# Team U5 - WarZone (Risk Computer Game)

## SOEN-6441: Advanced Programming Practices

Build 3

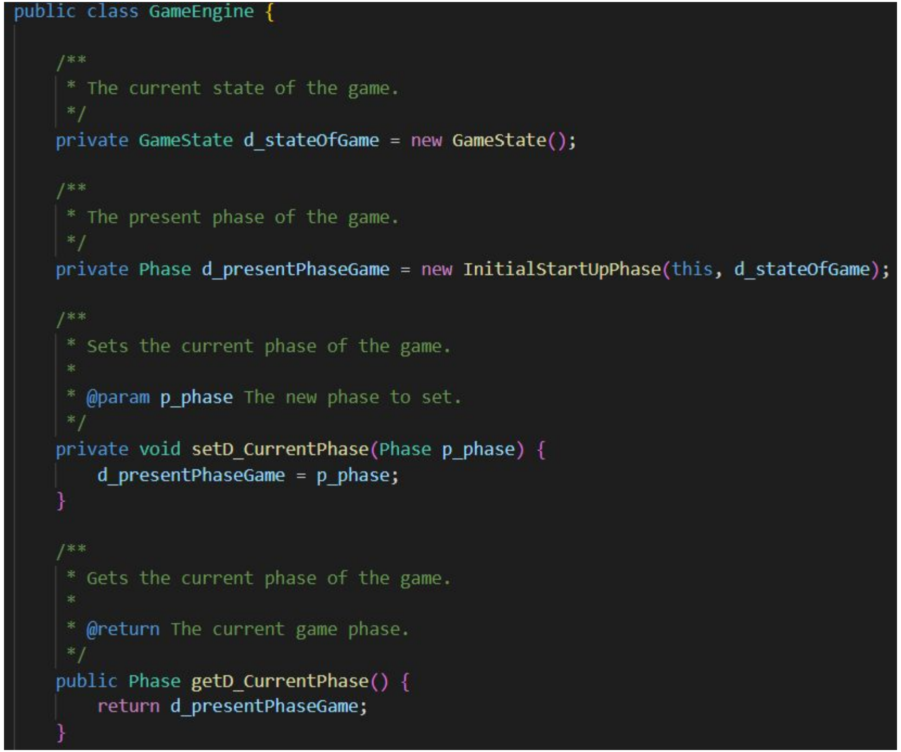
Team Members:

| | |
|---|---|
| Abhishek Handa | (40231719) |
| Rajat Sharma | (40196467) |
| Harman Singh Jolly | (40204947) |
| Amanpreet Singh | (40221947) |
| Anurag Teckchandani | (40263724) |

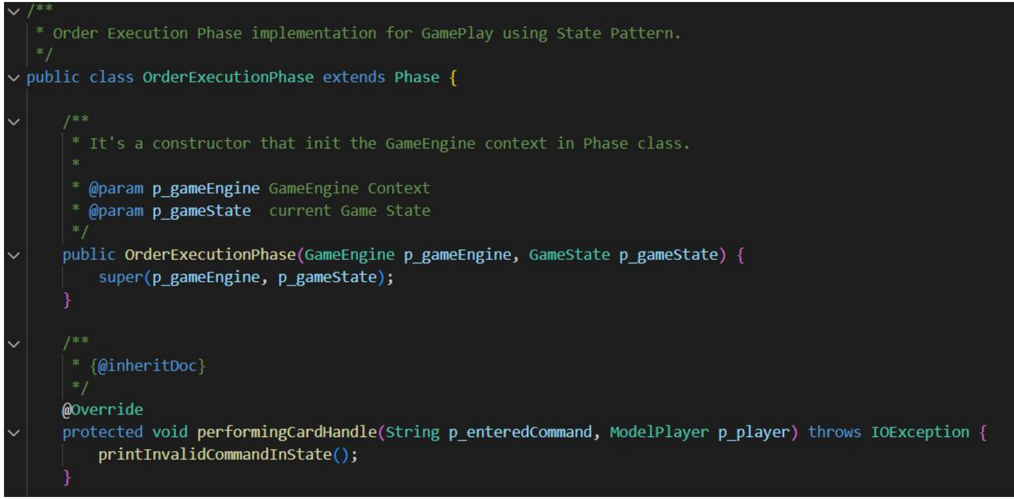
# Naming Conventions

- Data members, Member functions and Method Parameters
  - All are in lower camelCase like *int thisIsExampleFunction (int p1, int p2)*
  - And data members like *int d_gameEngine*
- Classes
  - Class names are in upper CamelCase like
  - *GameEngine.java and OrderExecutionPhase.java*
- Local Variables
  - They follow lower camelCase along with "I_." stating it is a local variable like
  - *L_reader, l_continue*
- Static Members/Constants
  - All static members are in UPPER_SNAKE_CASE letters with underscores in between the words like *int EXAMPLE_VALUE = alpha;*

