Concordia University
Department of Computer Science
and Software Engineering

Advanced program design with C++
COMP 345 --- Fall 2025

Team project assignment #2

Deadline:	November 11th, 2025
Evaluation:	7% of final mark
Late submission:	not accepted
Teams:	this is a team assignment

Problem statement

This is a team assignment. It is divided into distinct parts. Each part is about the development of a part of the topic presented as the team project. Even though it is about the development of a part of your team project, each assignment is to be developed/presented/tested separately. The description of each part describes what are the features that the part should implement, and what you should demonstrate. Note that the following descriptions describe the baseline of the assignment, and are related to the project description. See the course web page for a full description of the team project, as well as links to the details of the game rules to be implemented.

Specific design requirements

1. All data members of user-defined class type must be of pointer type.
2. All file names and the content of the files must be according to what is given in the description below.
3. All different parts must be implemented in their own separate .cpp/.h file duo. All functions' implementation must be provided in the .cpp file (i.e. no inline functions are allowed).
4. All classes must implement a correct copy constructor, assignment operator, and stream insertion operator.
5. Memory leaks must be avoided.
6. Code must be documented using comments (user-defined classes, methods, free functions, operators).
7. If you use third-party libraries that are not available in the labs and require setup/installation, you may not assume to have help using them and you are entirely responsible for their proper installation for grading purposes.
8. All the code developed in assignment 2 must stay in the same files as specified in assignment #1:
   - Map loading and in-game map implementation: **`Map.cpp/Map.h`**
   - Player implementation: **`Player.cpp/Player.h`**
   - Card hand and deck implementation: **`Cards.cpp/Cards.h`**
   - Orders implementation: **`Orders.cpp/Orders.h`**
   - Game controller implementation: **`GameEngine.cpp/GameEngine.h`**

Parts to be implemented

Part 1: Command processor and command adapter

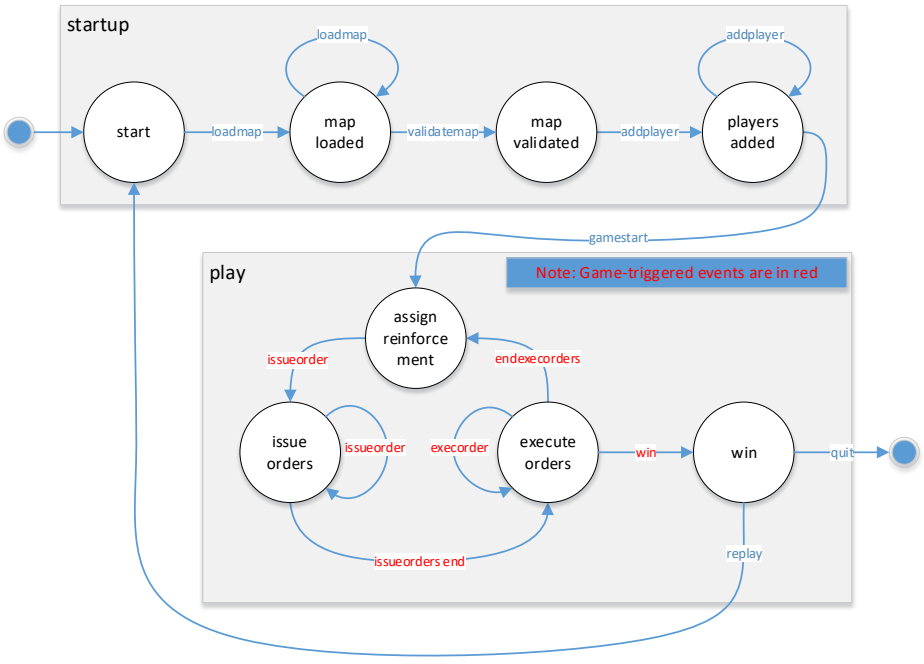
Implement a **`CommandProcessor`** class that gets commands from the console as a string using its **`readCommand()`** private method which stores the command internally in a collection of **`Command`** objects using the **`saveCommand()`** method, and provides a public **`getCommand()`** method to other objects such as the **`GameEngine`** or the **`Player`**. Once a command gets executed, the effect of the command can be stored as a string in the **`Command`** object using the **`saveEffect()`** method. Any game component that operates using commands must get its commands from a **`CommandProcessor`** object. The command processor should have a **`validate()`** method that can be called to check if the command is valid in the current game state. If the command is not valid, a corresponding error message should be saved in the effect of the command.

