Design Patterns For A3

Sprint 1

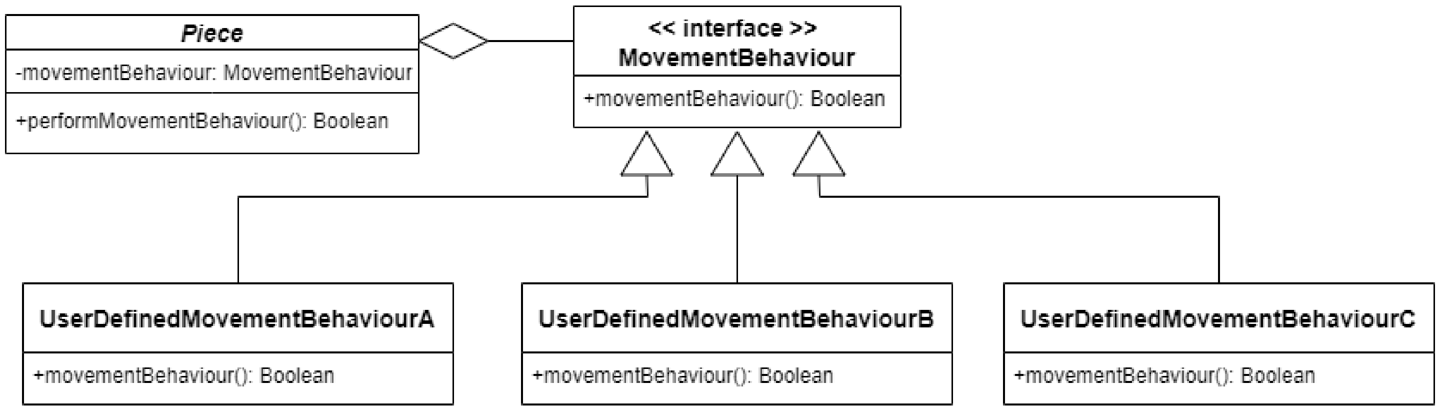
Mahmoud Zeidan
Faisal Mehmood
Harshil Patel
Aneeq Hassan

Table Of Contents

1. Strategy Pattern
    a. Description
    b. UML Diagram
2. Null Object Pattern
    a. Description
    b. UML Diagram
3. Singleton Pattern
    a. Description
    b. UML Diagram
4. Factory Pattern
    a. Description
    b. UML Diagram
5. Command Pattern
    a. Description
    b. UML Diagram
6. Visitor Pattern
    a. Description
    b. UML Diagram
7. Singleton Pattern II
    a. Description
    b. UML Diagram

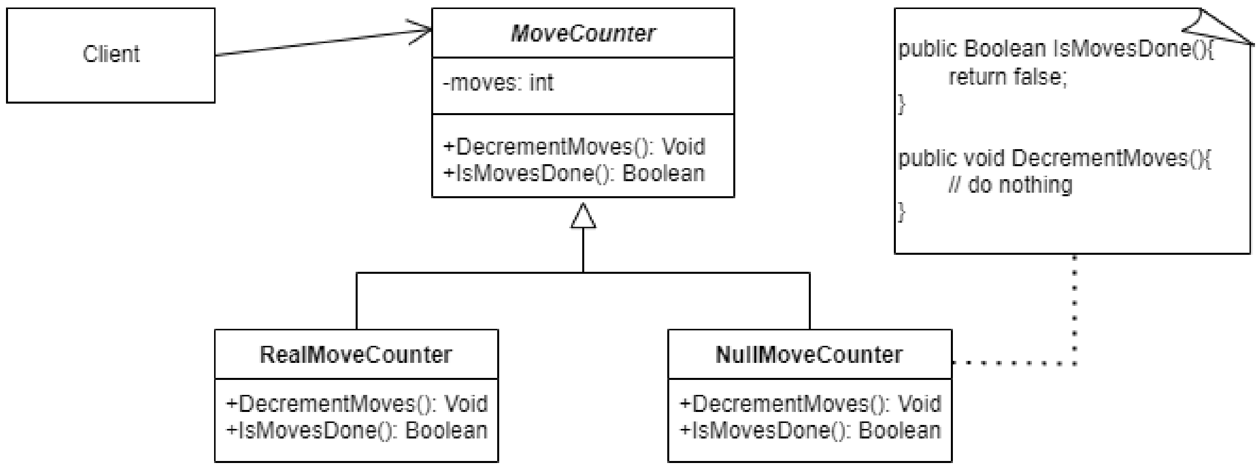
Strategy Pattern

The Strategy Pattern solves the problem of having classes that only differ by behaviour. The pattern isolates every behaviour into its own class, and allows other classes to use these behaviours and interchange them at runtime. In our case, we want the user to be able to define his own piece movement behaviours, and allow the user to choose which behaviour the pieces should use at runtime. Moreover, we want to be able to switch behaviours at any point in the game.


