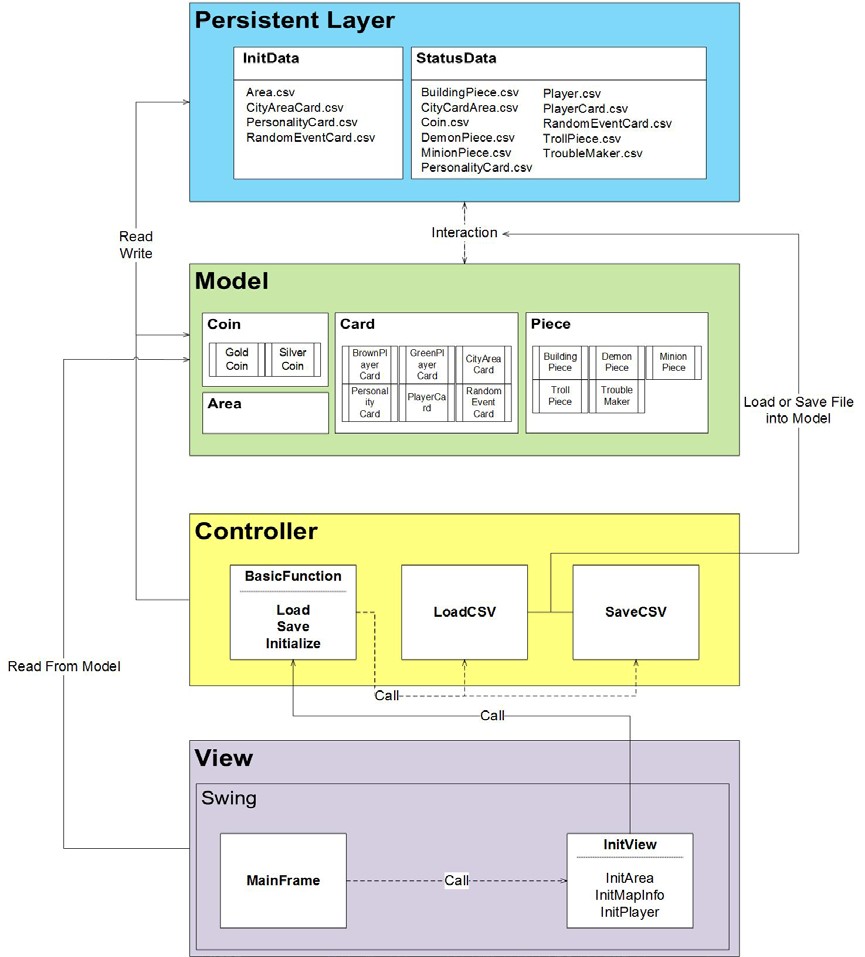
**Software Architecture**

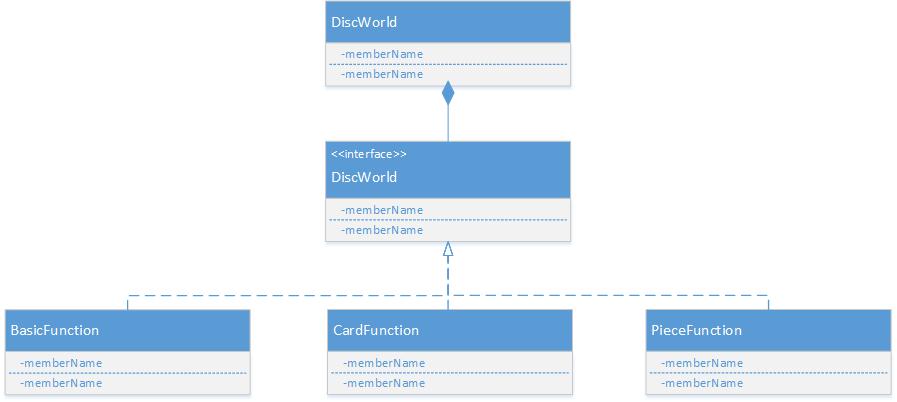
**SOEN 6441 Advanced Programming Practice**

**Project Build 3**

**MVC**



**Strategy Pattern**



By using strategy pattern, we encapsulate our main function or algorithm in single class so that we can use them independently from the whole system. Even something is wrong in some algorithm we do not need to modify other parts of the system but only that class file which contains the algorithm. The diagram above shows the architecture of strategy pattern. The DiscWorld on the top is the client class that will use these algorithms. In the middle, it provides an interface to access the concrete class that implements the interface. To access these algorithms we do not need to concern about the detail but just parameters and return value. The UML below is the list of five main classes that we use in our project:



**BasicFunction**:

Defines basic Function that the system will use including saving and loading file, shuffling player and random cards from deck and so on. The most important role that it does is that it is the port we access the DiscWorld instance, and from this instance we can access the list of hash table of players, cards, areas and all other factors in Disc World.

**CardFunction**:

This Class provides operations on all kinds of card, including grabbing and playing card, grabbing random event cards.

**MoneyFunction**:

This Class provides operations on Money flow. In Disc World, we can trade between each player and bank. According to the rule of game, we need to get and send money between each player with arbitrary amount, in addition with bank.

**playerGetCard**:

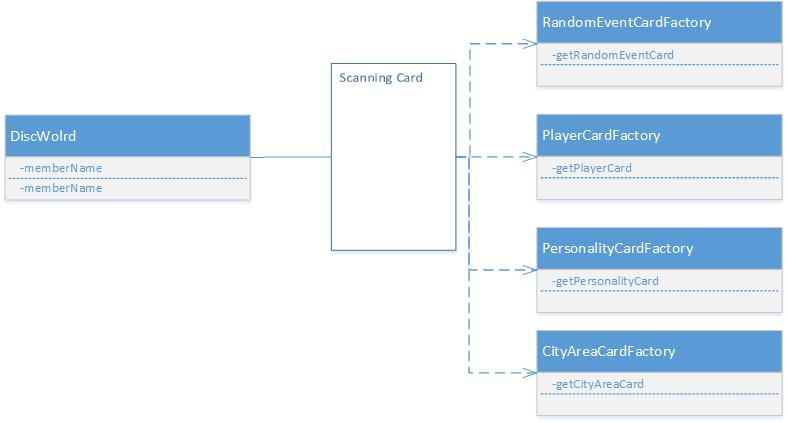
This Class provides operations between player and card. One player has access to card on top of the card deck and is able to show his card every turn of the game.

**PieceFunction**:

This Class provides the main operations on five main factors in this game: minion, troll, demon, building and trouble marker. Because of the rule, each part has relationship with some other parts. For example, one player can only add minion in some areas that has its minion or the adjacent area has its minion. Encapsulation makes it easy to use because we do not need to concern about the logic in UI part.

**Factory Pattern**

We need factory pattern to provide some access point to the factors of the game independently whenever we want. Factory Pattern provides interface to access these data without need to instantiating new instance but directly using the function inside any classes. The major use of factory in this system is that we need to scan the card at any time of the game. To implement that, we need some prior created instances that can use its functions to search the cards deck.



**Singleton Pattern**

There is only one single instance of DiscWorld in this project. The reason is that we want to synchronize our system. Sometimes we need to use factory function to scan the card which also uses DiscWorld, if we do not use singleton pattern, we may mass up this object by abusing this object. Another reason to single it is memory efficiency, as the limitation of storages.