# Example of Naming Conventions Used (Snapshots)



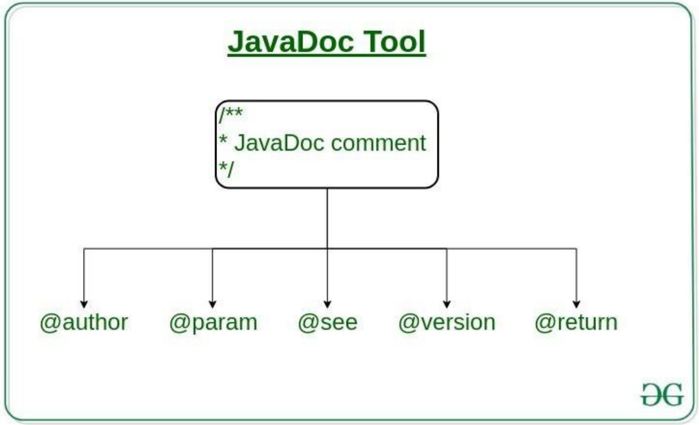
```
public class GameEngine {

    /**
     * The current state of the game.
     */
    private GameState d_stateOfGame = new GameState();

    /**
     * The present phase of the game.
     */
    private Phase d_presentPhaseGame = new InitialStartUpPhase(this, d_stateOfGame);

    /**
     * Sets the current phase of the game.
     *
     * @param p_phase The new phase to set.
     */
    private void setD_CurrentPhase(Phase p_phase) {
        d_presentPhaseGame = p_phase;
    }

    /**
     * Gets the current phase of the game.
     *
     * @return The current game phase.
     */
    public Phase getD_CurrentPhase() {
        return d_presentPhaseGame;
    }
```



```
/**
 * Order Execution Phase implementation for GamePlay using State Pattern.
 */
public class OrderExecutionPhase extends Phase {

    /**
     * It's a constructor that init the GameEngine context in Phase class.
     *
     * @param p_gameEngine GameEngine Context
     * @param p_gameState  current Game State
     */
    public OrderExecutionPhase(GameEngine p_gameEngine, GameState p_gameState) {
        super(p_gameEngine, p_gameState);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected void performingCardHandle(String p_enteredCommand, ModelPlayer p_player) throws IOException {
        printInvalidCommandInState();
    }
```

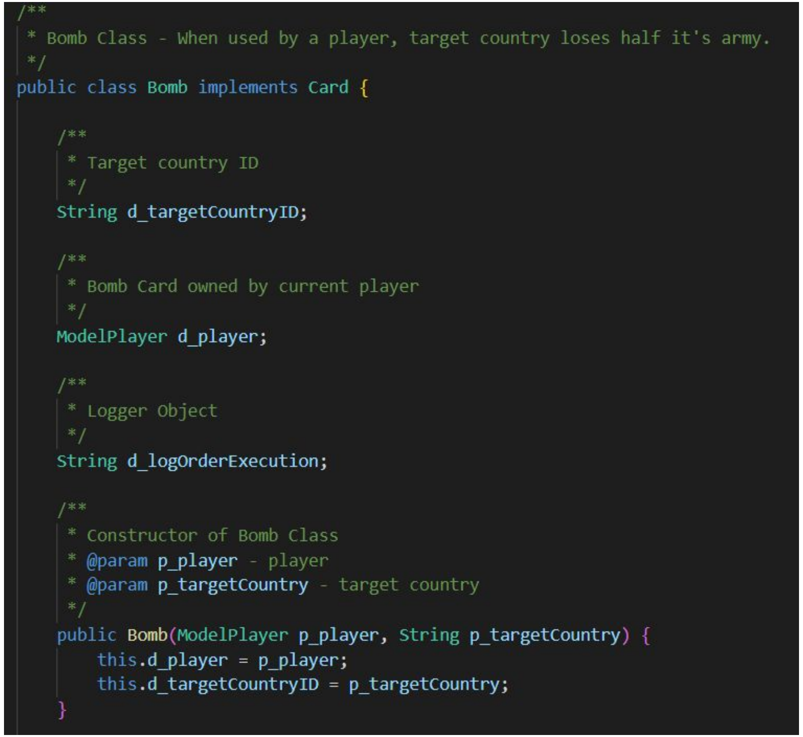
# Javadocs

JavaDoc tool is a document generator tool in Java programming language for generating standard documentation in HTML format. It generates API documentation. It parses the declarations ad documentation in a set of source file describing classes, methods, constructors, and fields.

