Concordia University
Department of Computer Science
and Software Engineering

Advanced program design with C++
COMP 345 --- Fall 2019

Team project assignment #2

Deadline:	November 2nd, 2019
Evaluation:	8% 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 1 must stay in the same files as specified in assignment #1:

- Map implementation: `Map.cpp/Map.h`.
- Map loader implementation: `MapLoader.cpp/MapLoader.h`.
- Dice rolling facility implementation: `Dice.cpp/Dice.h`.
- Player implementation: `Player.cpp/Player.h`.
- Card hand and deck implementation: `Cards.cpp/Cards.h`.

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 (2-6 players). The code should then use the map loader to load the selected map, create all the players, assign dice rolling facilities to the players, 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. This must be implemented in a single `.cpp/.h` file duo named `GameEngine.cpp/GameEngine.h`.

Part 2: Game play: startup phase

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

1. The order of play of the players in the game is determined randomly.
2. All countries in the map are randomly assigned to players one by one in a round-robin fashion.
3. Players are given a number of armies (A), to be placed one by one in a round-robin fashion on some of the countries that they own, where A is:
    - If 2 players, A=40
    - If 3 players, A=35
    - If 4 players, A=30
    - If 5 players, A=25
    - If 6 players, A=20

You must deliver a driver that demonstrates that 1) all countries in the map have been assigned to one and only one player; 2) all players have eventually placed the right number of armies on their own countries after army placement is over. 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 game of Risk. During the main game loop, proceeding in a round-robin fashion as setup in the startup phase, every player is given the opportunity to do sequentially each of the following actions during their turn:

1. Reinforcements phase
2. Attack phase
3. Fortification phase

This loop shall continue until only one player controls all the countries in the map, at which point a winner is announced and the game ends. You must deliver a driver that demonstrates that 1) every player gets turns in a round-robin fashion and that their `reinforcement()`, `attack()` and `fortification()` methods are called 2) the game ends when a player controls all the countries (the driver should explicitly give all the countries to one player, i.e. no real code for battles needs to be executed). This must be implemented in a single **`.cpp/.h`** file duo named **`GameEngine.cpp/GameEngine.h`**.

Part 4: Main game loop: reinforcement phase

Provide a group of C++ classes that implement the reinforcement phase following the official rules of the game of Risk. In the reinforcement phase, the player gets a number of armies (A) to place on its countries, where A is:

- Number of countries owned on the map, divided by 3 (rounded down), with a minimum of 3.
- Continent-control value of all continents totally controlled by that player.
- Armies resulting in card exchange, if possible. If a player owns more than 5 cards, it must exchange cards (exchanging cards should be done inside the Hand's `exchange()` method).

The player must then place all these armies on some of the countries it owns, as it sees fit (for now, it does not matter which). This code must be implemented within the `reinforce()` method of the player class in the **`Player.cpp/Player.h`** files. You must deliver a driver that demonstrates that 1) a player receives the right number of armies in the reinforcement phase (showing different cases); 2) the player has effectively placed this exact number of new armies somewhere on the map by the end of the reinforcement phase.

Part 5: Main game loop: attack phase

Provide a group of C++ classes that implement the attack phase following the official rules of the game of Risk. In this phase, the player is allowed to declare a series of attacks to try to gain control of additional countries, and eventually control the entire map. The attack phase follows the following loop:

- The player decides if it will attack or not. If not, the attack phase is over.
- The player selects one of its countries to attack from, and one of the neighbors of this country to attack (i.e. the attacked country belongs to another player). The attacking country must have at least 2 armies on it.
- The attacker and defender players choose the number of dice to roll for their attack/defense. The attacker is allowed 1 to 3 dice, with the maximum number of dice being the number of armies on the attacking country, minus one. The defender is allowed 1 to 2 dice, with the maximum number of dice being the number of armies on the defending country.
- The dice are rolled for each player and sorted, then compared pair-wise. For each pair starting with the highest, the player with the lowest roll loses one army. If the pair is equal, the attacker loses an army.
- If the attacked country runs out of armies, it has been defeated. The defending country now belongs to the attacking player. The attacker is allowed to move a number of armies from the attacking country to the attacked country, in the range [1 to (number of armies on attacking country -1)].
- The player is allowed to initiate any number of attacks per turn, including 0.

This code must be implemented within the `attack()` method of the player class in the **`Player.cpp/Player.h`** files. You must deliver a driver that demonstrates that 1) only valid attacks can be declared (i.e. valid attacker/attacked country); 2) only valid number of dice can be chosen by the attacker/defender; 3) given known dice values, that the right number of armies are deducted on the attacker/defender; 4) the attacker is allowed to initiate multiple attacks, until it declares that it does not want to attack anymore.

Part 6: Main game loop: fortification phase

Provide a group of C++ classes that implement the fortification phase following the official rules of the game of Risk. In the fortification phase, the player is allowed to move a number of armies (X) from one of its countries (the source country) to one of its neighboring countries that it also owns (the target country). X must be in the range [1 to (number of armies on the source country - 1)]. This code must be implemented within the `fortify()` method of the player class in the **`Player.cpp/Player.h`** files. You must deliver a driver that demonstrates that 1) only valid countries can be selected as source/target; 2) only a valid number of armies can be moved; 3) the right number of armies is effectively moved.

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, 6). 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 ENCS Electronic Assignment Submission system under the category “programming assignment 1”. 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 the labs.

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 game of Risk.*	**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)**