Author: Joshua Pocsidio 18267495

Object Oriented Software Engineering COMP2003

Combat Game Application

1.0 Program Overview

1.1 Architecture

For this program, Mode-View-Controller (MVC) architecture has been implemented. This allows separation of concerns and responsibilities into three different general purpose:

- **Model**
  - Classes which store data and information that represent real-life concepts. Minor logic that are specific to the represented 'concept' are also included in model classes to ensure that it is aware of the restrictions and logics being imposed on itself.
  - Updates the view of necessary events or state changes
- **Controller**
  - Classes which holds the major logic that determines the overall decisions before executing actions dependent on the model classes. Such classes also hold substantial amount of validity checks so that model classes can focus on logic that outside classes cannot handle.
  - Modifies and manipulates the data contained inside the model classes
- **View**
  - Classes which interfaces between the program and the users of the program. Such classes handle and checks the integrity of incoming input from users before passing the information to the controllers. These classes also read and decide which information will be displayed or be available to the user. It is important to ensure that views are not given the authority to modify or manipulate information stored in model classes to maintain the separation of concerns.
  - Passes user input for controller to handle and listens to necessary state change(s) from the model

This architecture allows the program to maintain **cohesion**, **low coupling**, and clear **separation of responsibilities**. However, having the architecture itself does not provide these benefits alone and must be implemented with carefully devised design patterns. Such design patterns will be discussed in this report.

Author: Joshua Pocsidio 18267495

2.0 Design Patterns

2.1 Strategy Pattern – Item Database Loader

Part of the future extensibility design propositions, it should be made easy to add alternate soures of shop item data in the future. To allow this game to scale accordingly, Strategy classes are used. Each strategy class implementing *ItemDatabaseLaoder* will have be enforced to implement a method called 'load()' which returns a list of *GameItem* objects that will be passed into the *ItemDatabaseManager*.

Benefit(s)

- If an alternate source of shop item data are to be added, simply create a Strategy class which implements the interface *ItemDatabaseLoader* **–** improving overall **extensibility** since *ItemDatabaseManager* is not required to be remodified despite new source of data.

UML Diagrams.pdf

README.txt


Combat Game - Joshua Pocsidio 182

2.2 Template Pattern – Combat Player

For the parent class, two *Template Methods* are defined: (1) attack and (2) defend methods. These are methods which can be executed by any type of player within the game. However, *CharacterPlayer* and EnemyPlayer are subclasses which both use such methods but have different logic behind the calculation of damage and defence. Each method is explained below.

(1) The attack method on the other hand, was originally not to be implemented as a template method. However, both CharacterPlayer and EnemyPlayer had to act as subjects for *AttackObservers.* By implementing this method as a template method, both subclasses do not have to declare and implement *addAttackObservers, removeAttackObservers* and *notifyAttackObservers*, and therefore prevent duplication of code.

(2) The defend method is implemented as a template method due to the common algorithm of calculating the final damage from an attack due to defence and the logic to prevent passing negative damage when updating the health of the player object (if defence is calculated greater than the damage). Part of the common algorithm is the call to notify *DefendListeners*. Like the attack method, this provides the benefit of reusing code between sharing subclasses.

Benefit(s)

- If more observers are wished to be attached to both attack and defend events, subject methods only need to be added onto this parent class instead of both, improving overall **code reuse**.
- If more subclasses are to extend from *CombatPlayer* class, the common algorithm and logic are already implemented. Likewise, this improves overall **code reuse** and indirectly eases the extensibility as future subclasses only need to focus on the new concepts.

Author: Joshua Pocsidio 18267495

2.3 Decorator Pattern - Weapon Item & Weapon Enchantments

In this game, weapon items can be upgraded by modifications which provides additional effects depending on the type of enchantment used. Weapons in this game can only deal damage and therefore enchantments are implemented to only have an effect on modifying the damage that can be dealt by that weapon. Since any given weapon can be enchanted with zero to many enchantments, the decorator pattern is used. In this design, any *WeaponItems* and subclasses inheriting from it can be given such modifications. Currently, only one type of weapon is implemented as specified but any future types of weapon subclass can be implemented easily with the option of being modified by already existing enchantments.