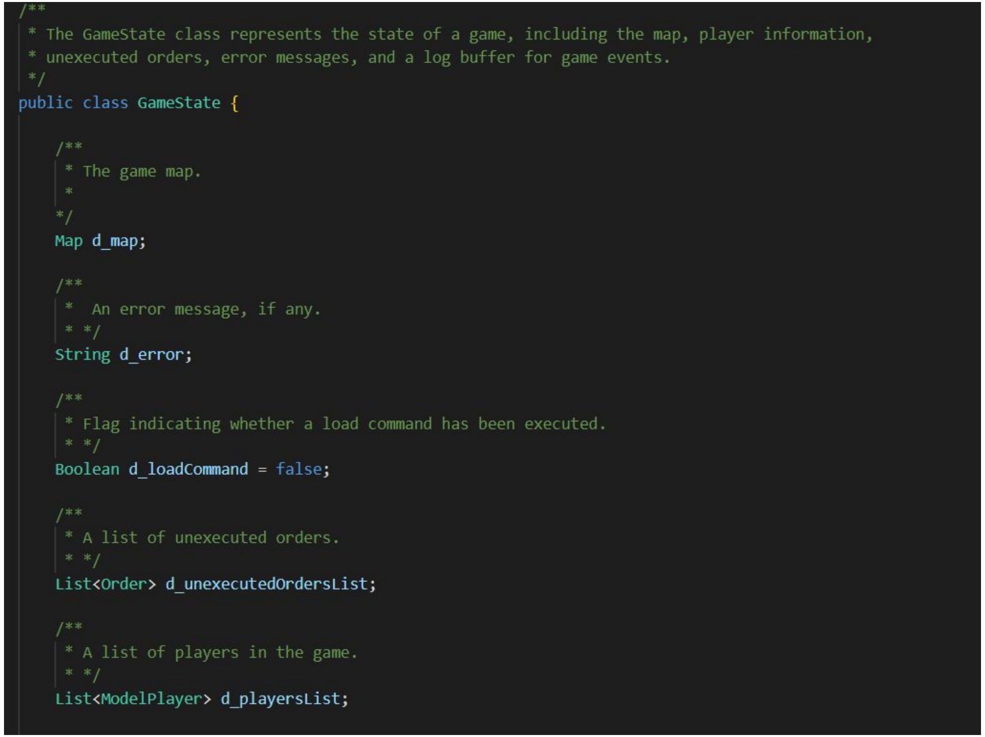


From
GeeksForGeeks

# Example of Javadocs Used (Snapshots)



```
/**
 * Bomb Class - When used by a player, target country loses half it's army.
 */
public class Bomb implements Card {

    /**
     * Target country ID
     */
    String d_targetCountryID;

    /**
     * Bomb Card owned by current player
     */
    ModelPlayer d_player;

    /**
     * Logger Object
     */
    String d_logOrderExecution;

    /**
     * Constructor of Bomb Class
     * @param p_player - player
     * @param p_targetCountry - target country
     */
    public Bomb(ModelPlayer p_player, String p_targetCountry) {
        this.d_player = p_player;
        this.d_targetCountryID = p_targetCountry;
    }
```



```
/**
 * The GameState class represents the state of a game, including the map, player information,
 * unexecuted orders, error messages, and a log buffer for game events.
 */
public class GameState {

    /**
     * The game map.
     *
    */
    Map d_map;

    /**
     *  An error message, if any.
     * */
    String d_error;

    /**
     * Flag indicating whether a load command has been executed.
     * */
    Boolean d_loadCommand = false;

    /**
     * A list of unexecuted orders.
     * */
    List<Order> d_unexecutedOrdersList;

    /**
     * A list of players in the game.
     * */
    List<ModelPlayer> d_playersList;
```

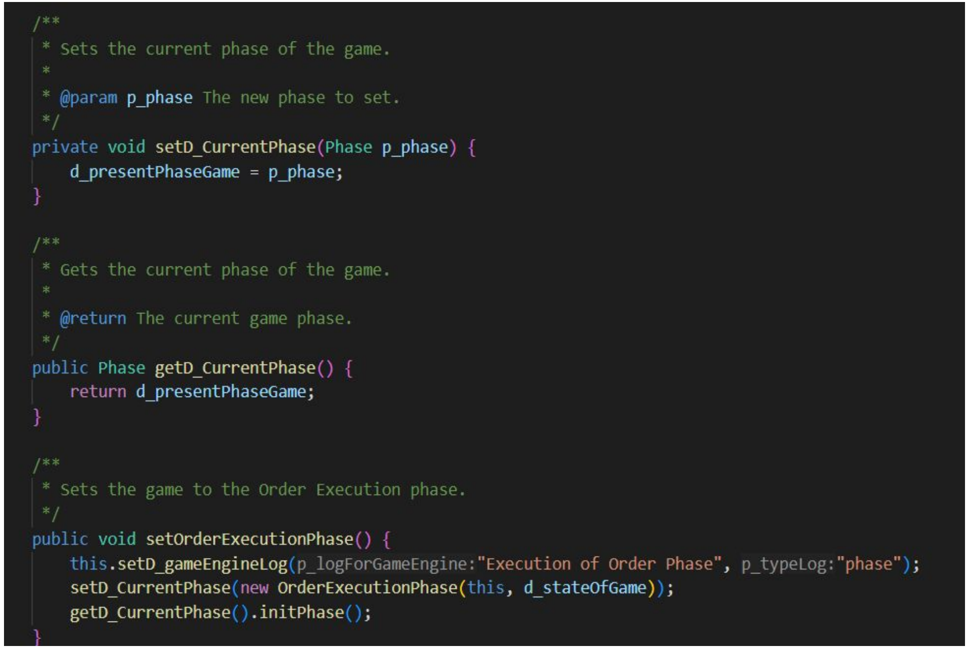
# Architectural Design - State Pattern

**Context:** This represents the object whose behavior is state-dependent. It holds a reference to a ConcreteState object that defines the current state of the Context. Some of its methods leverage the state-specific behavior of the State object to offer context-specific functionality. A change in the state object results in a change in the behavior of the Context object.

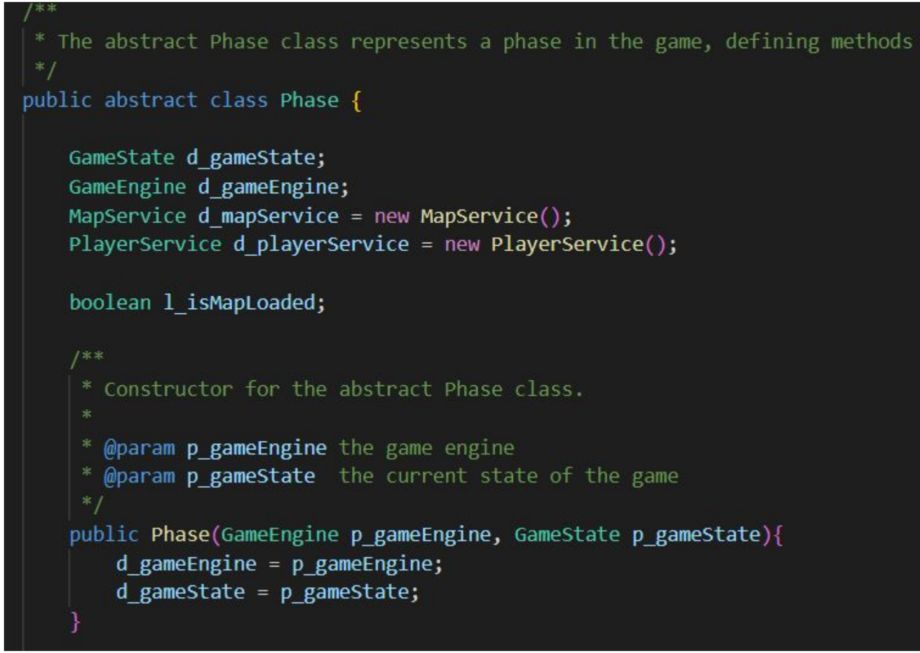
**State:** This class defines the operations that each state must handle. It is typically implemented as an abstract class or interface.

