# Coursework Report for COMP62524

1. **URL of the source code of this project:**

https://github.com/RoyalOakShield/level\_up\_game

1. **Brief description of application system and the main functional features**

Our application system is a level up game that can run on the browser.

Its main functional features are:

1. User login and registration.

2. Create player. We have 4 branches. Each branch has different ability value, such as HP, AFK, DEF, SPD.

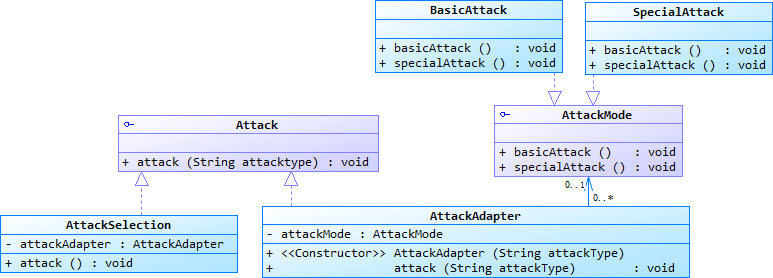
3. The player fights the enemy, enemy is divide into Zombie, Ghost and other characters.

4. Player can level up and get items by winning in fighting.

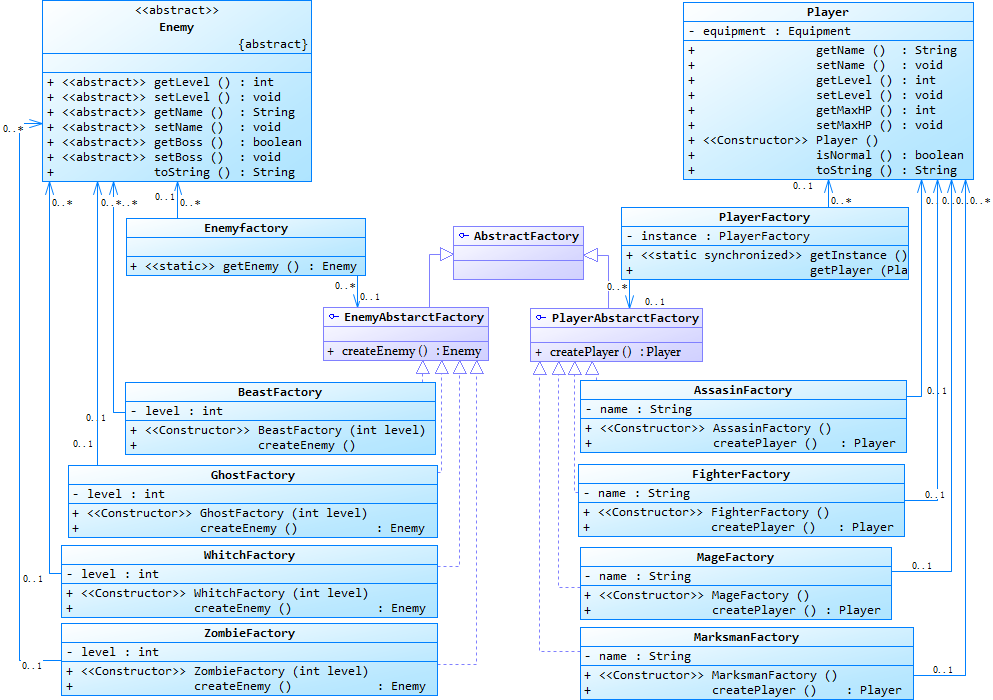
5. Items are mainly divided into potions and equipment, potion can direct use to increase the ability value, equipment is divided into shoes, armors and other items. Each part can only be equipped with one equipment.

6. User data and player data will be stored in MySQL database.

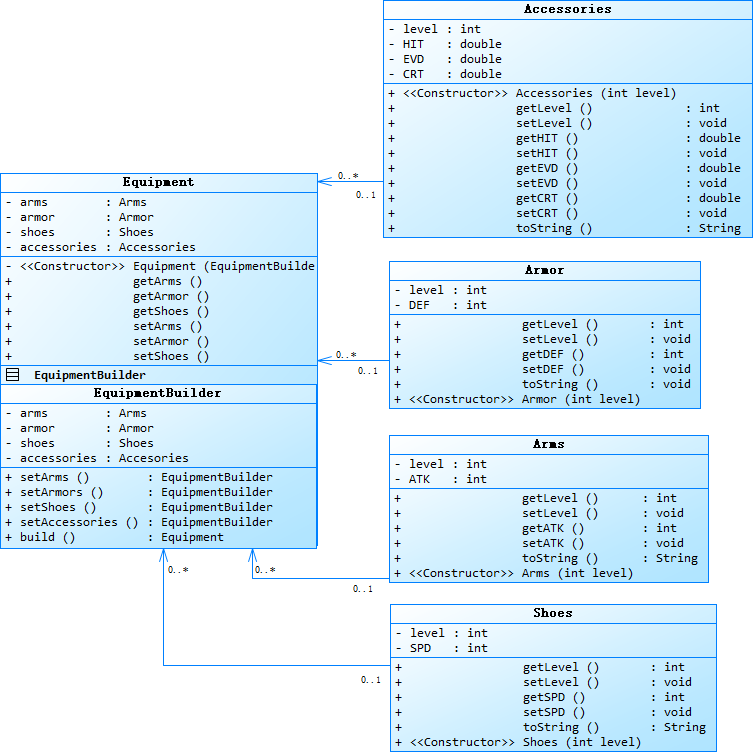
1. **UML Class Diagrams**
   1. **Adapteer Pattern**