Benefit(s)

- Modifications to a weapon (or enchantments) are decoupled from the *WeaponItem* class. Instead of implementing subclasses to define such modifications for any given item, the **responsibility of adding modifications is separated** from the *WeaponItem* class which should coherently focus on defining what a weapon is. This also leads to better **extensibility** as creating new modifications only require creating a new subclass to extend from a *WeaponEnchantment* class and does not require any further modifications in the *WeaponItem* class.
- By implementing decorator pattern, there is no need to implement an excessive number of subclasses to implement all the combinations of modifications available – **code reuse**. This also improves **maintainability** since if a modification is required to a certain type of enchantment, then it is only required to modify the specific enchantment as opposed to modifying all of the combinations involving that specific enchantment.
- Enchantments to be added onto a weapon is determined during runtime and therefore using decorator wrappers allow for dynamic creation of modified weapons without affecting or directly accessing the *WeaponItem* object.

NOTE: It is important to note that a subclass Weapon is implemented to allow for this *WeaponEnchantment* class to reap the full benefits of the decorator pattern. There is a compromise of lack of implementation within Weapon class but this allows for other types of *Weapons* that can be added into the game. For example, the *weaponType* field can be used to create different types of weapons that have different damage calculation algorithm. If done so, the only modification required from the *WeaponItem* class is removing the *weaponType* field. After that, simply creating new subclasses inheriting from *WeaponItem* will allow as much types of weapons that can be decorated (or enchanted in this context).

2.4 Composite Pattern - Menu Interface

A hierarchical menu system is present in the specifications of this game. Without any thought, the user interface or views can be implemented in one huge class (or a god class). However, this raises numerous code smells and therefore a Composite Pattern is applied. In this design:

- Node Interface: *MenuInterface*
- Composite Class: *MenuDirectory*
- Leaf Class: *MenuAction*

MenuDirectory is essentially a menu that contains other menu directories or actions. Such type of menu interface only handles navigation between different parts of the menu hierarchy. In this program, these are the *MainMenuView* and *ShopMenuView*.

MenuAction is the class which does not handle such navigation functionality but contains more user extensive input handling and control of flow of events. A good example of this is *BattleView* which does contain any sub-menus but have to be able to handle the complexity of flow of battle mechanics. Other simple examples that do not require complex logics are the *ChangeWeaponView*, *ChangeArmourView*, and *ChangeNameView*.

Benefit(s)

- Prevents one god class which contains all the user input related interactions and therefore improving overall **cohesion** by separating the responsibilities for each view of shop, battle, selling, etc.
- This **decouples** different views or user interfaces between one another as they can all exist independently.
- All the code for validity checks for user inputs are **reused** and allows subclasses to focus on implementing the logic for the flow of events within the segment of program being implemented.
- If the hierarchy of the menu requires to be modified by either removing, displacing, or adding new menu interfaces, then it is only required to modify the controller or factory class responsible for initialising the hierarchy. This improves the overall **extensibility** of the program as the code is implemented in a way that it is not restricted to the current hierarchical structure of menu interfaces.

NOTE: It is important to note that subclasses inheriting from *MenuAction* does not necessarily mean that such subclasses cannot navigate through a different set of option list. Some subclasses may choose to extend *MenuAction* class despite having multiple sets of option list. Depending on the nature of the implementation, some *MenuAction* subclasses may require holding state variable, field variable, and/or model object(s) throughout the inner hierarchy of options. A perfect example of this is the *BattleView* which requires to hold the *CharacterPlayer* and *EnemyPlayer* throughout the whole battle until the battle and/or game is over. Implementing this view class to be a *MenuDirectory* may cause complexities in passing such objects around, in and out of menus.

2.5 Observer Pattern – Battle View & Main Menu View with Observers

During battle, events such as defending, attacking, healing, etc. are required to be outputted to the user's view to notify the effects of each action/decision they take. To prevent the model classes being coupled to each view class required to react to such events, the Observer Pattern is implemented. The view(s) should only output such events if a user input or action passes through all error and validity checks through the controllers. Therefore, it may not be appropriate for the view classes to decide when an output must be displayed.

