Concordia University
Department of Computer Science
and Software Engineering

Advanced program design with C++
COMP 345 --- Fall 2020

Team project assignment #2

Deadline:	November 13th, 2020
Evaluation:	10% 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. All the code developed in assignment 2 must stay in the same files as specified in assignment #1:

- Map implementation: **`Map.cpp/Map.h`**.
- Map loader implementation: **`MapLoader.cpp/MapLoader.h`**.
- Orders list implementation: **`OrdersList.cpp/orders.h`**.
- Player implementation: **`Player.cpp/Player.h`**.
- Card hand and deck implementation: **`Cards.cpp/Cards.h`**.

Specific design requirements

1. All data of user-defined class type must be of pointer type.
2. In case using smart pointer, the data of user-defined class type must not be all smart pointers.
3. All file names and the content of the files must be according to what is given in the description below.
4. 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).
5. All classes must implement a correct copy constructor, assignment operator, and stream insertion operator.
6. Memory leaks must be avoided.
7. Code must be documented using comments (user-defined classes, methods, free functions, operators).
8. 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.

Part 1: Game start

Provide a group of C++ classes that implement a user interaction mechanism to start the game by allowing the player to 1) select a map from a list of map files as stored in a directory 2) select the number of players in the game (2-5 players) and 3) turn on/off any of the observers as described in Part 5. The code should then use the map loader to load the selected map, create all the players, assign a set of battle orders, create a deck of cards, and assign an empty hand of cards to each player. You must deliver a driver that demonstrates that 1) different valid maps can be loaded and their validity is verified (i.e. it is a connected graph, etc.), and invalid maps are gracefully rejected; 2) the right number of players is created, a deck with the right number of cards is created; 3) the observers can be turned on/off during the game start phase. This must be implemented in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**.

Part 2: Game play: game startup phase

Provide a group of C++ classes that implements the startup phase following the official rules of the Warzone game. This phase is composed of the following sequence:

1. The order of play of the players in the game is determined randomly.
2. All territories in the map are randomly assigned to players one by one in a round-robin fashion.
3. Players are given a number of initial armies (A), where A is:
    - If 2 players, A=40
    - If 3 players, A=35
    - If 4 players, A=30
    - If 5 players, A=25

These armies are placed in the reinforcement pool. This must be implemented in a function/method named **`startupPhase()`** in the game engine. You must deliver a driver that demonstrates that 1) all territories in the map have been assigned to one and only one player; 2) Players are given a number of armies; 3) all players have all the orders for playing in a turn. This must be implemented in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**.

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 as setup in the startup phase, the main game loop has three phases and is implemented in a function/method named **`mainGameLoop()`** in the game engine:

1. Reinforcement Phase—Players are given a number of armies 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 armies corresponding to the continent's control bonus value. In any case, the minimal number of reinforcement armies per turn for any player is 3. These armies 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 (i.e. the "order issuing phase"—see below) through a call to the **`issueOrder()`** method. This method is called in round-robin fashion by the game engine. 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 armies still to deploy (see startup phase and reinforcement phase), it will issue a deploy order and no other order. Once it has deployed all its available armies, it can proceed with other kinds of orders.
- The player issues advance orders to either (1) move armies from one of its own territory to the other in order to defend them (using **`toDefend()`** to make the decision), or (2) move armies from one of its 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.

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 their highest-priority order in their list of orders (priorities: 1:deploy 2: airlift 3:blockade 4:all the others). Once the game engine receives the order, it calls **`execute()`** on the order, which should 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 that demonstrates that 1) a player receives the correct number of armies 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 armies in their reinforcement pool; 3) the game engine will only execute non-deploy orders when all the deploy orders of all players have been executed; 4) a player can issue advance orders to either defend or attack, based on the **`toAttack()`** and **`toDefend()`** lists; 5) a player can play cards to issue orders; 6) a player that does not control any territory is removed from the game; 7) the game ends when a single player controls all the territories. All of this except the **`issueOrder()`** method must be implemented in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**.

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 armies 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 armies is added to the number of armies on that territory.

Advance order: An advance order tells a certain number of army units to move from a source territory to a target territory.
- If the source territory does not belong to the player that issued the order, 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, 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 armies 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.

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

- If the source or target 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 armies is attacking that territory (see “advance order”).

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 armies 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 belongs to an enemy player, half of the armies 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 doubling the number of armies on the territory and transferring the ownership of the territory to the Neutral player.

- If the target territory belongs to an enemy player, the order is declared invalid. The blockade order can only be created by playing the blockade card.
- If the target territory belongs to the player issuing the order, the number of armies on the territory is doubled and the ownership of the territory is transferred to the Neutral player.

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 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 or airlift 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.

Part 5: Observers

Provide a group of C++ classes that implements observers who are used to display information to the user as the game is being played. The user shall have the option to turn on/off any of the two observers during the game start phase (see Part 1).

Phase Observer

Using the Observer design pattern, implement a view that displays information happening in the current phase. It should first display a header showing what player and what phase is currently being played, e.g. “Player 3: Reinforcement phase” or “Player 1: Issue orders phase” Then it should display important information related to what is happening in this phase, which should be different depending on what phase is being played. This should dynamically be updated as the game goes through different players/phases and be visible at all times during

game play. You must deliver a driver that demonstrates that (1) the information displayed by the phase view is cleared every time the phase is changing (2) the phase view is displaying the correct player:phase information as soon as the phase changes; (3) the phase view displays relevant information which is different for every phase. The Observer and Observable classes code must be implemented in a new **`GameObservers.cpp/GameObservers.h`** file.

Game Statistics Observer
Using the Observer design pattern, implement a view that displays some useful statistics about the game, the minimum being a "players world domination view" that shows using some kind of graph or table depicting what percentage of the world is currently being controlled by each player. This should dynamically be updated as the map state changes and be visible at all times during game play. You must deliver a driver that demonstrates that (1) the game statistics view updates itself every time a country has been conquered by a player; (2) the game statistics updates itself when a player has been eliminated and removes this player from the view; (3) as soon as a player owns all the countries, the game statistics view updates itself and displays a celebratory message. The Observer and Observable classes code must be implemented in a new **`GameObservers.cpp/GameObservers.h`** file duo.

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 *driver* (i.e. a `main` function or a free function called by the main function) for each part that allows the marker to observe the execution of each part during the lab demonstration. Each driver should simply create the components described above and demonstrate that they behave as mentioned above.

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.*	**10 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.*	**4 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)**