- Virtual in ItemFactory Script, with getitem class function

- Virtual is used to override the base class member in its derived class based on the requirement

- override is used in specific items, to override item attributes

- get in c# returns the property value or the indexer element.

- What are you testing? Why did you choose these tests?

- Stress test (coin max and min, checking whether the player would not pick up the coin if going at a specific speed. Chose this to test the bounds and to test what speeds the player stops working at

- Coin pickup, for setting the score, tested this because it was important to make sure the score was setting accurately when a coin is picked up

Prefab: Coin

Who would be asking? Josie for UI with score, Riley for saving score, might need to know how items are saved in inventory for loading the game

Super class: ItemFactory

Sub Class: KatanaFactory

Virtual Function: GetItemClass()

Choose a dynamically bound method. What method gets called now?

- ItemFactory - new ItemClass() gets called

Change the dynamic type. What method gets called now?

- KatanaFactory - new Katana() gets called

Pick a statically bound method. Which one would be called in each case:

- for both, ItemFactory - StaticDebug() because this is the parent and it goes directly to StaticDebug() every time its called, regardless the case.

Copyright violation: Zelda Master Sword

- I violated copyright with the Zelda Master Sword. This violates copyright because, according to the Copyright Act, as long as there is some creative or original act involved in development, it is copyrightable. The Zelda Master Sword is an original and key component of the Zelda series, therefore encompassed within Nintendo’s copyright of Zelda.

My patterns:

- Factory Pattern

-defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Used for selecting specific attributes for items.

- Strategy Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. Used in IInteractables. A strategy is abstract and doesn't define itself, but instead defers the definition to all classes that implement it.

Why I chose Factory:

I chose the Factory pattern, one of the creational patterns, because it enables the ItemClass to defer instantiation to specific subclasses correlating to each item.

Why I chose Strategy pattern:

I chose the Strategy pattern, one of the behavioral patterns, for the IInteractable script because this creates an interface that includes an interact() function that can be accessed independently by anyone on the team that needs objects to interact, without them having to write up their own interact script or edit anyone else’s to make theirs work.

Strategy pattern for my project:

GroupMemebers is implementation, IInteractables is abstraction

Diagram

Description automatically generated

This pattern would have worked the best, but a builder design pattern could’ve worked too.

- Separate the construction of a complex object from its representation so that the same construction process can create different representations. It would be a bad time to use this pattern if all your object’s fields are required upon construction.

Would be bad to use Strategy because it must be aware of all the strategies to select the right one for the right situation.