Concordia University
Department of Computer Science
and Software Engineering

Advanced Programming Practices
SOEN 6441 --- Winter 2021

Project Build 1 Grading

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.

Identification

Team	Evaluator	Signature	Date

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: The game must include a command prompt that is available throughout the game (even if you have a GUI). All the commands should be validated and give proper feedback on their effect or invalidity. Each of the commands should only work in the game phase for which they are designed for. The commands should use the exact same syntax as defined below. Some commands have options preceded by a dash. Options may be used from zero to multiple times in a command. Command parameters are noted in italics.		**30**
Map editor		**10**
User-driven creation/deletion of map elements: country, continent, and connectivity between countries. Map editor commands: **`editcontinent -add` *`continentID continentvalue`* `-remove` *`continentID`*** **`editcountry -add` *`countryID continentID`* `-remove` *`countryID`*** **`editneighbor -add` *`countryID neighborcountryID`* `-remove` *`countryID neighborcountryID`***		2
Display the map as text **`showmap`** (show all continents and countries and their respective neighbors)		2
Save a map to a text file exactly as edited (using the "domination" game map format). Map editor command: **`savemap` *`filename`***		2
Load a map from an existing "domination" map file, or create a new map from scratch if the file does not exist. Map editor command: **`editmap` *`filename`***		2
Verification of map correctness. The map should be automatically validated upon loading and before saving (at least 3 types of incorrect maps). The **`validatemap`** command can be triggered anytime during map editing. Map editor command: **`validatemap`**		2
Game play		**20**
Implementation of a GameEngine class implementing and controlling the game phases according to the Warzone rules.		2
Game play command: **`showmap`** (show all countries and continents, armies on each country, ownership, and connectivity in a way that enables efficient game play)		2
Startup phase		
Game starts by user selection of a user-saved map file, which loads the map as a connected directed graph. Startup phase command: **`loadmap` *`filename`***		2
User creates the players, then all countries are randomly assigned to players. Startup phase commands: **`gameplayer -add` *`playername`* `-remove` *`playername`*** **`assigncountries`**		2
There must be a Player class that must hold (among other things) a list of Country objects that are owned by the Player and a list of Order objects that have been created by the Player during the current turn, and has a method "issue_order()" (no parameters, no return value) whose function is to add an order to the list of orders held by the player when the game engine calls it during the *issue orders phase*. The player class must also have a "next_order()" (no parameters) method that is called by the GameEngine during the *execute orders phase* and returns the first order in the player's list of orders, then removes it from the list.		4
Main game loop		
Loop over each player for the *assign reinforcements*, *issue orders* and *execute orders* main game loop phases		1
Main game loop: assign reinforcements phase		
Assign to each player the correct number of reinforcement armies according to the Warzone rules.		1
Main game loop: issue orders phase		
The GameEngine class calls the issue_order() method of the Player. This method will wait for the following command, then create a deploy order object on the player's list of orders, then reduce the number of armies in the player's reinforcement pool. The game engine does this for all players in round-robin fashion until all the players have placed all their reinforcement armies on the map. Issuing order command: **`deploy` *`countryID num`*** (until all reinforcements have been placed)		3
Main game loop: execute orders phase		
The GameEngine calls the next_order() method of the Player. Then the Order object's execute() method is called, which will enact the order. The effect of a deploy order is to place ***`num`*** armies on the country ***`countryID`***.		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.		3
Software versioning repository—well-populated history with dozens of 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. 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
Javadoc API documentation—completed for all files, all classes and all methods.		3
Unit testing framework—at least 10 relevant test cases testing the most important aspects of the code. Must include tests for: (1) map validation – map is a connected graph; (2) continent validation – continent is a connected subgraph; (3) calculation of number of reinforcement armies; (4) player cannot deploy more armies that there is in their reinforcement pool.		3
Coding standards—Consistent use of the coding conventions described 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