OOSE Assignment 2020  
Written Discussion

Jonathan Wright – 19779085

Contents

How The Design Works 3  
Alternative Design Choices 5

**How My Design Works**

**Characters**: Characters in my game use the ***Template Pattern***, this means that I have a superclass ‘Character’ declared abstract and having hook methods i.e getDamage, takeDamage. The reason I chose the Template Pattern over the Strategy Pattern is to eliminate code reuse (both enemies and player have same constructor + class fields, no need to have to do it twice just code it once) and to allow future extensibility, i.e in future if you wanted to add an Ally class which is friendly AI you could simply extend Character and implement the hook methods and it should just slot into the game. I adapted the Template Pattern to go two levels deep, by this I mean that while Player is a concrete implementation, Enemy is an abstract implementation that is also a Template Pattern, I did this to again allow future extensibility and reduce code reuse, it meant that to implement a new type of enemy you just have to extend Enemy and call the super, you also don’t have to reimplement the hook method if you don’t want to for an enemy, this is because I thought about it and not all enemies the future developer wants to add will have special abilities (it will just use the default implementation of getDamage), so rather than forcing them to implement a special version of getDamage they can just call the super and be done, and if they do want a special ability (defensive or offensive) they can override the getDamage and or takeDamage. Using Characters as the superclass allows for battles to be ***Polymorphic*** and in future the dev could choose to do AI vs AI or Player vs Player as I have designed it in a way that allows this. Each subclass and superclass’s cohesion is high as each enemy and player as a clear defined singular purpose of being a Character in the game. The template pattern also makes it likely that future characters developers implement will also have high cohesion as (if they are following good practices) it should end up having the clear goal of being a character in the game by implementing each method.

**Items:** Items in my game also use the ***Template*** pattern, I have a superclass ‘Item’ declared abstract and it has hook methods such as getEffect which all subclasses implement. The reason I chose Template pattern over Strategy Pattern is again to eliminate code reuse (weapon, armour, potions all have similar class fields and methods so only code once) and to allow future extensibility as to add a new type of item you would just extend item and implement getEffect. I didn’t have to adapt the template pattern for items as it just required me to simply slot in Weapon, Armour, and Potion (which damage potion and healing potion inherit Potion). Enchantments in my game use the ***Decorator*** pattern, this is to allow code re-use, (not all enchantments need to implement every method as the superclass Enchantment implements it so you can just implement what you need to implement). The Enchantments do not extend Item and this is something that I wish I could have gotten to work, however, I think I came up with a more logical solution of instead having Melee inherit both Item and Weapon, and Enchantment inherits Weapon meaning and Enchantment isa Weapon. I also allowed for future extensibility in the thoughts of while it is easy to add new enchantments, it is also easy to rework the code to allow for defensive enchantments as well (create a defensive interface and have enchantment inherit defensive as well), this would mean that you could create both offensive and defensive enchantments easily. Each subclass and superclass’s cohesion is high as each melee, armor, and potion has a clear defined potion i.e a melee is a melee weapon, armor is an in-game defensive item and a potion is an in-game consumable. The template pattern should also make it likely that future item types the developer implements will have a high cohesion as (if following good practices) it should have the clear goal of being an item in the game by implementing each method.

**How My Design Works**

**Shop:** The shop does not use a design pattern and is just a class that holds a few lists, I chose to not use lists of items as this would require the use of instanceof and it is an expensive operation. The way the data is loaded in however is **Polymorphic** as I load in the shop data using a DataLoader, the DataLoader can be a TextLoader or any other DataLoader the developer implements. The DataLoader itself uses the ***Template*** Pattern as well as a ***Factory*** pattern this allows for future extensibility as the future developer can simply extend the DataLoader and implement the way the data is loaded in from the new data source i.e a web-based loader and also allows for easy use by the developer as rather than constructing the associated data loader they can simply call the factory method to figure it out for them, future increasing future extensibility as the future dev will not need to worry about which data loader to use for each data source. This unfortunately also increased coupling as the Shop is now coupled to DataLoader, but I still went through with this implementation as it made future extensibility. The Shop has high cohesion as it has a clearly defined goal of being an in-game shop, it only accomplishes that purpose which means it has one clear goal, as does each DataLoader which is simply to load Data. The DataLoaders in the future will also have high cohesion as a strategy pattern means if following good practices, you have one clear goal of loading data.

**Battle:** Battles in the game use a Battle model, this model uses the ***Observer*** model I chose this because it makes it easy to update the view each turn, rather than passing data to control then to view can just notify observers to update. This increases coupling because now Battles knows about BattleObservers, however, it increases future extensibility as now any form of a view can be swapped in by simply implementing the BattleObserver i.e in future you may want 2 or 5 views which you can do by adding it as an Observer. Battle has a clear purpose of being the data representation of a battle in the game as such it has high cohesion. Battle is also coupled to Characters as it has two participants, this is another flaw with my battle implementation and unfortunately I do not have the time to reimplement how battles would work to not have it so coupled.

Alternative Design Choices

**Enchantments:** A possible flaw in my design that enchantment is not an item, I decided this because logically I didn’t want to make an enchantment an item, because it's not, however, the same can be said about an enchantment being a weapon, I perhaps could have used a template pattern for items rather than a strategy pattern, and made equippable a subclass of the item, then made enchantment, weapon, and armour a subclass of equippable. This would be in the future that an enchantment could also be used for armour easily as it is equippable so you can just make a defensive enchant and consume the armour rather than the weapon.

**Shop:** Another flaw with my design is the fact that Shop is coupled to data loaders, rather than doing this another design choice you could do is to instead have the lists fed into the shop and just give the returns from the DataLoader to the shop, this would mean the shop is not coupled to the data loaders, however, the reason I didn’t implement it this is way is that this is harder for the developer to use, I think that a developer would prefer to simply give the shop the data source in a factory method so the factory can sort out which data loader to the user, then the data loader can sort out giving the returns.

**Items/Characters:** A alternative to the template pattern is the strategy pattern i.e using interfaces rather than an abstract. From what I have read and in my opinion, this is worse for the item/character as you would have a lot of duplicated code i.e the class fields would need to be defined multiple times for player and enemy as interfaces cannot have classfields as well as the constructors, you also cannot have common methods defined in the superclass as interfaces can only be purely abstract.

**Battle:** Rather than using the observer pattern on the battle itself, you could instead have the observer pattern used on the individual participants of the battle and have the view update itself according to the character’s data rather than the battle (reduces coupling a lot), this would mean you wouldn’t even need the battle model and could simply handle battles in the battle controller (harder to do as you also have to maintain battle statistics in the controller), however, it’s a trade-off as coupling would be greatly reduced. If I had the time I would implement it this way rather than the decision I have made.