# Concordia University
# Department of Computer Science
# and Software Engineering

# Advanced Programming Practices
# SOEN 6441 --- Fall 2018

# Project Build 3 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 the Electronic Assignment Submission System under “project 3”.

## 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** | | **30** |
| **Map editor** | | **5** |
| User-driven creation of map elements, such as country, continent, and connectivity between countries. | | 1 |
| Saving a map to a file exactly as edited (using the "conquest" game map format). Loading a map from an existing "conquest" map file, then editing the map, or create a new map from scratch. | | 1 |
| Successfully load the `3dCliff.map` and `World.map` file. Reject loading of the `TwinVolcano.map` file by correctly verifying that continents are connected subgraphs. | | 3 |
| **Game Play** | | **25** |
| Implementation of a "phase view" and "players world domination view" using the Observer pattern. | | 1 |
| The "phase view" show enough information to clearly demonstrate that the game phases and rules are properly implemented using the Risk game rules. | | 3 |
| Implementation of the different player behaviors using the Strategy pattern, where the strategies implement different versions of the reinforcement, attack and fortification as methods of the Player class (see Player Behavior Strategies below). | | 4 |
| **Single game mode:** Game starts by user selection of a user-saved map file, then loads the map as a connected graph. User chooses the number and behavior of players (see Player Behavior Strategies below). The game proceeds until one of the players has conquered the whole map. If no human player is selected, the game proceeds fully automatically without any user interaction. | | 2 |
| **Tournament Mode:** When the game starts, provide an option for a Tournament Mode (see "Tournament" below). The tournament should proceed without any user interaction and show the results of the tournament at the end. | | 6 |
| **Startup phase:** All countries are randomly assigned to players. Players are allocated a number of initial armies, depending on the number of players. In round-robin fashion, the players place one by one their given armies on their own countries. | | 1 |
| **Reinforcement phase:** Calculation of correct number of reinforcement armies according to the Risk rules. Players place reinforcement armies on the map. Reinforcement ends automatically when all armies have been placed. Implementation of a "card exchange view" using the Observer pattern. | | 1 |
| **Attack phase:** Player can declare an attack by selecting attacker and attacked country. Attacker and attacked player decide how many dice to roll. Proper number of armies are deducted from attacker/defender country. If defender is conquered, attacker can move any number of its armies in the conquered country. If it results in conquering the whole map, the attacker is declared the winner and the game ends. Player may decide to attack or not to attack again. If attack not possible, attack automatically ends. | | 2 |
| **Fortification phase:** Implementation of a valid fortification move according to the Risk rules. Fortification ends automatically when the armies have been moved. | | 1 |
| **Game Save/Load:** As a game is being played, allow the user to save the game in progress to a file, and allow the user to load the game in exactly the same state as saved. | | 4 |

Notes

| 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. | | 2 |
| **Software versioning repository**—well-populated history with many dozens of commits, distributed evenly among team members, as well as evenly distributed over the time allocated to the build and the whole project. A tagged version should have been created for build 1 and 2. | | 3 |
| **API documentation**—completed for all files, all classes and all methods, including private members. All test classes and test cases properly documented. Run Javadoc to demonstrate that the documentation is complete for the whole code. | | 2 |
| **Unit testing framework**—at least 40 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 attack phase – including attacker/defender validation, valid move after conquering, and end of game; (6) validation of a correct fortification phase (7) saving/loading a game (8) tournament mode. There must be a 1-to-1 relationship between implementation classes and test classes. Presence of a single test suite from which to run all test cases. | | 3 |
| **Coding standards**—documented description of coding standard used. Consistent and proper use of code layout, naming conventions and comments, absence of "commented out" code, presence of comments to describe the process followed by long methods. | | 2 |
| **Refactoring**—demonstrate that you have applied a refactoring operation after build #2. Explain how you came up with a list of potential refactoring targets, how you chose the refactoring targets among the list, and explain at least 3 refactoring operations that you have applied. Refactoring operations must be on code that has some unit tests in place. | | 3 |
| **Total** | | **50** |

Notes

**Player Behavior Strategies**

- A **human** player that requires user interaction to make decisions.
- An **aggressive** computer player strategy that focuses on attack (reinforces its strongest country, then always attack with it until it cannot attack anymore, then fortifies in order to maximize aggregation of forces in one country).
- A **benevolent** computer player strategy that focuses on protecting its weak countries (reinforces its weakest countries, never attacks, then fortifies in order to move armies to weaker countries).
- A **random** computer player strategy that reinforces random a random country, attacks a random number of times a random country, and fortifies a random country, all following the standard rules for each phase.
- A **cheater** computer player strategy whose `reinforce()` method doubles the number of armies on all its countries, whose `attack()` method automatically conquers all the neighbors of all its countries, and whose `fortify()` method doubles the number of armies on its countries that have neighbors that belong to other players.

**Tournament Mode**

A tournament starts with the user choosing 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, D = 10 to 50 maximum number of turns for each game. A tournament is then 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 displayed, e.g.

M: Map1, Map2, Map3
P: Aggressive, Benevolent, Random, Cheater.
G: 4
D: 30

| | Game 1 | Game 2 | Game 3 | Game 4 |
|---|---|---|---|---|
| Map 1 | Aggressive | Random | Cheater | Cheater |
| Map 2 | Cheater | Draw | Cheater | Aggressive |
| Map 3 | Cheater | Aggressive | Cheater | Draw |