****

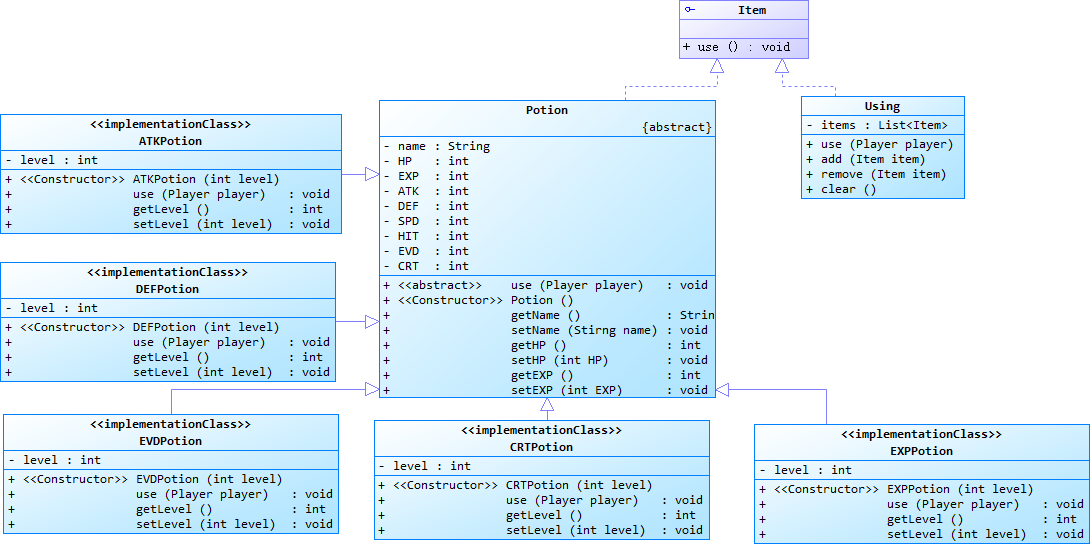
* 1. **Abstract Factory Pattern and Singleton Pattern**

