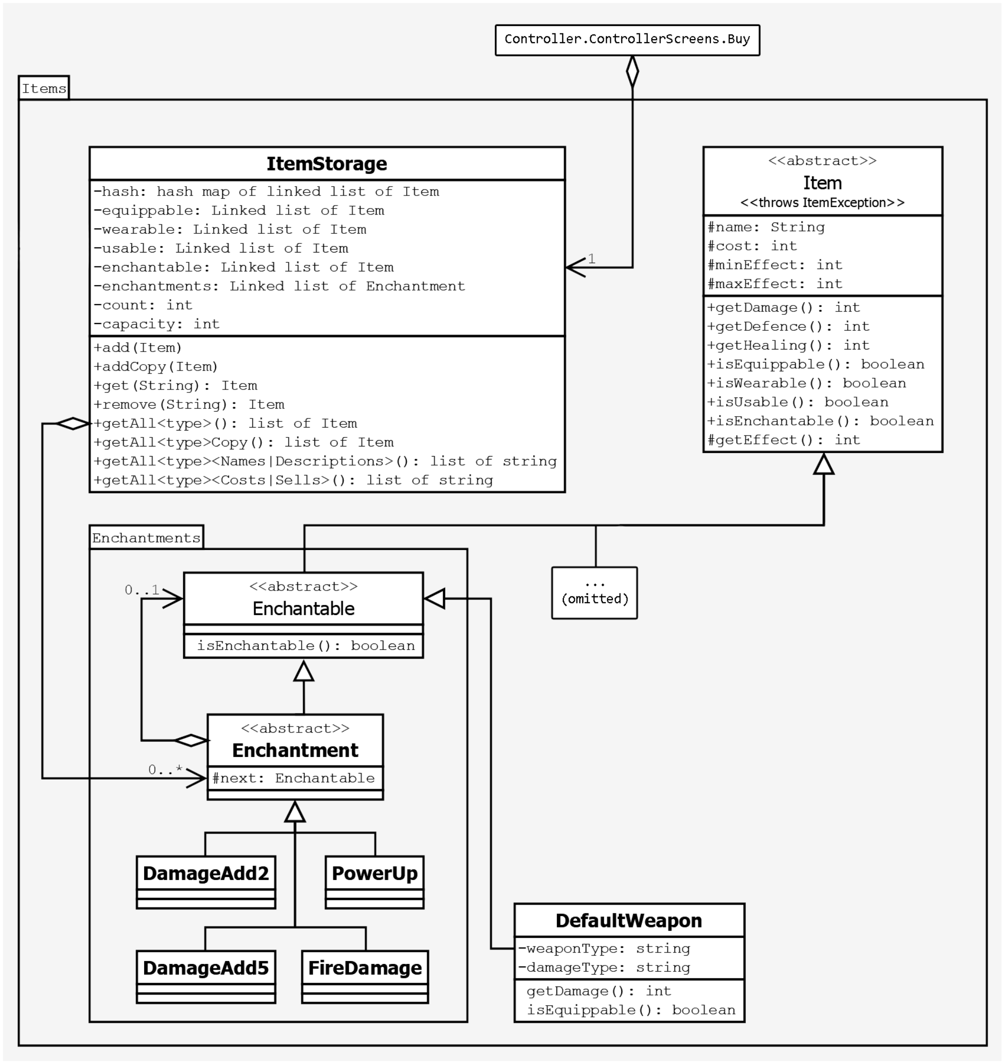
Discussion

Design

The Decorator Pattern for Enchantments

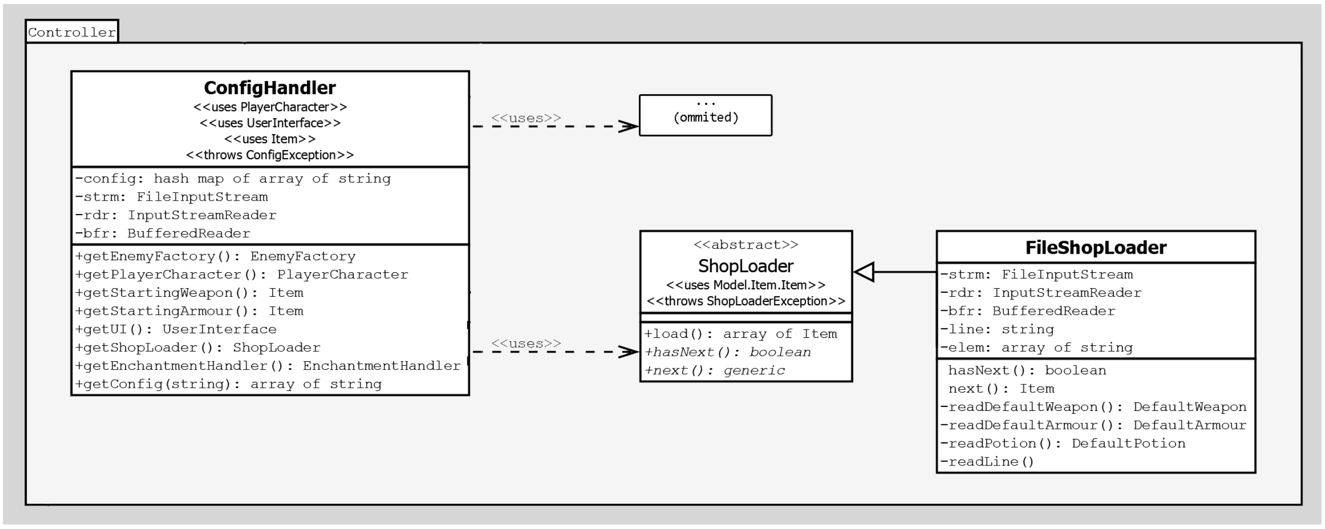
I thought that enchantments were the perfect use for the decorator pattern. I ended up with a modified Decorator Pattern where the recursive aggregation is zero-to-one rather than exactly one because I wanted enchantments to be able to be stored in the shop without having a 'next' field set.
Items that can be enchanted inherit from Enchantable rather than Item. Enchantment is the decoration class that holds the 'next' Enchantable field. Both DefaultWeapon and all the implemented concrete Enchantments override the getDamage method from Item, with Enchantments returning a modification of 'next.getDamage()', with DefaultWeapon acting as normal - returning a number between it's min and max damage.



Discussion

The Template Method Pattern for Shop Loading

I thought that the Template Method Pattern would be a good way of implementing a polymorphic shop loader. I used a 'load()' method in a ShopLoader abstract class that calls hook methods for the concrete shop loader to