Piece
-movementBehaviour: MovementBehaviour
+performMovementBehaviour(): Boolean
<< interface >>
MovementBehaviour
+movementBehaviour(): Boolean
UserDefinedMovementBehaviourA
+movementBehaviour(): Boolean
UserDefinedMovementBehaviourB
+movementBehaviour(): Boolean
UserDefinedMovementBehaviourC
+movementBehaviour(): Boolean

Null Object Pattern

The Null Object Pattern solves the problem of having cases in code where there should be no functionality. It solves this by defining a placeholder object that is supposed to handle cases where no functionality should occur, or default values should be used. In our case, when the user does not pick to play with a move counter, there should be no move counter functionality and the game should have an infinite amount of moves. We would hence create a NullMoveCounter object that has unlimited moves and implements "Do nothing" code where it is needed.

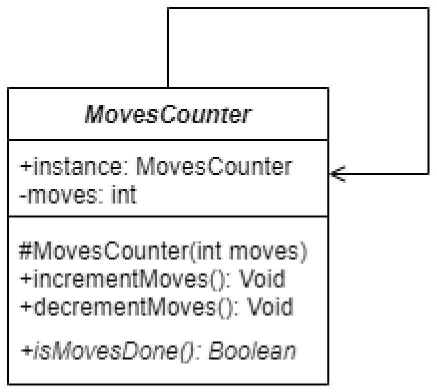


```
public Boolean IsMovesDone(){
        return false;
}

public void DecrementMoves(){
        // do nothing
}
```