**ConcreteState:** These classes implement the state-specific behavior.

# Architectural Design - State Pattern (Example)



```java
/**
 * Sets the current phase of the game.
 *
 * @param p_phase The new phase to set.
 */
private void setD_CurrentPhase(Phase p_phase) {
    d_presentPhaseGame = p_phase;
}

/**
 * Gets the current phase of the game.
 *
 * @return The current game phase.
 */
public Phase getD_CurrentPhase() {
    return d_presentPhaseGame;
}

/**
 * Sets the game to the Order Execution phase.
 */
public void setOrderExecutionPhase() {
    this.setD_gameEngineLog(p_logForGameEngine:"Execution of Order Phase", p_typeLog:"phase");
    setD_CurrentPhase(new OrderExecutionPhase(this, d_stateOfGame));
    getD_CurrentPhase().initPhase();
}
```



```java
/**
 * The abstract Phase class represents a phase in the game, defining methods
 */
public abstract class Phase {

    GameState d_gameState;
    GameEngine d_gameEngine;
    MapService d_mapService = new MapService();
    PlayerService d_playerService = new PlayerService();

    boolean l_isMapLoaded;

    /**
     * Constructor for the abstract Phase class.
     *
     * @param p_gameEngine the game engine
     * @param p_gameState  the current state of the game
     */
    public Phase(GameEngine p_gameEngine, GameState p_gameState){
        d_gameEngine = p_gameEngine;
        d_gameState = p_gameState;
    }
```

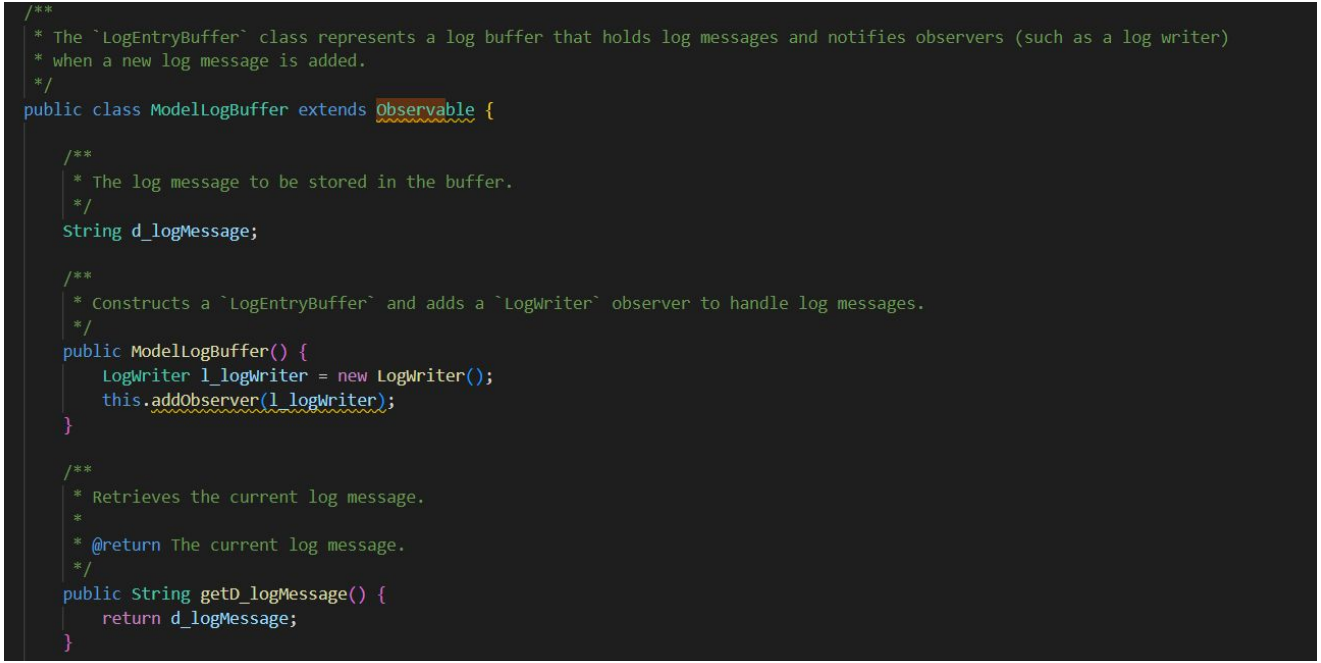
# Architectural Design - Observer Pattern

The elements of observer pattern are as follows:

**Subject** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. It is often referred to as “Observable”.

**ConcreteSubject** - concrete Subject class. It maintains the state of the observed object and when a change in its state occurs it notifies the attached Observers. If used as part of MVC, the ConcreteSubject classes are the Model classes that have Views attached to them.

# Architectural Design - Observer Pattern (Example)



```java
/**
 * The `LogEntryBuffer` class represents a log buffer that holds log messages and notifies observers (such as a log writer)
 * when a new log message is added.
 */
public class ModelLogBuffer extends Observable {

    /**
     * The log message to be stored in the buffer.
     */
    String d_logMessage;

    /**
     * Constructs a `LogEntryBuffer` and adds a `LogWriter` observer to handle log messages.
     */
    public ModelLogBuffer() {
        LogWriter l_logWriter = new LogWriter();
        this.addObserver(l_logWriter);
    }

    /**
     * Retrieves the current log message.
     *
     * @return The current log message.
     */
    public String getD_logMessage() {
        return d_logMessage;
    }
```

# Architectural Design - Command Pattern

The Command pattern comprises the following components:

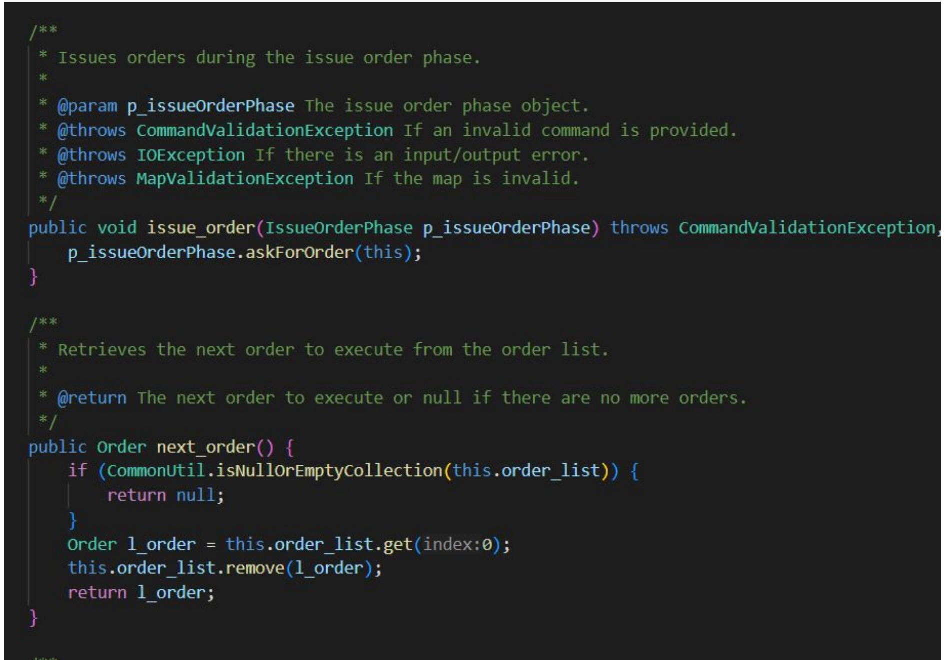
**Invoker**: This is an entity responsible for generating the command object required to execute a specific operation.

**Receiver**: It's an entity that will be impacted or utilized when the command is executed.

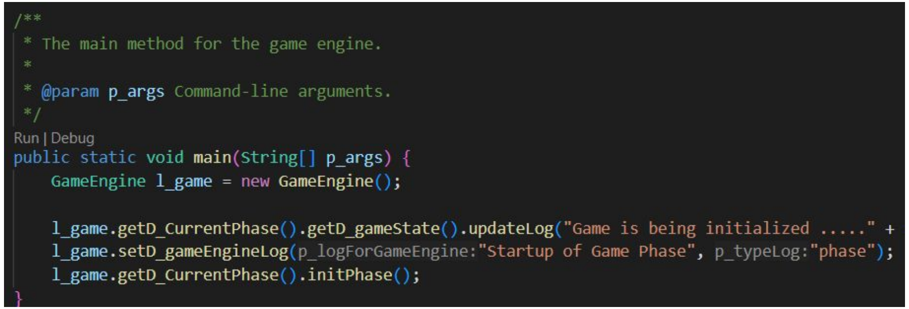
**Command**: This class defines the operations that each command must address. It's commonly realized as an abstract class or interface.

**ConcreteCommand**: This object holds the context required for operation execution and contains the code that performs the actual operation.

# Architectural Design - Command Pattern (Example)



```java
/**
 * Issues orders during the issue order phase.
 *
 * @param p_issueOrderPhase The issue order phase object.
 * @throws CommandValidationException If an invalid command is provided.
 * @throws IOException If there is an input/output error.
 * @throws MapValidationException If the map is invalid.
 */
public void issue_order(IssueOrderPhase p_issueOrderPhase) throws CommandValidationException,
    p_issueOrderPhase.askForOrder(this);
}

/**
 * Retrieves the next order to execute from the order list.
 *
 * @return The next order to execute or null if there are no more orders.
 */
public Order next_order() {
    if (CommonUtil.isNullOrEmptyCollection(this.order_list)) {
        return null;
    }
    Order l_order = this.order_list.get(index:0);
    this.order_list.remove(l_order);
    return l_order;
}
```



```java
/**
 * The main method for the game engine.
 *
 * @param p_args Command-line arguments.
 */
Run | Debug
public static void main(String[] p_args) {
    GameEngine l_game = new GameEngine();

    l_game.getD_CurrentPhase().getD_gameState().updateLog("Game is being initialized ....." +
    l_game.setD_gameEngineLog(p_logForGameEngine:"Startup of Game Phase", p_typeLog:"phase");
    l_game.getD_CurrentPhase().initPhase();
}
```

# Architectural Design - Adapter Pattern

**Context**: Object with behavior tied to an interface, relying on an Adapter for seamless integration and leveraging interface-specific functionality.

