Concordia University
Department of Computer Science
and Software Engineering

Advanced Programming Practices
SOEN 6441 --- Fall 2018

Project Build 2 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 2”.

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.		2
Saving a map to a file exactly as edited (using the “conquest” game map format).		1
Loading a map from an existing “conquest” map file, then editing the map, or create a new map from scratch.		1
Verification of map correctness upon loading and before saving (at least 3 types of incorrect maps, including unconnected map and unconnected continent).		1
Game Play		**25**
Implementation of a “phase view” using the Observer pattern. The phase view should display: (1) the name of the game phase currently being played (2) the current player’s name (3) information about actions that are taking place during this phase. The phase view should be cleared at the beginning of every phase.		3
Implementation of a “players world domination view” using the Observer pattern. The players world domination view should display (1) the percentage of the map controlled by every player (2) the continents controlled by every player (3) the total number of armies owned by every player.		3
Implementation of the reinforcement, attack and fortification as methods of the Player class.		3
Startup phase		**2**
Game starts by user selection of a user-saved map file, then loads the map as a connected graph. User chooses the number of players, then 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.		2
Reinforcement phase		**4**
Calculation of correct number of reinforcement armies according to the Risk rules. Players place reinforcement armies on the map.		1
Implementation of a “card exchange view” using the Observer pattern. The card exchange view should be created only during the reinforcement phase. It should display all the cards owned by the current player, then allow the player to select some cards to exchange. If the player selects cards, they are given the appropriate number of armies as reinforcement. The player can choose not to exchange cards and exit the card exchange view. If the player own 5 cards or more, they must exchange cards. The cards exchange view should cease to exist after the cards exchange.		3
Attack phase		**8**
Player can declare an attack by selecting attacker and attacked country.		1
Attacker and attacked player decide how many dice to roll.		1
Proper number of armies are deducted from attacker/defender country during the attack(s).		2
If defender is conquered, attacker can move any number of its armies in the conquered country (see the Risk rules). If it results in conquering the whole map, the attacker is declared the winner and the game ends.		1
Player may decide to attack or not to attack again. If attack not possible, attack automatically ends.		1
Implementation of an “all-out” mode, where once the attacker and attacked country have been identified, the attack proceeds with maximum number of dice and end only when either the attacker conquers the attacked, or the attacker cannot attack anymore.		2
Fortification phase		**2**
Implementation of a valid fortification move according to the Risk rules.		2

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 25 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. 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, as well as one test suite for each folder/package in the implementation.		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 #1. 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