Concordia University
Department of Computer Science
and Software Engineering

Advanced Programming Practices
SOEN 6441 --- Winter 2024

Project Build 3

Code submission deadline:	April 9, 2024
Presentation dates:	March 10-12, 2024
Evaluation:	20% of final grade
Late submission:	not accepted

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**
Map save/load		**4**
Refactor your code to use the **Adapter** pattern to enable the application to read/write from/to a file using the "conquest" game map format (http://www.windowsgames.co.uk/conquest_maps.html). The application should be able to decide to use either the original "domination" file reader or the "conquest" file reader adapter when a file is opened, depending on the file type. When a map file is saved, the user should be given the option as to which file format to use as output. At any point in time, only one of the original file reader or the adapter should be instantiated. The Adapter class should be designed so that if exception are thrown while reading the file, they should be handled internally by the Adapter and never thrown to the Game Controller.		5
Map editing/loading/saving		**2**
Saving a map using the either of the game map formats. Loading a map from an existing map file in either format, then editing the map, or create a new map from scratch.		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		**24**
Refactor your current user-driven player code so that the implementation of the Player's issueOrder() method's behavior is using the **Strategy** pattern. Then, during the main development phase, implement different computer player behaviors using the Strategy pattern, where the strategies provide varying behavior that support the Player class to expose varying behavior when executing the issueOrders() method (see Player Behavior Strategies below). The Strategy class should be designed so that if exception are thrown while reading the file, they should be handled internally by the Adapter and never thrown to the Game Controller.		5
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). Cards are used in the issueOrders() Player method to create orders.		1
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. Tournament game play commands: **tournament -M *listofmapfiles* -P *listofplayerstrategies* -G *numberofgames* -D *maxnumberofturns***		2
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. Game save/load commands: **savegame *filename*** (see below) **loadgame *filename*** (see below)		3
Game startup phase		**2**
Startup phase commands: **loadmap *filename*** **gameplayer -add *playername* -remove *playername*** **assigncountries**		2
Reinforcement phase		**2**
A the beginning of every turn a Player is given a number of reinforcement armies, calculated according to the Warzone rules.		2
Order creation phase		4
Implementation of the human, aggressive, benevolent, random, and cheater behaviors. The aggressive, benevolent, random, and cheater players automatically make decisions and create orders without any kind of user interaction. The human player is relying on user commands to make decisions and create orders.		2
Human player order creation commands: **deploy *countryID numarmies*** **advance *countrynamefrom countynameto numarmies*** **bomb *countryID*** **blockade *countryID*** **airlift *sourcecountryID targetcountryID numarmies*** **negotiate *playerID***		2
Order execution phase		**2**
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.		2

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, Observer, Adapter, and Strategy patterns were incorporated in the design.		2
Software versioning repository— well-populated history with at least 75 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, 2 and 3. 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 50 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 execution phase – including attacker/defender validation, valid move after conquering, and end of game; (6) 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— Consistent use of the coding conventions described below.		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 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.

Player Behavior Strategies

- A **human** player that requires user interaction to make decisions.
- An **aggressive** computer player strategy that focuses on centralization of forces and then attack, i.e. it deploys on its strongest country, then always attack with its strongest country, then moves its armies in order to maximize aggregation of forces in one country.
- A **benevolent** computer player strategy that focuses on protecting its weak countries (deploys on its weakest country, never attacks, then moves its armies in order to reinforce its weaker country).
- A **random** computer player strategy that deploys on a random country, attacks random neighboring countries, and moves armies randomly between its countries.
- A **cheater** computer player strategy whose `issueOrder()` method conquers all the immediate neighboring enemy countries, and then doubles the number of armies on its countries that have enemy neighbors. Note that in order to achieve this, the cheater's strategy implementation will still be called when the issueOrder() method, but will not end up creating orders, but rather implement the above-stated behavior by directly affecting the map during the order creation phase.

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