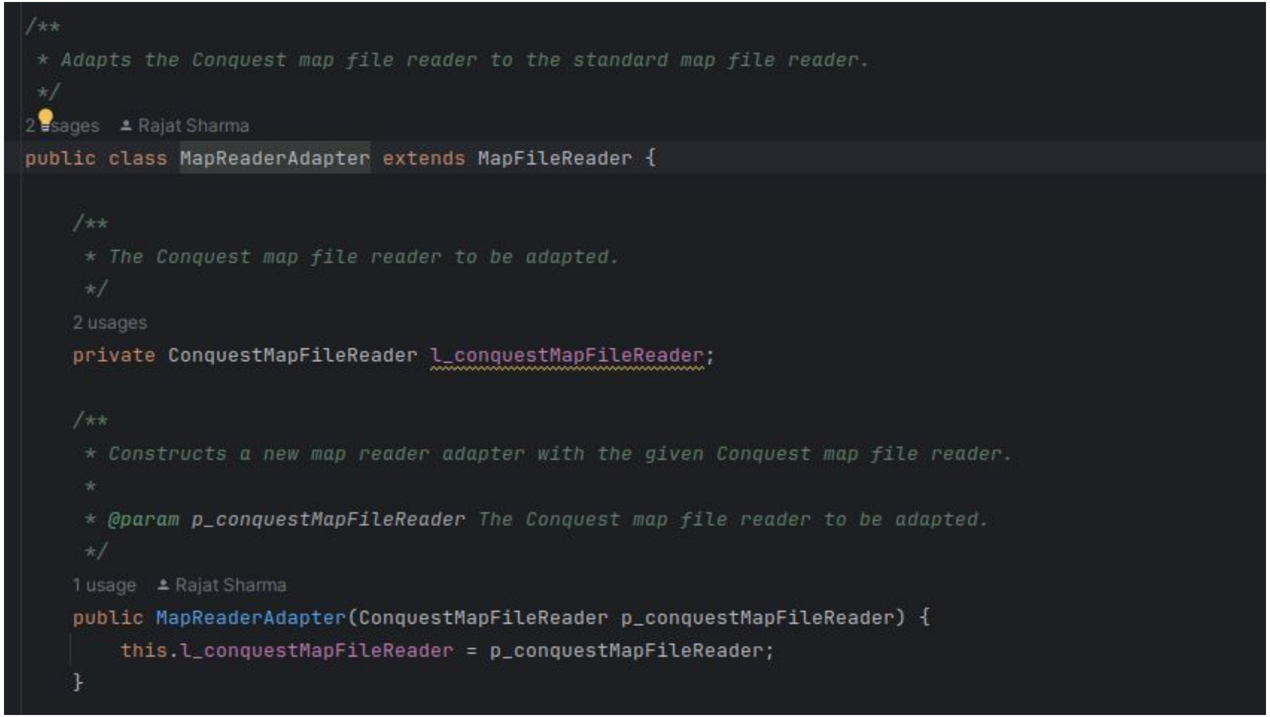
**Interface**: Abstract class/interface specifying methods for adaptee classes, ensuring standardized integration.

**Adapter**: Acts as a bridge, enabling the Context object to interact with adaptee functionalities through a common interface.

**Adaptee**: Holds existing functionality for integration, necessitating an Adapter for compatibility.

**Benefits**: Facilitates flexible and modular code, allowing the Context object to interact seamlessly with diverse adaptee classes.

# Architectural Design - Adapter Pattern (Example)



```
/**
 * Adapts the Conquest map file reader to the standard map file reader.
 */
2 usages  Rajat Sharma
public class MapReaderAdapter extends MapFileReader {

    /**
     * The Conquest map file reader to be adapted.
     */
    2 usages
    private ConquestMapFileReader l_conquestMapFileReader;

    /**
     * Constructs a new map reader adapter with the given Conquest map file reader.
     *
     * @param p_conquestMapFileReader The Conquest map file reader to be adapted.
     */
    1 usage  Rajat Sharma
    public MapReaderAdapter(ConquestMapFileReader p_conquestMapFileReader) {
        this.l_conquestMapFileReader = p_conquestMapFileReader;
    }
```

# Player Behavior Strategies in Game Dynamics

**Human Player**: Requires user interaction for decision-making.

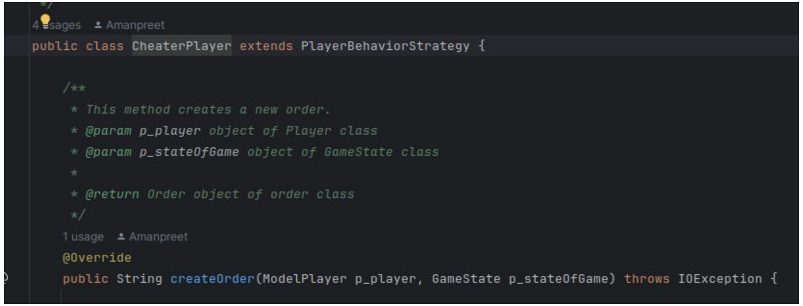
**Aggressive Computer Player**: Centralizes forces, attacks with the strongest country, and maximizes force aggregation.

**Benevolent Computer Player:** Focuses on protecting weak countries, reinforcing rather than attacking.

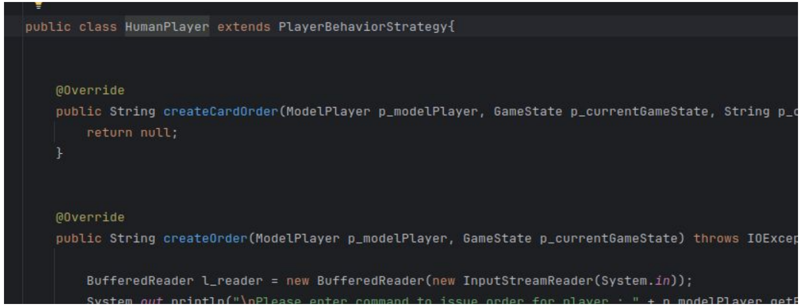
**Random Computer Player**: Deploys, attacks, and moves armies randomly within the game.

**Cheater Computer Player**: Conquers immediate neighboring enemies and doubles armies on territories with enemy neighbors, directly affecting the map during order creation.

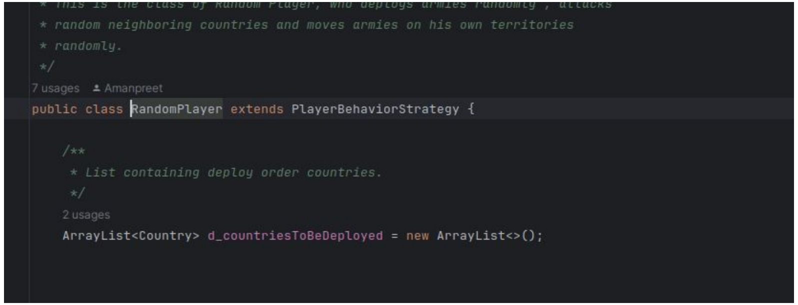
# Player Behavior Strategies (code snippets)