implement. At first I was defining 'getNextItem()' and 'isComplete()' hooks but I realised that I was just re-defining an Iterator, so I ended up making ShopLoader an Iterator and an Iterable and having the abstract Iterator methods be the hooks, and have ShopLoader's template method simply iterate over itself to get Items.



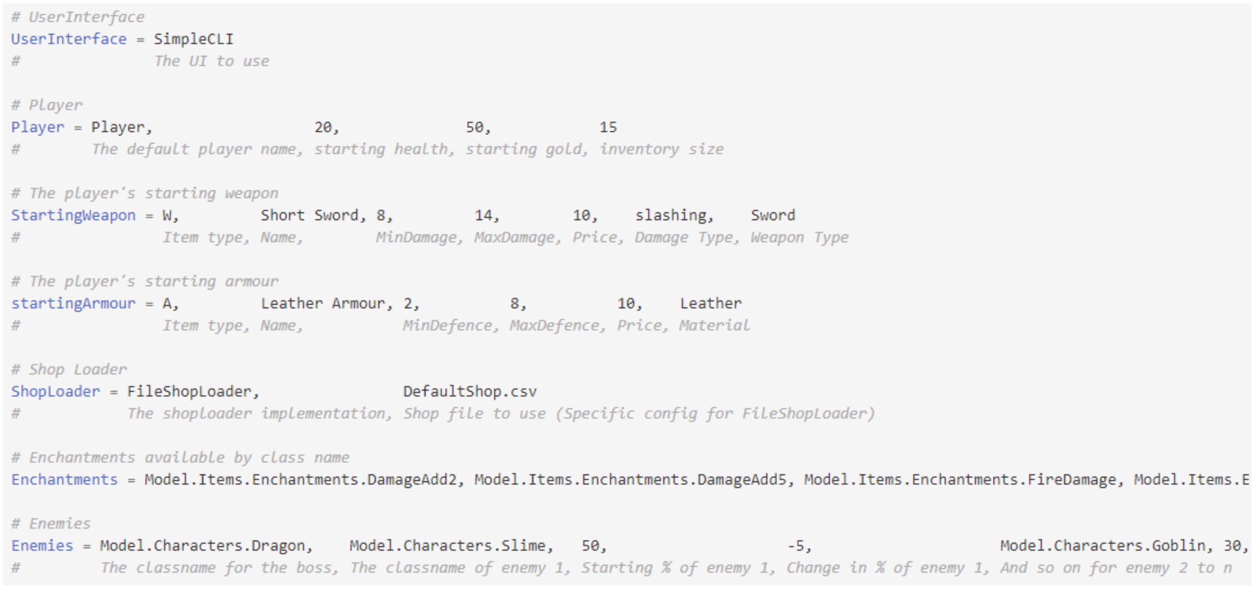
Architectural Separation

With a project this large there was no way I could stay sane without some sort of packaging, and I found Model-View-Controller helped me a lot in planning and good overall design with (fairly) minimal cross-package communication.
I further separated my Model into Character (For Player and Enemies), Items (For Item storage, and concrete items) and Items.Enchantments.
My View just contains a UserInterface Interface and a concrete class that implements it as a simple command line interface. I made my UI polymorphic because I planned to later implement a flashier UI, but I didn't end up having time to do that.
My Controller is further separated into ControllerScreens, which are all of the different states that the program can be in

