COMP 345 – Fall 2025

Team Project Assignment #1 – Detailed Elaboration

This document provides a detailed explanation of each part of development of Warzone for COMP 345 Assignment 1, including the design purpose, required classes, and behavior of each method. It also explains how components interact with each other following the Warzone game rules.

Part 1: Map and MapLoader

The Map and MapLoader classes define the structure of the game world as a connected graph. Territories act as nodes, and edges represent adjacency between them. A map object also contains a hashmap that contains the sum of all numeric (obtained through rolling polynomial hashing) territory IDs for every continent. It is used to allow for O(1) lookup time for detecting if the player controls a continent, or if a player has won.

Key points:
- Each Territory belongs to exactly one Continent.
- Continents are connected subgraphs of the overall map.
- The Map class must verify the map's structural integrity using the `validate()` method.

The `validate()` method performs:

1. Checks that the entire map is connected (every territory reachable), using Depth First Search.
2. Verifies that each continent is internally connected, using Depth First Search.
3. Confirms that every territory belongs to exactly one continent.

The `MapLoader` reads `.map` files in the Conquest format and builds Map objects. It should correctly handle both valid and invalid maps. For invalid maps, it's able to detect what the error is. A driver named `testLoadMaps()` demonstrates valid and invalid file handling, with appropriate validation outputs.

Part 2: Player

The Player class represents an individual player in the game. A player owns territories, holds a hand of cards, and maintains a list of issued orders. They also hold a hashmap, like that found in the map object. The game engine is able to make use of these hashmaps to make constant O(1) comparisons, significantly increasing the speed at which the game runs.

Required methods:

- `toDefend()`: Returns a list of territories owned by the player that should be defended. At this stage, this list is arbitrary but will later prioritize strategic defense.
- `toAttack()`: Returns a list of target territories to attack, chosen arbitrarily for now.
- `issueOrder()`: Creates an Order object (e.g., Deploy, Advance, Bomb) and adds it to the player's order list.

Important behavioral rules:
- A player may only issue certain orders if they possess the corresponding card in their Hand.
  * For example, to issue a Bomb order, the player must have a Bomb card.
  * The `issueOrder()` method should check for this and only create valid orders. At the moment, the logic for which a player decides to issue an order is completely arbitrary.
- The player interacts with the Map module to select territories for orders.
- Orders are stored in the player's `OrderList` for later execution by the GameEngine.

The `testPlayers()` driver should:

1. Create players.
2. Assign them territories (from Part 1).
3. Demonstrate that they can produce defend/attack lists.
4. Show that orders can be issued and stored.

Part 3: Orders and OrdersList

Orders represent actions taken by players. The `Order` class is abstract, with subclasses for each type of action. The `OrdersList` stores all issued orders in sequence and allows rearranging or removal.

Each subclass must implement:

- `validate()`: Checks whether the order is legal (e.g., the source territory is owned by the player, the target is adjacent, the player is not negotiating with the target, etc.).
- `execute()`: Performs the action if valid; otherwise prints an error and does nothing.

Order types:

- `Deploy`: Places reinforcements on a player-owned territory.
- `Advance`: Moves or attacks adjacent territories.
- `Bomb`: Destroys half of the armies in an enemy territory. Requires a Bomb card.
- `Blockade`: Triples armies on a territory and transfers it to Neutral. Requires a Blockade card.
- `Airlift`: Moves armies between any two owned territories (ignores adjacency). Requires an Airlift card.
- `Negotiate`: Prevents conflict between two players for one turn. Requires a Diplomacy card.

The `OrdersList` supports:
- `addOrder()`: Appends new orders.
- `move()`: Reorders the list.
- `remove()`: Deletes a specific order.

The driver `testOrdersLists()` should create each order type, add them to an OrdersList, demonstrate moving/removing, and print their validation and execution results.

Part 4: Cards, Deck, and Hand

This component handles the card system that gives players access to special order types.

Classes:
- `Card`: Represents a single card (Bomb, Reinforcement, Blockade, Airlift, Diplomacy).
- `Deck`: Global pool of cards. The `draw()` method allows a player to draw a random card from the deck.

- `Hand`: A player’s set of cards. Players play cards using `playCard()`.

Mechanics:
- `Deck::draw()`: Randomly selects a card and moves it into the player's Hand.
- `Card::play()`: Converts the card into an Order, adds it to the player’s OrdersList, and returns the card to the Deck.
- When `Card::play()` is used, the card is removed from the Hand automatically.
- Playing a card is the only way to create Bomb, Airlift, Blockade, or Negotiate orders.

Driver `testCards()` should demonstrate:
1. Creating a Deck with all card types.
2. Drawing cards into a Hand.
3. Playing all cards and showing that Orders are created and cards return to the Deck.

Part 5: Game Engine

The GameEngine controls the overall game flow using a state machine that manages transitions between game phases. It enforces that only valid commands can be executed in each state.

States:
Start → MapLoaded → MapValidated → PlayersAdded → AssignReinforcement → IssueOrders → ExecuteOrders → Win → End

Core methods:

- `loadmap()`: Loads a map file using MapLoader. Can only be called in Start state.
- `validatemap()`: Validates the loaded map. Transitions to MapValidated state.
- `addplayer(name)`: Adds a new player to the game. Can only be called after the map is validated.
- `gamestart()`: Initializes the deck, assigns territories to players, and gives each two cards.
- `assignreinforcement()`: Gives reinforcements to each player based on the number of owned territories and controlled continents.
- `issueorder()`: Players issue orders (requires valid cards for special orders).
- `endissueorders()`: Transitions to ExecuteOrders phase.
- `executeOrder()`: Executes all player orders in an interleaved fashion.

- `win()`: Checks victory condition (player controls all territories).
- `end()`: Terminates the game session.

The driver `testGameStates()` should demonstrate console interaction with all valid and invalid commands. Invalid commands should print an error message and not change the current state. The secondary function ‘simulateRealGame’ demonstrates what a game of warzone (with no player strategy, just purely random) would look like.