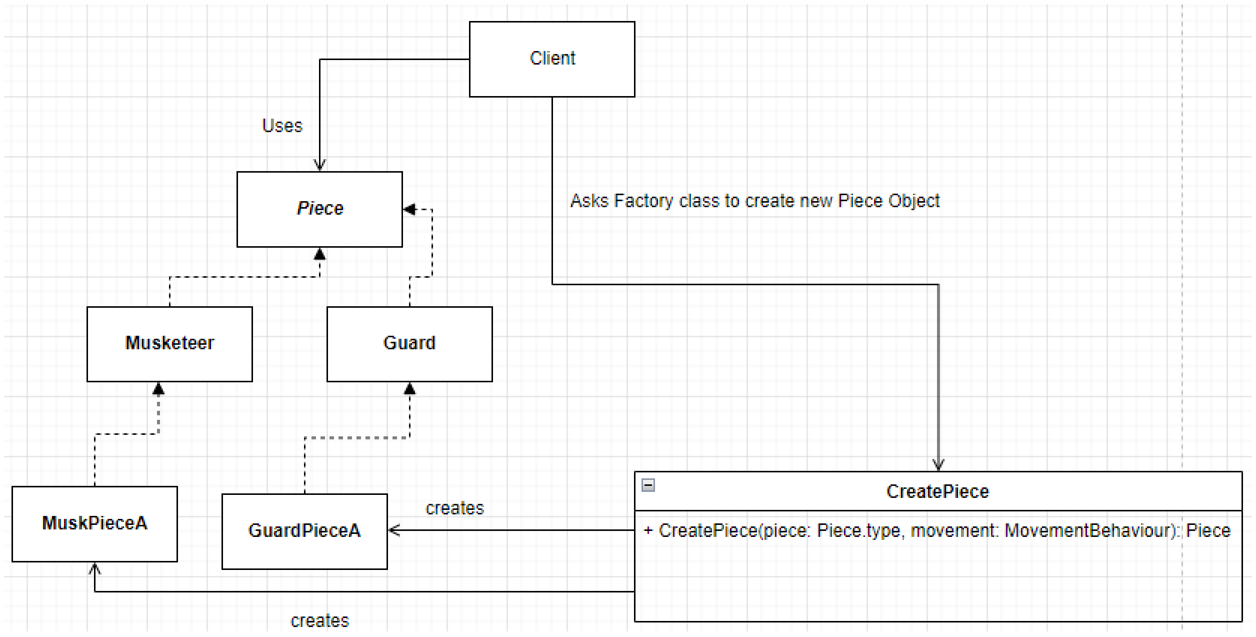
Singleton Pattern

The Singleton Pattern solves the problem where many classes in a program need to rely on one source of information or one collection of functionalities. The Singleton Pattern solves this by introducing a single instance that is accessible by the entire program, which contains all the functionality and data needed. In our case, we want various parts of the program to be able to access the MoveCounter and execute some of its methods. Hence, make MoveCounter a Singleton accessible by the entire program.



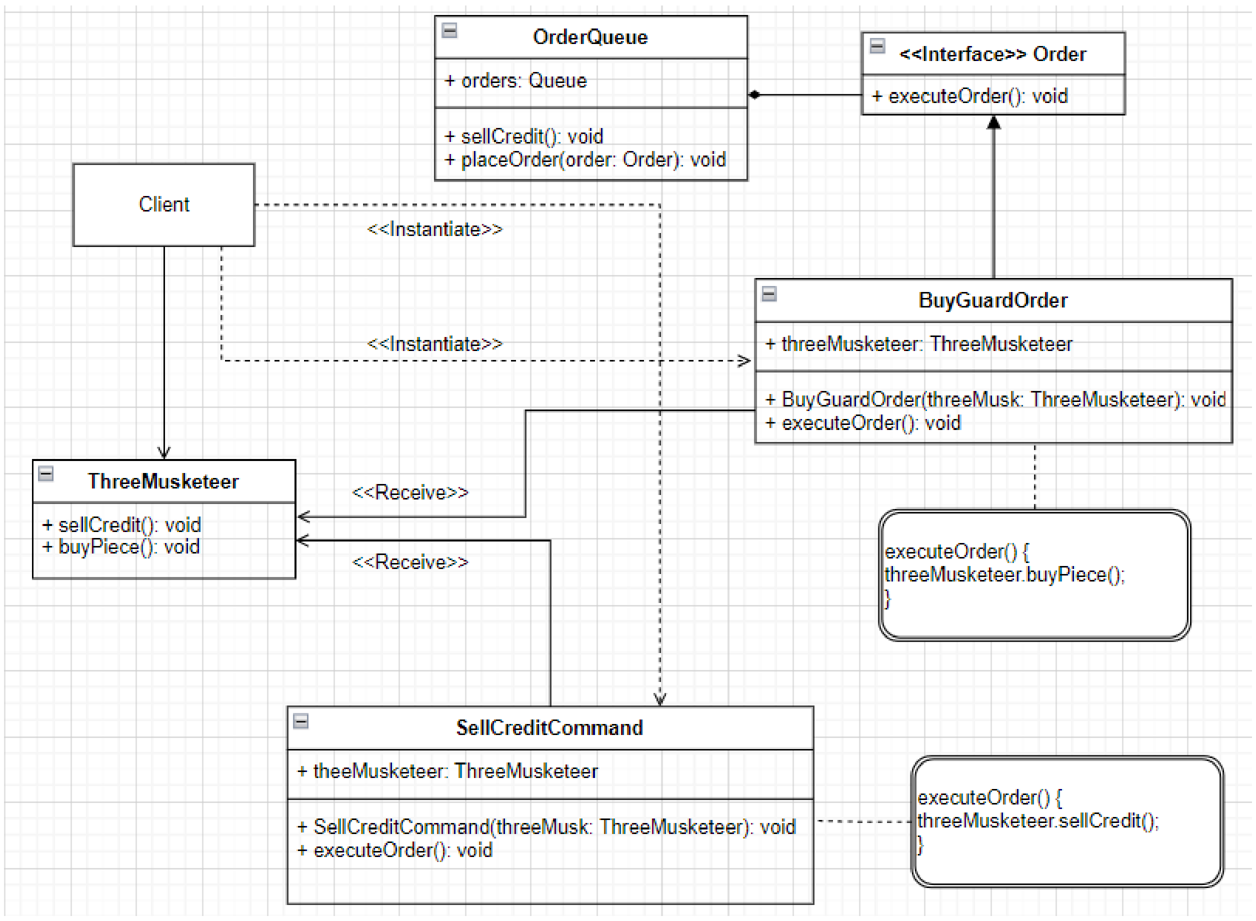
Factory Pattern

The factory pattern solves the problem where the client requests a new object, but the instantiation logic should not be exposed to the client, and the client should be able to use these objects without being aware of their concrete implementation. In our case, we want the player to be able to choose which movement behaviour they want to play with at runtime, without exposing their implementation. The user should be able to play the game exactly the same way regardless of what behaviour they choose while the controls remain the same.