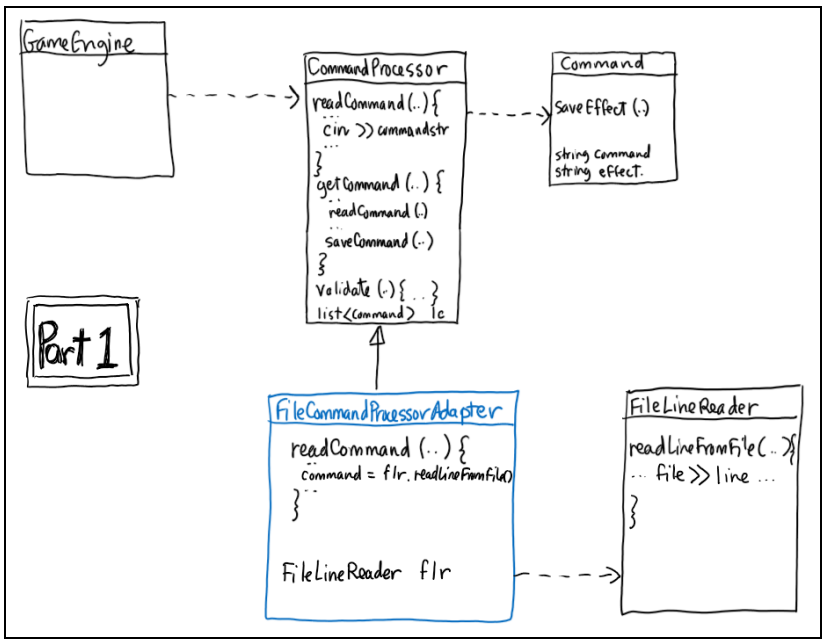
Implement a **`FileCommandProcessorAdapter`** class that enables the same functionality as the above, except that it reads the commands sequentially from a previously saved text file. This class needs to abide with all the design characteristics and restrictions of the Adapter design pattern.

The application should accept a command line argument that enables the user to choose between accepting the commands from the console (**`-console`**) or from a file (**`-file <filename>`**).

The commands are:

command	valid in state	transitions to
loadmap <mapfile>	start, maploaded	maploaded
validatemap	maploaded	mapvalidated
addplayer <playername>	mapvalidated, playersadded	playersadded
gamestart	playersadded	assignreinforcement
replay	win	start
quit	win	exit program





Sum-up of the proposed design for Part 1.

This must be implemented as part of a single **.cpp/.h** file duo named **CommandProcessing.cpp/ CommandProcessing.h**. You must deliver a driver as a free function named **testCommandProcessor()** that demonstrates that (1) commands can be read from the console using the **CommandProcessor** class (2) commands can be read from a saved text file using the **FileCommandProcessorAdapter** (3) commands that are invalid in the current game state are rejected, and valid commands result in the correct effect and state change. This driver function must be in the **CommandProcessingDriver.cpp** file.

Part 2: Game startup phase

Provide a **startupPhase()** method in the **GameEngine** class that implements a command-based user interaction mechanism to start the game by allowing the user to proceed with the game startup phase:

1) use the **loadmap <filename>** command to select a map from a list of map files as stored in a directory, which results in the map being loaded in the game.
2) use the **validatemap** command to validate the map (i.e. it is a connected graph, etc – see assignment 1).
3) use the **addplayer <playername>** command to enter players in the game (2-6 players)
4) use the **gamestart** command to
   a) fairly distribute all the territories to the players
   b) determine randomly the order of play of the players in the game
   c) give 50 initial army units to the players, which are placed in their respective reinforcement pool
   d) let each player draw 2 initial cards from the deck using the deck's **draw()** method
   e) switch the game to the **play** phase

This must be implemented as part of the pre-existing **.cpp/.h** file duo named **GameEngine.cpp/GameEngine.h** You must deliver a driver as a free function named **testStartupPhase()** that demonstrates that 1-4 explained above are implemented correctly, using either console input or file input of the commands (see Part 1). This driver function must be in the **GameEngineDriver.cpp** file.

