Concordia University
Department of Computer Science
and Software Engineering

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

Team project assignment #3

Deadline:	December 1st, 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 3 must stay in the same files as specified in assignment #1 and 2:

- 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`**
- Game Engine implementation: **`GameEngine.cpp/GameEngine.h`**
- Game Observers implementation: **`GameObservers.cpp/GameObservers.h`**

Specific design requirements

1. All data members of user-defined class types must be of pointer type.
2. If smart pointers are used, the data members of user-defined class types 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.
5. All functions' implementation must be provided in the **`.cpp`** file (i.e. no inline functions are allowed).
6. All classes must implement a correct copy constructor, assignment operator, destructor, and stream insertion operator.
7. Memory leaks must be avoided.
8. Code must be documented using comments (user-defined classes, methods, free functions, operators).
9. 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: Player Strategy Pattern

Using the Strategy design pattern, implement different kinds of players that make different decisions during the issuing orders phase by implementing different versions of **`issueOrder()`**, **`toAttack()`** and **`toDefend()`** in different ConcreteStrategy classes, whose respective behaviors are described below. The kinds of players are: (1) human player that requires user interaction to make decisions, (2) an aggressive computer player that focuses on attack (deploys or advances armies on its strongest country, then always advances to enemy territories until it cannot do so anymore), (3) a benevolent computer player that focuses on protecting its weak countries (deploys or advances armies on its weakest countries, never advances to enemy territories), (4) a neutral player that never issues any order. You must deliver a driver that demonstrates that (1) different players can be assigned different strategies that lead to different behavior for using the strategy pattern; (2) the strategy adopted by a player can be changed dynamically during play, (3) the human player makes decisions according to user interaction, and computer players make decisions automatically, which are both implemented using the strategy pattern. The code for the Strategy class and its ConcreteStrategies must be implemented in a new **`PlayerStrategies.cpp/PlayerStrategies.h`** file duo. In order to have a real adapter implementation, the following conditions must be present in your resulting implementation:

- Your **`Player`** class does not have subclasses that implement different behaviors.
- You have a **`PlayerStrategy`** class that is not a subclass of the **`Player`** class.
- For each strategy as described above, you have a ConcreteStrategy class: **`HumanPlayerStrategy`**, **`AggressivePlayerStrategy`**, **`BenevolentPlayerStrategy`**, and **`NeutralPlayerStrategy`** that are subclasses of the **`PlayerStrategy`** class.
- Each of the ConcreteStrategy classes implement their own version of the **`issueOrder()`**, **`toAttack()`**, and **`toDefend()`** methods.
- The **`Player`** class contains a data member of type **`PlayerStrategy`**.
- The **`issueOrder()`**, **`toDefend()`**, and **`toAttack()`** methods of the player do not implement behavior and simply delegate their call to the **`PlayerStrategy`** object member of the **`Player`**.

Part 2: File Reader Adapter

Using the Adapter pattern, implement a new file reader that reads maps written in the Conquest map format (http://www.windowsgames.co.uk/conquest.html). You must deliver a driver that demonstrated that the game can use either the original reader to read Domination map files, or use the reader Adapter. The Adapter class code must be in the already existing **`MapLoader.cpp/MapLoader.h`** file duo, as specified in assignment #1. In order to have a real adapter implementation, the following conditions must be present in your resulting implementation:

- Your existing file reader class (i.e. the class that contains code to read the domination files) is left unchanged. It should not implement the code for reading the Conquest map files.
- You have a new **`ConquestFileReader`** file reader class that is not a subclass of your initial file reader class. This class implements the code that reads the Conquest map files.
- You have a new **`ConquestFileReaderAdapter`** class created that is a subclass of your initial file reader class. It should contain a **`ConquestFileReader`** data member, that overrides the method called to read Conquest files from your initial file reader class, whose implementation delegates the reading of the file to the file reading method of its **`ConquestFileReader`** data member.

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). 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 on the moodle page for the course 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 on zoom during the 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)**