Client
Uses
Piece
Asks Factory class to create new Piece Object
Musketeer
Guard
MuskPieceA
GuardPieceA
creates
CreatePiece
+ CreatePiece(piece: Piece.type, movement: MovementBehaviour): Piece
creates

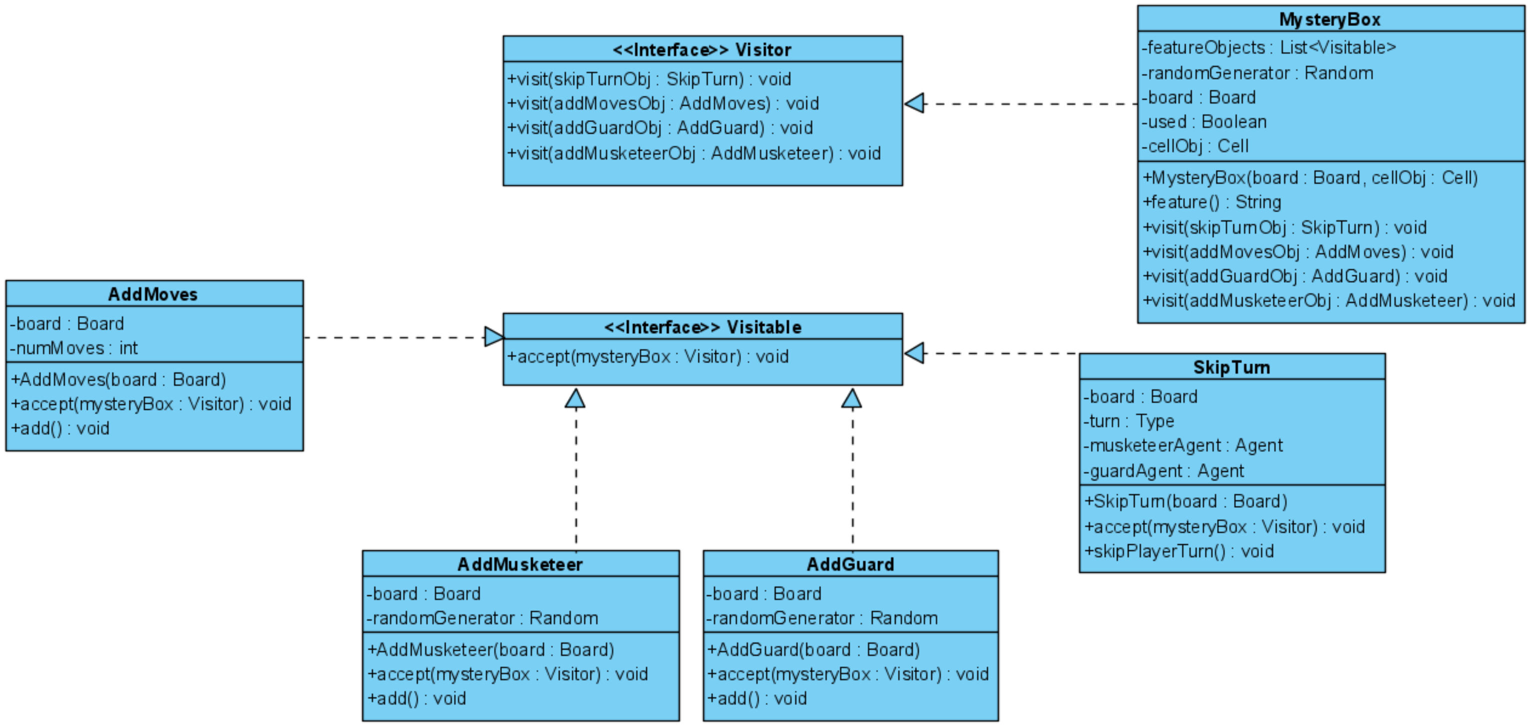
Command Pattern

The command pattern solves the problem where a series of actions are to be taken place sequentially, possibly at different intervals of time. The command pattern solves this by encapsulating an action into a Command, which is then added to a queue and executed when the time is right. In our case, the player playing as the guard has the ability to respawn dead guards. The player does this by collecting credits every time one of his guards die. When the player chooses to respawn the guards, they respawn sequentially one at a time every turn. We will encapsulate the action of respawning a guard and add it to a queue where the respawn command will be executed every turn.



Visitor Pattern

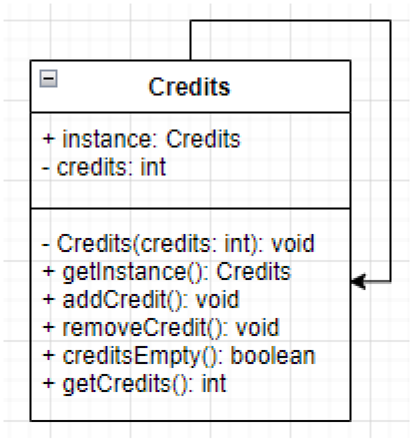
When coding, we are often interested in applying operations to all elements of a structure. However, the iterator and composite pattern both require all elements of a structure to be the same data type. The Visitor pattern solves this issue by enabling us to define an operation to be performed on every element without changing their individual type. The visitor pattern works by having two abstractions; the Visitable and Visitor interfaces/abstract classes. Every element in a collection must implement the Visitable abstraction so that it can be visited by a concrete Visitor. The Visitor abstraction declares abstract visit operations for all elements that are type Visitable. In our project, the Visitor pattern allows us to implement a Mystery Box feature. Two random cells are created at the beginning of the game with a Mystery Box that prompts an agent (Musketeer or Guard) when they land on it to open it and obtain a special feature. Special features include skipping the other player's turn, obtaining free moves if they are in a limited number of moves game mode, and to add an extra guard or musketeer (depending on the current agent). The visitor pattern allows us to create a collection of features and easily apply each one, regardless of their individual type.