Part 3: Game play: main game loop

Provide a group of C++ classes that implements the main game loop following the official rules of the Warzone game. During the main game loop, proceeding in a round-robin fashion in the order setup in startup phase 4b. The main game loop has three phases and is implemented in a function/method named **mainGameLoop()** in the **GameEngine** class:

1. Reinforcement Phase—Players are given a number of army units that depends on the number of territories they own, (# of territories owned divided by 3, rounded down). If the player owns all the territories of an entire continent, the player is given a number of army units corresponding to the continent's control bonus value. In any case, the minimal number of reinforcement army units per turn for any player is 3. These army units are placed in the player's reinforcement pool. This must be implemented in a function/method named **`reinforcementPhase()`** in the game engine.
2. Issuing Orders Phase—Players issue orders and place them in their order list through a call to the **`Player::issueOrder()`** method. This method is called in round-robin fashion across all players by the game engine. This phase ends when all players have signified that they don't have any more orders to issue for this turn. This must be implemented in a function/method named **`issueOrdersPhase()`** in the game engine.
3. Orders Execution Phase—Once all the players have signified in the same turn that they are not issuing one more order, the game engine proceeds to execute the top order on the list of orders of each player in a round-robin fashion (i.e. the "Order Execution Phase"—see below). Once all the players' orders have been executed, the main game loop goes back to the reinforcement phase. This must be implemented in a function/method named **`executeOrdersPhase()`** in the game engine.

This loop shall continue until only one of the players owns all the territories in the map, at which point a winner is announced and the game ends. The main game loop also checks for any player that does not control at least one territory; if so, the player is removed from the game.

Orders Issuing phase
The issuing orders phase decision-making is implemented in the player's **`issueOrder()`** method, which implements the following:

- The player decides which neighboring territories are to be attacked in priority (as a list return by the **`toAttack()`** method), and which of their own territories are to be defended in priority (as a list returned by the **`toDefend()`** method).
- The player issues deploy orders on its own territories that are in the list returned by **`toDefend()`**. As long as the player has army units in their reinforcement pool (see startup phase and reinforcement phase), it will issue a deploy order and no other order. Once it has deployed all its available army units, it can proceed with other kinds of orders.
- The player issues advance orders to either (1) move army units from one of its own territory to another of its own territories in order to defend it (using **`toDefend()`** to make the decision), and/or (2) move army units from one of its own territories to a neighboring enemy territory to attack them (using **`toAttack()`** to make the decision).
- The player uses one of the cards in their hand to issue an order that corresponds to the card in question.

Note that the orders should not be validated as they are issued. Orders are to be validated only when they are executed in the orders execution phase. This must be implemented in a function/method named **`issueOrdersPhase()`** in the game engine. The decision-making code must be implemented within the **`issueOrder()`** method of the player class in the **`Player.cpp/Player.h`** files.

Orders execution phase
When the game engine asks the player to give them their next order, the player returns the next order in their order list. Once the game engine receives the order, it calls **`execute()`** on the order, which should first validate the order, then enact the order (see Part 4: orders execution implementation) and record a narrative of its effect stored in the order object. The game engine should execute all the deploy orders before it executes any other kind of order. This goes on in round-robin fashion across the players until all the players' orders have been executed.

You must deliver a driver as a free function named **`testMainGameLoop()`** that demonstrates that (1) a player receives the correct number of army units in the reinforcement phase (showing different cases); (2) a player will only issue deploy orders and no other kind of orders if they still have army units in their reinforcement pool; (3) a player can issue advance orders to either defend or attack, based on the **`toAttack()`** and **`toDefend()`** lists; (4) a player can play cards to issue orders; (5) a player that does not control any territory is removed from the game; (6) the game ends when a single player controls all the territories. All of this must be implemented in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**, except the **`issueOrder()`** method, which is a member

of the **`Player`** class, implemented in the **`Player.h/Player.h`** file duo. This driver function must be in the **`GameEngineDriver.cpp`** file.

Part 4: Order execution implementation

Provide a group of C++ classes that implements the execution of orders following the official rules of the Warzone game. The code that implements the execution of the orders must be placed within the **`execute()`** method of the order class/subclasses in the **`Orders.cpp/Orders.h`** files. Each specific order kind (listed below) must be a subclass of an abstract class named **`Order`** that has a pure virtual method named **`execute()`**.

Deploy order: A deploy order tells a certain number of army units taken from the reinforcement pool to deploy to a target territory owned by the player issuing this order.

- If the target territory does not belong to the player that issued the order, the order is invalid.
- If the target territory belongs to the player that issued the deploy order, the selected number of army units is added to the number of army units on that territory.

Advance order: An advance order tells a certain number of army units to move from a source territory to a target adjacent territory.