Discussion

Expandable Dependency Factories

Am I using enough OOSE jargon in that heading?
I ended up putting a lot of effort into making a configuration file system where a configuration file is supplied that details the dependencies of the program.
This is all handled by a class called ConfigHandler, which basically reads the config file and provides a bunch of factories based on it.
For enemies and enchantments, the class name is given and the EnemyFactory and EnchantmentHandler respectively use reflective constructor objects, so they are completely unaware of the classes that they are dealing with.
UserInterface, StartingWeapon, StartingArmour and ShopLoader are all slightly less insane and just construct their objects from hard coded options.
All of this reduces coupling, making the program much more expandable and customisable, for example by swapping two elements in the config file you could make the Slime the final boss and have Dragons as a regular occurrence, if you are crazy or something.
Shown below is the config file provided in the submission.



```
# UserInterface
UserInterface = SimpleCLI
#               The UI to use

# Player
Player = Player,                  20,              50,            15
#        The default player name, starting health, starting gold, inventory size

# The player's starting weapon
StartingWeapon = W,         Short Sword, 8,         14,        10,    slashing,    Sword
#                Item type, Name,        MinDamage, MaxDamage, Price, Damage Type, Weapon Type

# The player's starting armour
startingArmour = A,         Leather Armour, 2,          8,          10,    Leather
#                Item type, Name,           MinDefence, MaxDefence, Price, Material

# Shop Loader
ShopLoader = FileShopLoader,                DefaultShop.csv
#            The shoploader implementation, Shop file to use (Specific config for FileShopLoader)

# Enchantments available by class name
Enchantments = Model.Items.Enchantments.DamageAdd2, Model.Items.Enchantments.DamageAdd5, Model.Items.Enchantments.FireDamage, Model.Items.E

# Enemies
Enemies = Model.Characters.Dragon,    Model.Characters.Slime,   50,                    -5,                     Model.Characters.Goblin, 30,
#         The classname for the boss, The classname of enemy 1, Starting % of enemy 1, Change in % of enemy 1, And so on for enemy 2 to n
```

I also made sure to implement my Items for expandability. Rather than the player only being able to equip/use a ‘Weapon’ class, I took a more open approach where Items decide if they are equippable (Eg. Weapons), wearable (Eg. Armour) or usable (Eg. Potions), and the checks are done based on this rather than checking the class itself to reduce coupling.
This way, new types of items can very easily be added, eg. enchantable armour, a weapon that can be thrown at the enemy when not equipped, and so on - just create the class, defining ‘isEquippable’, ‘isWearable’ and ‘isUsable’ accordingly, and update your shop loader to support it.

Other Various Design Decisions

I used dependency injection wherever it made sense for testability and reduced coupling.

ControllerScreens are easily expandable by creating a new implementation and adding it as an option in the Controller class.

PlayerCharacter and Enemy both inherit from 'GameCharacter', but this is more for code reuse than polymorphism.

I defined custom exceptions wherever I felt that a class should throw exceptions. All exceptions thrown from public methods are user defined. An *exception* to this is shop loader implementations, as you can't just throw whatever you want from Iterator methods.

When errors occur, exception messages are prepended with each throw up the call stack with a semicolon to separate messages, making for nice messages like "A fatal error occured: Couldn't construct ShopLoader FileShopLoader; File I/O error opening input file; Shop.csv (The system cannot find the file specified)". This is main() catching a ConfigException caused by a ShopLoaderException caused by an IOException.

Alternative Design Choices

The State Pattern

The ControllerScreens classes resemble something like the State pattern but do not explicitly implement it. This is mainly because I learnt the state pattern after implementing this base functionality.
I don't think this would have a major effect on the coupling and cohesion of the overall program, but could potentially improve code reuse and overall elegance.

A Different Approach to Enchantments

While I think the decorator pattern is the perfect way to implement enchantments, I'm not 100% happy with my final version of it.
The zero-to-one recursive aggregation of next rather than a strict exactly one aggregation like described in the Decorator pattern would have made for a more cohesive result, as currently the 'next' being set to null (As it is when the enchantments are stored in the shop) acts as a control flag for the behaviour of almost every method in the enchantments.
Also, having the enchantments inherit from Item makes for inelegant situations where enchantments have to set their min and max effect fields even though they don't actually use them for anything.