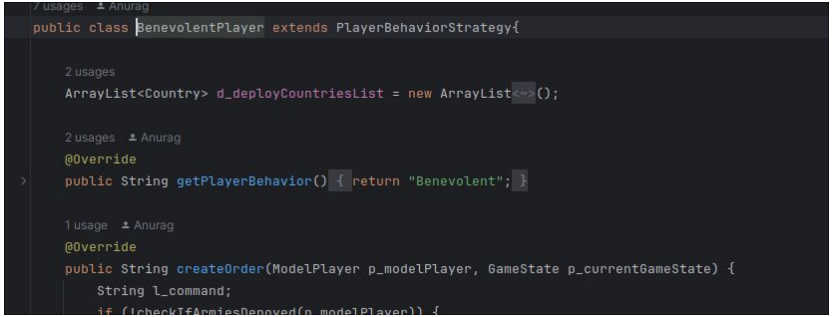
```
*/
4 usages  Amanpreet
public class CheaterPlayer extends PlayerBehaviorStrategy {

    /**
     * This method creates a new order.
     * @param p_player object of Player class
     * @param p_stateOfGame object of GameState class
     *
     * @return Order object of order class
     */
    1 usage  Amanpreet
    @Override
    public String createOrder(ModelPlayer p_player, GameState p_stateOfGame) throws IOException {
```



```
public class HumanPlayer extends PlayerBehaviorStrategy{


    @Override
    public String createCardOrder(ModelPlayer p_modelPlayer, GameState p_currentGameState, String p_c
        return null;
    }


    @Override
    public String createOrder(ModelPlayer p_modelPlayer, GameState p_currentGameState) throws IOExcep

        BufferedReader l_reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("\nPlease enter command to issue order for player : " + p_modelPlayer.getP
```



```
 * This is the class of Random Player, who deploys armies randomly , attacks
 * random neighboring countries and moves armies on his own territories
 * randomly.
 */
7 usages  Amanpreet
public class RandomPlayer extends PlayerBehaviorStrategy {

    /**
     * List containing deploy order countries.
     */
    2 usages
    ArrayList<Country> d_countriesToBeDeployed = new ArrayList<>();
```



```
7 usages  Anurag
public class BenevolentPlayer extends PlayerBehaviorStrategy{

    2 usages
    ArrayList<Country> d_deployCountriesList = new ArrayList<~>();

    2 usages  Anurag
    @Override
    public String getPlayerBehavior() { return "Benevolent"; }

    1 usage  Anurag
    @Override
    public String createOrder(ModelPlayer p_modelPlayer, GameState p_currentGameState) {
        String l_command;
        if (!checkIfArmiesDepoyed(p_modelPlayer)) {
```

# Tournament Mode Overview

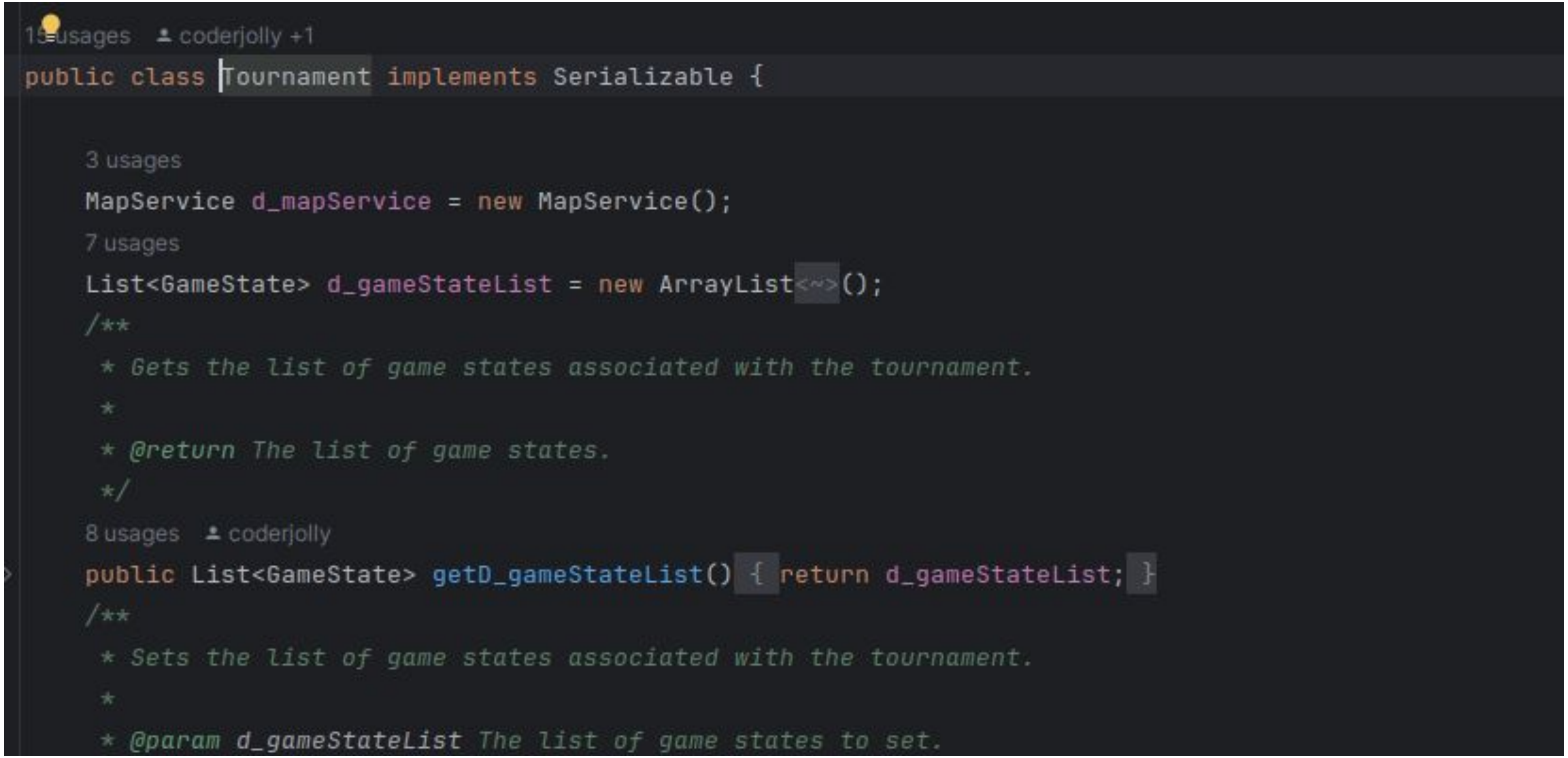
**Tournament Configuration:**

User chooses M (1-5 maps), P (2-4 player strategies), G (1-5 games per map), and D (10-50 turns per game).