****

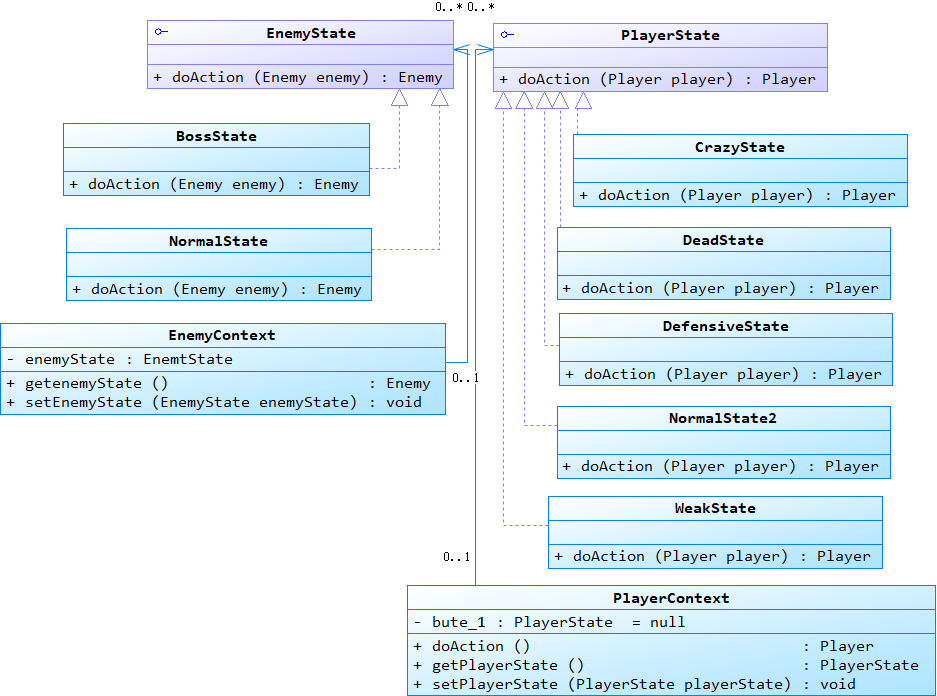
* 1. **Builder Pattern**

****

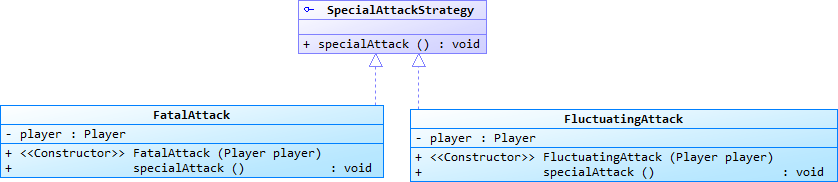
* 1. **Composite Pattern**

****

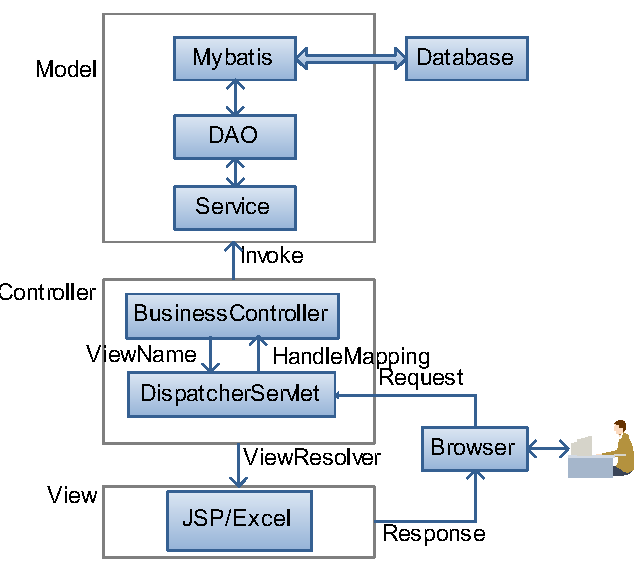
* 1. **State Pattern**

****

* 1. **Strategy Pattern**

****

* 1. **Model-View-Controller Pattern**

****

1. **The problem each pattern aim to solve and why the pattern is needed and if there is a better solution without using the pattern.**
   1. **Abstract Factory Pattern**

This pattern aims to reduce code coupling where the implementation of objects and the real use of them are coupled together, scilicet the requirement of the initial value, the selection of class type and the auxiliary classes in instantiation phase are separated with the operation execution and variable use of objects

This pattern satisfies the Dependency Inversion Principle and Liskov Substitution Principle by providing interfaces or abstract classes for building a family of similar objects without specifying the actctual class. Clients could use the operations defined in the interface or abstract class to do the real job by creating objects through the abstract factory instead of do it themselves. This method also takes advantage of the polymorphism of OOP.

Presently, other solutions to create a family of objects would possibly be less hierarchical explicit than the abstract factory Pattern. Because the creation operations would either be embedded in the working class or be done in other separated classes. In both cases, the coupling of codes may rises significantly.

* 1. **Singleton Pattern**

The singleton pattern is mainly used for access control of certain objects, which means the client could only access that objects in an order. Furthermore, this pattern can save the resources of system. Memory is saved because there is only one instance for a class. Time consumption is reduced because the creation and destruction phase is reduced.

On the other hand, singleton Pattern violate the Single Responsible Principle because the singleton class mix the creation and the use of the class together. Also, the project in the present stage only allows limited user access. Therefore it is not quite necessary and appropriate to use the singleton pattern in the project.

* 1. **Adapter Pattern**

The adapter pattern is mainly used for converting one certain interface to another interface that the client wants, which is the case in the project. Also, the reusability is better at the same time. Instead of creating a new class, the existing functions could be used again simply by changing the interfaces. Furthermore, the scalability also increased through the extension of class usage. This pattern comply the Open Close Principle as well.

Unfortunately, this pattern is probably the best way to implement our small application. Although too many adapters may cause chaos and not easy to grasp in a larger system.

* 1. **Builder Pattern**

The builder pattern is used for constructing complex objects step by step. A complex object could be created by delicate control flow without the awareness of the final user. The step of construction of complex objects can be easily replaced to create a class with other features. This pattern have things in common with the factory pattern: they all hide the construction details to the user so the user do not need to know the details of constructing a class. But builder pattern allows multiple steps for constructing and the alteration of attributes while factory pattern have only one step for construction and cannot change the attributes of its products.

