more readable and expandable while also containing less duplicate code.

Did you meet the course objectives you planned to meet with this enhancement in Module One? Do you have any updates to your outcome-coverage plans?

Yes I met the course objective of demonstrating an ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals. Although I didn't do it how I

wanted, originally my plan was to use generics in C# to bridge the gap between the Code and Potion ingredient classes but I found it easier to just combine them.

Reflect on the process of enhancing and modifying the artifact. What did you learn as you were creating it and improving it? What challenges did you face?

I learned to not over think things while developing. The problem stemmed from the use of two different classes for the ingredients. I originally did this because I thought they were going to be far different. This overengineering was done early on and bloat my project making more work than necessary. I will be more aware of this possibility in the future. The largest challenge I faced was the UI change I made making the UI load in separately from the rest of the scene. This was completely new territory for me and I was proud to finally get it working.