- If the source territory does not belong to the player that issued the order, the order is invalid.
- If the target territory is not adjacent to the source territory, the order is invalid.
- If the source and target territory both belong to the player that issued the order, the army units are moved from the source to the target territory.
- If the target territory belongs to another player than the player that issued the advance order, an attack is simulated when the order is executed. An attack is simulated by the following battle simulation mechanism:
  - Each attacking army unit involved has 60% chances of killing one defending army. At the same time, each defending army unit has 70% chances of killing one attacking army unit.
  - If all the defender's army units are eliminated, the attacker captures the territory. The attacking army units that survived the battle then occupy the conquered territory.
  - A player receives a card at the end of his turn if they successfully conquered at least one territory during their turn, i.e. a player cannot receive more than one card per turn.

Airlift order: An airlift order tells a certain number of army units taken from a source territory to be moved to a target territory, the source and the target territory being owned by the player issuing the order. The airlift order can only be created by playing the airlift card.

- The target territory does not need to be adjacent to the source territory.
- If the source or target territory does not belong to the player that issued the order, the order is invalid.
- If both the source and target territories belong to the player that issue the airlift order, then the selected number of army units is moved from the source to the target territory.

Bomb order: A bomb order targets a territory owned by another player than the one issuing the order. Its result is to remove half of the army units from this territory. The bomb order can only be created by playing the bomb card.

- If the target belongs to the player that issued the order, the order is invalid.
- If the target territory is not adjacent to one of the territory owned by the player issuing the order, then the order is invalid.
- If the order is valid, half of the army units are removed from this territory.

Blockade order: A blockade order targets a territory that belongs to the player issuing the order. Its effect is to double the number of army units on the territory and to transfer the ownership of the territory to the Neutral player. The blockade order can only be created by playing the blockade card.

- If the target territory belongs to an enemy player, the order is declared invalid.
- If the target territory belongs to the player issuing the order, the number of army units on the territory is doubled and the ownership of the territory is transferred to the Neutral player, which must be created if it does not already exist.

Negotiate order: A negotiate order targets an enemy player. It results in the target player and the player issuing the order to not be able to successfully attack each others' territories for the remainder of the turn. The negotiate order can only be created by playing the diplomacy card.

- If the target is the player issuing the order, then the order is invalid.
- If the target is an enemy player, then the effect is that any attack that may be declared between territories of the player issuing the negotiate order and the target player will result in an invalid order.

You must deliver a driver as a free function named **`testOrderExecution()`** that demonstrates that (1) each order is validated before being executed according to the above descriptions; (2) ownership of a territory is transferred to the attacking player if a territory is conquered as a result of an advance order; (3) one card is given to a player if they conquer at least one territory in a turn (not more than one card per turn); (4) the negotiate order prevents attacks between the two players involved; (5) the blockade order transfers ownership to the Neutral player; 6) all the orders described above can be issued by a player and executed by the game engine. This driver function must be in the **`OrdersDriver.cpp`** file.

Part 5: Game log observer: commands and orders

Use the Observer pattern (i.e. **`Subject`** and **`Observer`** classes) to implement a game log class named **`LogObserver`** that writes every game command read by a **`CommandProcessor`** object (or a **`FileCommandProcessorAdapter`** object) into a log file. The game log observer should be notified by the **`CommandProcessor`**::**`saveCommand()`** method that saves the command into the collection of commands, as well as the and **`Command::saveEffect()`** method that records the effect of the command into the command object**.** This should result in all the current game's commands and their effects to be logged into a "**`gamelog.txt`**" file.

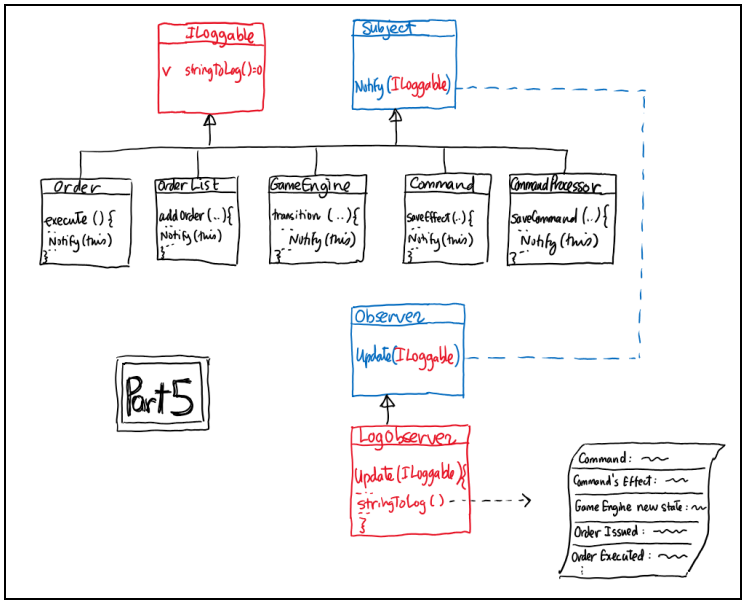
In the same way, when any order is put in a player's **`OrderList`** using **`OrderList::addOrder()`** the order should be output to the log file using the Observer notification mechanism. Later, when any order gets executed, its effect (as stored in the order itself) should be output to the log file, again using the Observer's notification mechanism.