Benefits(s)

- **Decreases coupling** between the model and view classes such as between *CombatPlayer* class and the *BattleView* class. In this pattern, *CombatPlayer* does not necessarily know about the existence of *BattleView*. Furthermore, *CombatPlayer* does not require specific observers for it to exist.
- **Improves extensibility** since any additional observers wishing to listen to such events simply needs to implement the observer interface and does not require modifying the subject classes to directly communicate with new observers. Similar benefit is apparent when removing observers.

2.6 Factory Pattern - Enchantment, Item, Enemy factories

In this game, there are many objects that can only be determined and created during run time – enchantments, items, and enemies. Enemies are spawned randomly, items are created based on a parsed data source, and enchantments are wrapped based on user inputs. To allow for dynamic creation of such objects, factory pattern is implemented.

Benefit(s)

- Encapsulates all the decision making and required checks to determine which subclass to create – **improving cohesion** since all the responsibility for dynamic object creation are focused into their respective factory classes.
- This isolates creation of objects and/or call of constructors away from other classes and therefore reducing the overall **dependency** between classes. This further improves the overall **testability** of model classes since the decision making of object creation of external classes is now within the responsibility of the factories.
- By reducing the dependency between model classes, any modifications done on classes will only modify the way it is created and passed from the injector without modifying other model classes – ultimately improving **maintainability**.

2.7 Builder Pattern – User Interface

This pattern is not covered within the scope of the unit but is a pattern which proved useful when constructing the views displayed to the user. This class implements and defines a common template used throughout the whole program in majority of the views. This provides the flexibility of optional components of the view (e.g. pre-heading, heading, body, footer, etc). Calling the code only requires to add extra functions when instantiating the object before calling the 'build()' function.

Benefit(s)

- The template does not need to be re-defined or re-implemented therefore improving **code-reuse**
- This improves **cohesion** as most of the menu subclasses only need to focus on flow of events logic without being concerned of the constructing a repeating template.
- Overall **maintainability** is improved as any changes required for the display structure only need to be modified within the *UserInterface* class instead of modifying within each menu subclasses.

3.0 Alternate Designs

3.1 Dynamic Item Factory Design

Current Design

Although the current implementation already uses a factory class to create *GameItem* objects, these objects are all created as soon as they are parsed from an external data source. These are then placed into *ItemDatabase* which contains a list of items that are unique from every other item within the database. The following are the reasons for this usage:

- It is not desirable to have invalid *GameItem* objects within the database. Having the *ItemFactory* and *GameItem* classes validate the item as they enter the database is advantageous since the users will not be able to view invalid objects when displayed to them.
- The point above is possible even without validating as data is parsed in by having another layer of validation in the IO file. However, *ItemFactory* and the *GameItem* classes have validity checks specific for item creation, and therefore allows for such responsibility to be separated from anywhere else in the program.

Alternate Design

A different approach to this problem is using the *ItemFactory* class to dynamically create the item objects only when required. To do so, implementing the *ItemDatabase* to contain string representations of each type instead of validated *GameItem* objects will cause the following changes:

- *GameItem* objects will only exist in the program when required. While not required, they exist simply as separated fields of string (e.g. name, cost, minEffect, maxEffect, attributes).

- *ItemFactory* will not be used by *ItemDatabaseManager* but instead by a *ShopController* so that items can be created dynamically

3.2 Strategy Method Pattern – Combat Player

Current Design

In this current game implementation, players can only simply be a *CombatPlayer* and therefore Template method pattern was used. All types of players inherit from the same common abstract parent class as they all have the same common fields.

Alternate Design

A different approach to this problem is implementing a system of strategy interfaces such as *CombatPlayer* and *SalesPlayer*. An abstract *GamePlayer* class will be created but without the abilities to perform battle actions. This will allow players to exist within the game that does not require to fight. This implementation will cause the following changes:

- The whole representation of a shop will be replaced by a *SalesPlayer* that has the ability to buy, sell, and enchant.
- The current *CharacterPlayer* classes will implement *SalesPlayer* to allow transactions with other *SalesPlayer* objects. For example, if a character sells an item, the other sales player (e.g. a shop) will have to call buy what the character sells.
- This prevents all types of players to be restrictedly combat players and allow for flexibility in the future what types of players each player can be.