**Automatic Tournament Execution:**

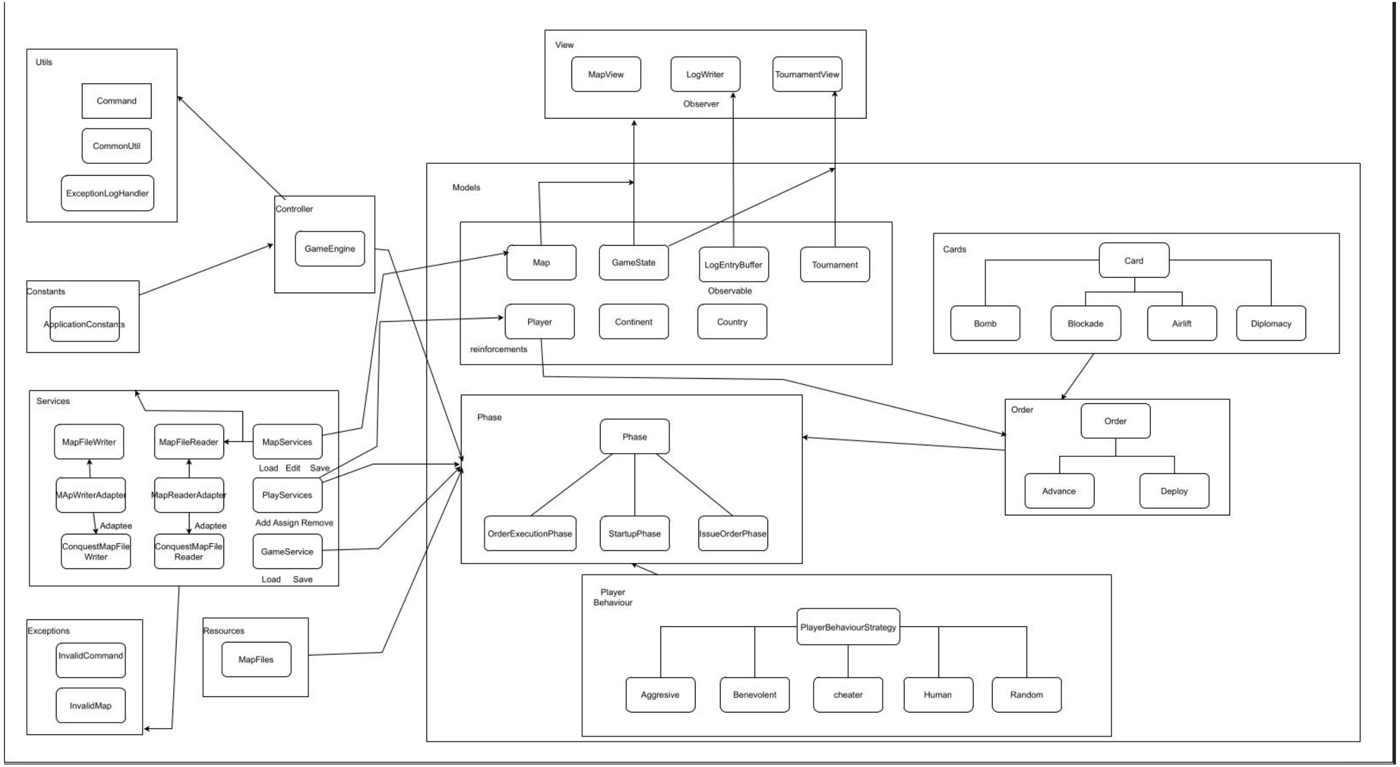
Plays G games on each of the M maps between chosen computer player strategies.
Games are automatically played without user interaction.
Draws are declared after D turns to minimize run completion time.

# Tournament Mode Overview (code snippet)



```
15 usages   coderjolly +1
public class Tournament implements Serializable {

    3 usages
    MapService d_mapService = new MapService();
    7 usages
    List<GameState> d_gameStateList = new ArrayList<~>();
    /**
     * Gets the list of game states associated with the tournament.
     *
     * @return The list of game states.
     */
    8 usages   coderjolly
    public List<GameState> getD_gameStateList() { return d_gameStateList; }
    /**
     * Sets the list of game states associated with the tournament.
     *
     * @param d_gameStateList The list of game states to set.
```

# Our Design


Utils
Command
CommonUtil
ExceptionLogHandler
View
MapView
LogWriter
TournamentView
Observer
Controller
GameEngine
Constants
ApplicationConstants
Models
Map
GameState
LogEntryBuffer
Tournament
Observable
Player
Continent
Country
reinforcements
Cards
Card
Bomb
Blockade
Airlift
Diplomacy
Services
MapFileWriter
MapFileReader
MapServices
Load Edit Save
MApWriterAdapter
MapReaderAdapter
PlayServices
Adaptee
Adaptee
Add Assign Remove
ConquestMapFile Writer
ConquestMapFile Reader
GameService
Load Save
Phase
Phase
OrderExecutionPhase
StartupPhase
IssueOrderPhase
Order
Order
Advance
Deploy
Exceptions
InvalidCommand
InvalidMap
Resources
MapFiles
Player Behaviour
PlayerBehaviourStrategy
Aggresive
Benevolent
cheater
Human
Random