Also in the same way, when the **`GameEngine`**'s state is changing (e.g. using **`GameEngine::transition()`**), a log line should be output to the log file stating what is the new game state, using the Observer notification mechanism.

As part of your design for this solution you must include a class named **ILoggable** that must be inherited by all classes that can be the subject of logging mechanism as described above. This class is essentially an interface that must contain a pure virtual member function name **`stringToLog()`** that creates and returns a string to be output to the log file.

The **`Observer`**, **`Subject`** and **`ILoggable`** classes must be implemented as part of a single **`.cpp/.h`** file duo named **`LoggingObserver.cpp/LoggingObserver.h`**. See below for a snapshot of the diagram used in the video.

You must deliver a driver as a free function named **`testLoggingObserver()`** that demonstrates that (1) The **`Command`**, **`CommandProcessor`** (see Part 1), **`Order`**, **`OrderList`**, and **`GameEngine`** classes are all a subclass of the **`Subject`** and **`ILoggable`** classes (2) the **`CommandProcessor::saveCommand()`**, **`Order::execute()`, `Command::saveEffect()`**, **`OrderList::addOrder()`**, and **`GameEngine::transition()`** methods are effectively using the Observer patterns' **`Notify(Subject)`** method to trigger the writing of an entry in the log file (3) the **`gamelog.txt`** file gets correctly written into when commands are entered on the console (4) when an order is added to the order list of a player, the game log observer is notified which results in outputting the order to the log file (5) when an order is executed, the game log observer is notified which results in outputting the effect of the order to the log file (6) when the **`GameEngine`** changes its state, the new state is output to the log file. This driver function must be in the **`LoggingObserverDriver.cpp`** file.



Sum-up of the proposed design for Part 5.

Assignment submission requirements and procedure

You are expected to submit a group of C++ files implementing a solution to all the problems stated above (Part 1, 2, 3, 4, 5). Your code must include a single main function that calls all of the five above-mentioned **`test*()`** free functions that allows the marker to observe the execution of each part during the lab demonstration. The **`main`** function should be in the file **`MainDriver.cpp`**. Each driver should simply create the components described above and demonstrate that they behave as mentioned above. If your code also now include a possibility for the game to be executed without running the tests, then your main function should include a command line option **`-test`** to only run the tests when the application is started, and run the game normally if the **`-test`** option is not given.

You have to submit your assignment before midnight on the due date using the Moodle page for the course, under the category “programming assignment 2”. Late assignments are not accepted. The file submitted must be a .zip file containing all your C++ code. Do not submit other files such as the project file from your IDE. You are allowed to use any C++ programming environment as long as you can demonstrate your assignment in on zoom during demonstration time.

Evaluation Criteria

Knowledge/correctness of game rules: *Mark deductions: during the presentation or code review it is found that the implementation does not follow the rules of the Warzone game.*	**2 pts (indicator 4.1)**
Compliance of solution with stated problem (see description above): *Mark deductions: during the presentation or code review, it is found that the code does not do some of which is asked in the above description.*	**12 pts (indicator 4.4)**
Modularity/simplicity/clarity of the solution: *Mark deductions: some of the data members are not of pointer type; or the above indications are not followed regarding the files needed for each part.*	**2 pts (indicator 4.3)**
Mastery of language/tools/libraries: *Mark deductions: constructors, destructor, copy constructor, assignment operators not implemented or not implemented correctly; the program crashes during the presentation and the presenter is not able to right away correctly explain why.*	**2 pts (indicator 5.1)**
Code readability: naming conventions, clarity of code, use of comments: *Mark deductions: some names are meaningless, code is hard to understand, comments are absent, presence of commented-out code.*	**2 pts (indicator 7.3)**
Total	**20 pts (indicator 6.4)**