Concordia University
Department of Computer Science and Software Engineering

Advanced Programming Practices
SOEN 6441 --- Fall 2023

Project Build 2 Grading

Deadline:	Submission of the project: Nov 8, 2023, 23:55pm Presentation: Nov 9 and 10, 2023, time schedule will be posted soon.
Evaluation:	Intermediate delivery 2: 20%
Late submission: Teams:	not accepted the project is a team assignment

Instructions for Incremental Code Build Presentation

You must deliver an operational version demonstrating a subset of the capacity of your system. This is about demonstrating that the code build is effectively aimed at solving specific project problems or completely implementing specific system features. The code build must not be just a "portion of the final project", but rather be something useful with a purpose on its own, that can be demonstrated by its operational usage.

The presentation should be organized as follows:
1. Brief presentation of the goal of the build.
2. Brief presentation of the architectural design of your project.
3. Demonstration of the functional requirements as listed on the following grading sheet.
4. Demonstration of the use of tools as listed on the following grading sheet.

You are graded according to how effectively you can demonstrate that the features are implemented. If you cannot really demonstrate the features through execution, you will have to prove that the features are implemented by explaining how your code implements the features, in which case you will get only partial marks.

During your presentation, you have to demonstrate that you are well-prepared for the presentation, and that you can easily provide clear explanations as questions are asked about the functioning of your code, or your required usage of the tools/techniques.

Before the presentation starts, you have to submit your current project code base to Moodle.

Grading

Presentation		**5**
Effectiveness, structure and demonstrated preparation of the presentation		2
Fluid exposition of knowledge of code base/clarity of explanations		3
Functional Requirements		**30**
Refactor your code to use the **State pattern** to implement the phases of the application, including the phases in the map editor, and the game play). The game play phase must be divided into the following phases: startup, issue order, and order execution phases. The context class of the State pattern must be you GameEngine class and the State class must be a new class named Phase.		3
If a command is invalid, a proper message should be given to the user that explains why the command is invalid.		1
If a command has options, each option can be used from 0 to multiple times in a single command.		1
Map editor		**1**

Map editor commands (as specified in Build #1):
`editcontinent -add` ***`continentID continentvalue`*** `-remove` ***`continentID`***
`editcountry -add` ***`countryID continentID`*** `-remove` ***`countryID`***
`editneighbor -add` ***`countryID neighborcountryID`*** `-remove` ***`countryID neighborcountryID`***`showmap`
`showmap`
`savemap` ***`filename`***
`editmap` ***`filename`***
`validatemap`
		1

Game play		**25**
Implementation of a game log file using the **Observer pattern**. For every action taken during the game (e.g. a command is executed, or an order is issued or executed), a LogEntryBuffer object is filled with information about the effect of the action. The LogEntryBuffer class should be an Observable. There should be a corresponding Observer that writes the content of the LogEntryBuffer to a log file when it is changed. The end result should be that the user can clearly see all the actions that happened during a game by looking at the file. The log file should clearly identify every phase of the game.		3
Refactor your code to use the **Command pattern** to implement the Orders. The Command class must be the Order class, the Invoker Class is the Player, and the Client class is the GameEngine. The orders are created as the player executes its issue_order() method, and the orders are executed when the GameEngine gets the Player's orders from the Players using the next_order() method, then executes the orders by calling the execute() method of the Order.		3
All Players have a hand of cards. Players start with no cards. Every turn, if a Player conquered at least one Country in their turn, they receive one random card (i.e. maximum one card per Player per turn).		1

Game play commands (as specified in Build #1):
`showmap`
		1

Game startup phase		**1**

Startup phase commands:
`loadmap` ***`filename`***
`gameplayer -add` ***`playername`*** `-remove` ***`playername`***
`assigncountries`
		1

Reinforcement phase		**1**
At the beginning of every turn a Player is given a number of reinforcement armies, calculated according to the Warzone rules.		1
Order creation phase		**11**
Deploy order command: `deploy` ***`countryID numarmies`***		1
Advance order command: `advance` ***`countrynamefrom countynameto numarmies`***		2
Bomb order command (requires bomb card): `bomb` ***`countryID`***		2
Blockade order command (required blockade card): `blockade` ***`countryID`***		2
Airlift order command (requires the airlift card): `airlift source`***`countryID targetcountryID numarmies`***		2
Diplomacy order command (requires the diplomacy card): `negotiate player`***`ID`***		2
Order execution phase		**3**
During the order execution phase, the GameEngine asks each Player for their next order using the next_order() method, then executes the order using the execute() method of the Order.		3

Programming process		**15**
Architectural design—short document including an architectural design diagram. Short but complete and clear description of the design, which should break down the system into cohesive modules. The architectural design should be reflected in the implementation of well-separated modules and/or folders. The document should explain how the State, Command, and Observer patterns were incorporated in the design.		2
Software versioning repository— well-populated history with at least 50 commits, distributed evenly among team members, as well as evenly distributed over the time allocated to the build. A tagged version should have been created for build 1 and 2. Use of a continuous integration solution that applies the following operation when code is pushed onto the repository: (1) project successfully compiles (2) all unit tests successfully pass (3) javadoc is compiled and reported as complete.		3
API documentation—completed for all files, all classes and all methods, including private members. All test classes and test cases are properly documented. Use the java command line option `-Xdoclint` to prove that the Javadoc is complete.		2
Unit testing framework—at least 30 relevant test cases testing the most important aspects of the code. Must include tests for: (1) map validation – including map and continents being connected graphs; (2) reading an invalid map file; (3) validation of a correct startup phase; (4) calculation of number of reinforcement armies; (5) various test for the order validation upon execution – including source/target validation for every order, conquering a country, moving of armies in the conquered country after conquering it, and end of game when conquering all countries. There must be a 1-1 relationship between implementation classes and test classes. Presence of a single test suite from which to run all test cases, as well as one test suite for each folder/package in the implementation.		3
Coding standards—Consistent use of the coding conventions described below		2
Refactoring—demonstrate that you have applied a refactoring operation after build #1. Explain how you came up with a list of potential refactoring targets, how you chose the refactoring targets among the list, and explain the 5 refactoring operations that you have applied. Refactoring operations must be on code that has some unit tests in place. Short document according to the description below.		3
Total		**50**

Coding conventions

Naming conventions
- class names in CamelCase that starts with a capital letter
- data members start with **`d_`**
- method parameters start with **`p_`**
- local variables start with **`l_`**
- global variables in capital letters
- static members start with a capital letter, non-static members start with a lower case letter

Code layout
- consistent layout throughout code (use an IDE auto-formatter)

Commenting convention
- javadoc comments for every class and method
- long methods (more than 10 lines) are documented with comments for procedural steps
- no commented-out code

Project structure
- one folder for every module in the high-level design
- tests are in a separate folder that has the exact same structure as the code folder
- 1-1 relationship between tested classes and test classes

Refactoring document

Potential refactoring targets.
- Explain how you have identified the potential refactoring targets.
- List of at least 15 refactoring targets.

Actual refactoring targets.
- Give the rationale explaining how you have chosen or excluded some of the potential refactoring targets.
- List 5 actual refactoring targets.

Refactoring operations

For each of the 5 actual refactoring targets.
- List all the tests that apply to the class involved in the refactoring operation.
- If not enough tests exist for these classes, add more and list them.
- Explain why this refactoring operation was deemed necessary.
- Give a before/after depiction of the refactoring operation.