Inventory System Team Assignment

IMM215 Programming For Immersive Experiences

# Objective

Group project in groups of 3.

Create a system that uses a parent/child class inheritance, a working example project that uses it. You do not have to create, literally, an inventory system. It can be something else.

First, I recommend building off of one of your character controllers, or use a character controller from the sample code.

* A parent/child class setup for something in your project . (Items)
* A system that allows other elements in your project to use these (The inventory system)
* This system calls a virtual function on the parent Item class, which have the children classes have override functions that they call. (Items could have “pickup”, “put down”, “use”, and the child items have their own behavior happen)
* An interface should exist to let other parts of your project hook into and out of the system (Check if you have an item that has some property (name==”blueKey”)) in your inventory). The degree to which the interface is used and what is required depends on your example project.
* A user interface to display the items is **not** required for the example project.

Every member of the team should have their own commits in the github, and clearly defined responsibilities on the project. Every student should understand the inventory *system*, but with the example implementation, it’s is less important for all students to have worked on all parts of it. Some students may not create the code for the system, but worked on something else, like the example project or the optional UI to display items.

# Submission

We have been talking about architecture and re-usable code, but this project will not held to such high standards. The inventory system doesn’t need to be perfectly separated from the rest of the project, or pluggable into any new project. Strive for these design patterns, but this is a more challenging assignment with more moving parts.

Submit a link to the repository**. Include a readme.md file in the base directory of your project.** It should include student names, a description of the project, and a brief description of how collaboration went (who did what).

# Purpose

Writing flexible code that has these kinds of useful parent/children classes is **fundamental** to creating immersive experiences. In VR systems, we will often create a parent class for something like “Interactable” that contains code that lets us pick up and put down an item, and calls a virtual function or event if we press a button while holding it. Guns, coffee mugs, whatever, might all be “Interactable”.

Also, Inventory systems are common in immersive media applications too. Consider the one in *Garden Of The Sea*.

We are staying with the 2D project world to minimize the time spent working on the example project, and maximize the time spent structuring the system underneath it.

It’s useful to start spending time with Unity’s UI system. While not really part of this class, and not required as a graded element of the project; We are going to need to know this eventually. In VR, the UI usually isn’t a HUD, but floating in front of us, but that’s still UI in a canvas. Created the same way.

This project also has us working in larger groups where communicating in real-time with other members isn’t always convenient. Effective work will require using our version control system well.

# Grading Information

Rubric Breakdown

40% Inheritance – You have a system with parent classes (with virtual functions) and child classes (that override those functions), and you call functions on the base class and have unique functionality happen because of child classes overriding those functions.

15% System other. The code. How well it works and how efficient it is. Code should be commented and use the inheritance system. If you do something other than an “inventory”

15% Example project should function with the system.

20% of the grade will be on how well the team collaborated, if at all. If only one student did most of the work, everyone will lose this part of the grade.

10% The Git repository. Are commit messages descriptive? Was code added to branches and then merged, by feature/if necessary?

# Getting Started

You have one major decision to make: Where do the instances of items “live”:

They could be, themselves, extensions to “monoBehaviour”, which are components on gameObjects. If we destroy the object, the item goes away. This might not be good! We may wan to destroy the object in the scene, because we picked the item up!

A second way to do it would be to hook it into the UI. We pickup some sceneItem object that has a reference to a prefab, and that prefab is the icon of the item in a menu. So the prefab being the “item” that is stored in our inventory, as well as being the UI stuff. So instantiating that prefab as a child object of the right UI manager thing creates the icon of the item.

A third way to do this would be to have items as classes that don’t extend monoBehaviour. We create new copies of them (item = new Item()), not worrying about “instantiating” because its just a class like any programming thing. Then some sceneObject tells us which one to create and store. I officially recommend this method to get started.

Inventory systems don’t need to be “items”. You could write a system for changing a characters outfit, their currently loaded weapon, or other such similar system. Still an inventory, but make it what you need for your example project (aka, what sounds interesting to your group).

A fourth way is to use a data layer, and… basically scriptableObjects. We will learn about ScriptableObjects in the lecture following this one, so maybe you might end up re-writing or refactoring your code to use this sort of system.

---

First, make sure you aren’t ignoring your warnings. There are lots of helpful warnings that *aren’t* errors that *are* telling us how to fix our code so it will work correctly.

For example, “name” is a thing classes already have, so if we make a string called “name” it will override something important.

The inventory system could be used for things like collecting items, keys, or picking up weapons, outfit-changes, and more. Free-reign on the “game” (“game-segment?”) you are creating.

This project involves not just the system, but a working example of the system.

For a UI, I would start with a Grid, Vertical, or Horizontal Layout Group. I would make the UI object of that group destroy all of it’s children (you can loop through children! foreach(Transform child in transform){Destroy(child.gameObject);}, then get all of items in the inventory (thanks, system-we-will-write-to-give-us-this), and instantiate prefabs. When instantiating the prefabs, we can run a GetComponent and change some property like a text’s text to some property of the item we are looping through, like an items name. (for every item in the inventory, make a new text prefab. Get a reference to it’s Text component, and change that components text property to be the string that is the name of out item).

The manager is then a child of some “display menu” manager – or whatever – that can enable and disable it, to show/hide it.

In order to have many types of items, we are going to create a parent class, lets call it “Item” I suppose, that will contain properties like “string name;” “sprite UIicon;” and “sprite sprite;” as well as *virtual void UseItem(){}*, or such a function that the children of item will overwrite with their unique thing.

Each item we have will be a child of item, similar to how all of our scripts so far have been children of monoBehaviour. We just replace the monoBehaviour with the name of our script in the class declaration line, and bing-bang-boom. It’s object oriented! All the children classes can get at the *public* or *protected* properties of the base class. So they all will share name, icon, etc. But will do different things when we try to Use(); them.

What’s great about this base-class thing is that the inventory manager can contain a list of Item’s, and we can add Key or Weapon or Hat or whatever child objects to that list.

We can keep a list that is of a bunch of *different* item types that do *different* things, but are basically the same in regards to the inventory, so we can treat them as if they are “just” their parent class.

Since they are “just” stored as their parent class in a list of <baseclass>, we can only reference the parent class properties and functions, since we don’t know if it’s a key or weapon or hat or literally just an “item”. (but the object still is what it is).

That’s where the override comes in! If the child overrides some function Use();, calling the base-class ‘virtual’ version of it, then because of the override the game will call the child’s version of it! Wow! Yay! Cool!