<<Interface>> Visitor
+visit(skipTurnObj : SkipTurn) : void
+visit(addMovesObj : AddMoves) : void
+visit(addGuardObj : AddGuard) : void
+visit(addMusketeerObj : AddMusketeer) : void
MysteryBox
-featureObjects : List<Visitable>
-randomGenerator : Random
-board : Board
-used : Boolean
-cellObj : Cell
+MysteryBox(board : Board, cellObj : Cell)
+feature() : String
+visit(skipTurnObj : SkipTurn) : void
+visit(addMovesObj : AddMoves) : void
+visit(addGuardObj : AddGuard) : void
+visit(addMusketeerObj : AddMusketeer) : void
AddMoves
-board : Board
-numMoves : int
+AddMoves(board : Board)
+accept(mysteryBox : Visitor) : void
+add() : void
<<Interface>> Visitable
+accept(mysteryBox : Visitor) : void
SkipTurn
-board : Board
-turn : Type
-musketeerAgent : Agent
-guardAgent : Agent
+SkipTurn(board : Board)
+accept(mysteryBox : Visitor) : void
+skipPlayerTurn() : void
AddMusketeer
-board : Board
-randomGenerator : Random
+AddMusketeer(board : Board)
+accept(mysteryBox : Visitor) : void
+add() : void
AddGuard
-board : Board
-randomGenerator : Random
+AddGuard(board : Board)
+accept(mysteryBox : Visitor) : void
+add() : void

Singleton Pattern II

Once again, the Singleton Pattern resolves the issue of many classes in a program needing to rely on only one source of information. The Singleton Pattern solves this issue by establishing a class with one instance which is attainable by any class in the program. This class contains all of the data required by the other classes in the program. In our case, we want to establish a credit system which will be available to the players during gameplay, hence we want other classes in the program to have access to Credits and execute the methods it provides.


Credits
+ instance: Credits
- credits: int
- Credits(credits: int): void
+ getInstance(): Credits
+ addCredit(): void
+ removeCredit(): void
+ creditsEmpty(): boolean
+ getCredits(): int