The builder pattern is essential to the project because the equipment class needed in the project is complicated. And in order to change the attributes of each equipment and make it easy to add equipment later on, it is the best choice. In other ways, the creation of such classes would be disordered and messy.

* 1. **Composite Pattern**

Composite Pattern treats a group of similar objects as one object and use a tree diagram to illustrate the composition relationship between objects. In many cases, it is common to overlook the differences between a single object and a combination of objects. With the hierarchy of the composition objects, higher layer or client classes can ignore the divergence and easy to control the whole thing. At the same time, the ignorance of the single class and composition would lead to simpler higher level code. Also, it is convenient to add leaf or composition to the whole structure, which comply the Open Close Principle

On contrary, it is not always promising to have a tree structure when the whole layer goes too deep or too many leaves adds up the width, which increase the complexity of the usage. Accordingly, it is likely to be a better practice to constrain the width and depth of the tree. If the tree goes too deep or wide, then split it to another tree.

* 1. **State Pattern**

The state pattern realize cascaded if-else structure by polymorphism. This pattern could avoid the error-prone code of traditional selection sentences by making each condition branch into a state class. This method could reduce the code coupling through separating each state from the same class and representing each state with a single class. The maintainability of the whole system would be benefited by this pattern because the modification of different states only affects that certain state class, not the whole selection logic.

Unfortunately, this pattern increases the amount of classes in the system. Therefore, it is probably not reasonable to use state pattern when the selection logic is not complicated. But in our project, there are a number of states that could be owned by the character. So use traditional selection structure would be unwise.

* 1. **Strategy Pattern**

The strategy pattern makes the whole structure clear and simple. This pattern is needed for separating the invocation of each algorithm. Also, the separation makes the data much more secured, which means different kind of data is unlikely to influence each other in a certain execution of the algorithm. Unlike state pattern, the behavior of strategy pattern is replaceable, which increase the versatility of the system.

The strategy pattern is appropriate for the situation where client need to make choices between different algorithms to accomplish certain goals. There is probably no better solution for this situation.

* 1. **Model-View-Controller Pattern.**

The MVC Pattern separates the whole system into three parts. This separation could reduce the coupling code. Also, modification on each parts would not affect the whole system. Each part remain independent. Furthermore, this pattern could make developers concentrate on function of each single layer. Although this pattern may add some complexity to the system, it is probably the best practice to solve most of the construction problems.

1. **Brief description of how we used different patterns together and their relationships**

The patterns are largely independent of each other.

We added the singleton pattern to the abstract factory pattern to implement the invocation of single instance.

Adapter pattern and strategy pattern implement the classification,selection,and use of player attack together,the adaptor pattern is mainly responsible for adapting attacks into basic attacks and special attacks,the strategy pattern divides the special attack into different attack modes and calls different special attack according to the different strategy.

The main purpose of composite pattern to use multiple items in bulk.Although equipment also belong to an item,each parts of the equipment can be assembled freely,so we introduced the builder pattern into the equipment.

State pattern is a completely separate pattern,which is only used to make corresponding actions according to the different states of players and enemies.

1. **Brief description of the implementation of each pattern**
   1. **Abstract Factory Design Pattern**

We will create abstract classes for player and enemy and entity classes that implement these interfaces. The next step is to create the abstract factory class AbstractFactory. Then define the factory classes PlayerFactory and EnemyFactory, both of which extend the AbstractFactory. Then create a sub-factor class, such as fighterFactory, beastFactory.

* 1. **Singleton Design Pattern**

The singleton pattern is inserted in the PlayerFactory which is designed to create a unique instance for the player.

* 1. **Builder Design Pattern**

we create a static nested class EquipmentBuilder and then copy all the arguments from the outer class to the Builder class.

and then we set the parameters which should return the same Builder object after setting the attribute.

The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

* 1. **Adapter Design Pattern**

We have an Attack interface and an entity class AttackSelection that implements the Attack interface.  
We also have another interface, AttackMode , and the entity classes BasicAttack and SpecialAttack that implement the AttackMode interface.  
We want to let AttackSelection choose different modes of attack. In order to implement this function, we need to create an adapter class AttackAdapter that implements the Attack interface, and use the AttackMode object to attack in different ways.

* 1. **Composite Design Pattern**

Composite pattern base component defines the common methods for leaf and composites. We can create an item class with a method use(Player player) .

Composite design pattern leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as potion，equipment and their sub class.

A composite object using contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

* 1. **State Design Pattern**

We will create a State interface for player and enemy and an entity state class that implements the State interface. Context is a class with a certain state.

* 1. **Strategy Design Pattern**

We will create a specialAttackStrategy interface that defines the activity and entity classes fatalAttack and FluctuatingAttack that implements the Strategy interface.