Concordia University
Department of Computer Science
and Software Engineering

Advanced program design with C++
COMP 345 --- Fall 2021

Team project assignment #3

Deadline:	December 3st, 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.

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`**
   - Command processing: **`CommandProcessing.cpp/CommandProcessing.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`**
   - Game logging: **`LoggingObserver.cpp/LoggingObserver.h`**

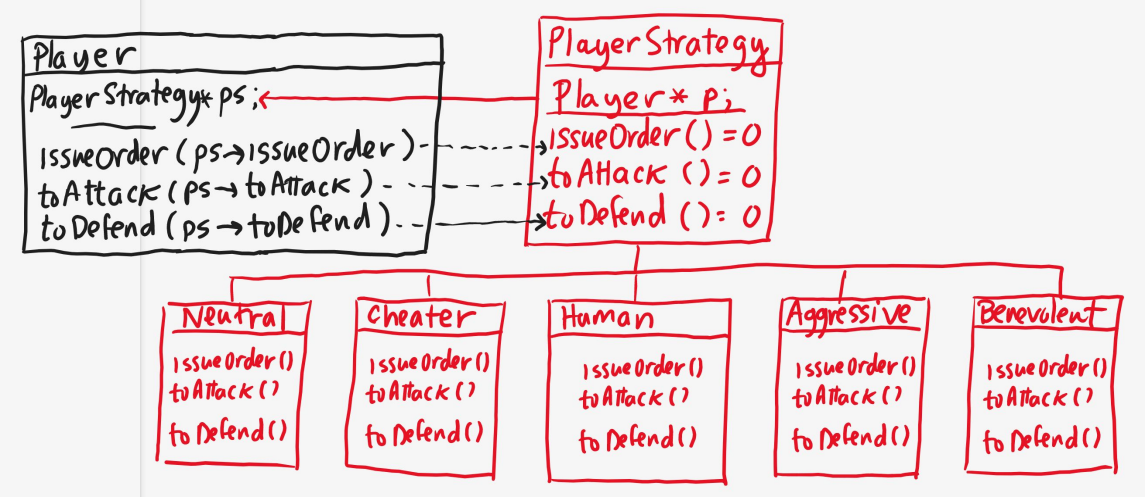
Part 1: Player strategy pattern

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:

- **Human player:** requires user interactions to make decisions.
- **Aggressive player:** 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).
- **Benevolent player:** computer player that focuses on protecting its weak countries (deploys or advances armies on its weakest countries, never advances to enemy territories).
- **Neutral player:** computer player that never issues any order. If a Neutral player is attacked, it becomes an Aggressive player.
- **Cheater player:** computer player that automatically conquers all territories that are adjacent to its own territories (only once per turn).

You must deliver a driver that demonstrates that (1) different players can be assigned different strategies that lead to different behavior using the Strategy design 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 strategy 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`** abstract 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 corresponding methods in the **`PlayerStrategy`** member of the **`Player`**.



Design of the implementation of the different player behaviors using the Strategy pattern

Part 2: Tournament mode

During the start game state, a new tournament command can be entered by the user, which triggers the Tournament Mode. While in the tournament mode, the game should proceed without any user interaction and show the results of the tournament at the end. The tournament command is the following:

`tournament -M <listofmapfiles> -P <listofplayerstrategies> -G <numberofgames> -D <maxnumberofturns>`

This command lets the user choose the parameters of the tournament, i.e.: *M = 1 to 5* different maps, *P = 2 to 4* different computer players strategies, *G = 1 to 5* games to be played on each map, and *D = 10 to 50* maximum number of turns for each game. Once the command is entered and validated, the tournament is automatically played by playing G games on each of the M different maps between the chosen computer player strategies. In order to minimize run completion time, each game should be declared a draw after D turns. Once started, the tournament plays all the games automatically without user interaction. At the end of the tournament, a report of the results should be output to the log file, e.g.

```
Tournament mode:
M: Map1, Map2, Map3
P: Aggressive, Benevolent, Neutral, Cheater.
G: 4
D: 30

Results:
```

	Game 1	Game 2	Game 3	Game 4
Map 1	Aggressive	Neutral	Cheater	Cheater
Map 2	Cheater	Draw	Cheater	Aggressive
Map 3	Cheater	Aggressive	Cheater	Draw

You must deliver a driver that demonstrates that (1) the tournament command can be processed and validated by the CommandProcessor, and executed by the GameEngine, resulting in a tournament being played as described above. The code for the processing of the tournament command must be implemented in the existing **`CommandProcessing.cpp/CommandProcessing.h`** file duo. The code for the execution of the tournament must be implemented in the existing **`GameEngine.cpp/GameEngine.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). 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 3”. 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: **2 pts (indicator 4.1)**

Mark deductions: during the presentation or code review it is found that the

implementation does not follow the rules of the Warzone game.

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)**