COSC 3P91
Assignment 3

Due date: March 23rd, 2023 at 23:59 (11:59pm)
Delivery method: the student needs to delivery the assignment only through D2L.
Delivery contents: document with a description and Java codes (see Submission instructions).
Attention: check the Late Assignment Policy.

Assignment Overview

The assignment consists of modifying the current implementation of the Strategy Game. The game makes use of a console as an interface for the interactions of a player. The console is basically System.in/out for I/O communication with the player. Ideally, your current implementation, obtained from Assignment 2, will form the basis of your modifications. The modifications aim to benefit transparency, encapsulation, and extensibility. Even though the patterns and aspects should be contemplated in initial parts of the design, implementing them at this stage will demand efforts that require fluent comprehension of the 'application' and Design Patterns.

The Game

The **game** is a simplified village war strategy game where each player must build, develop, and upgrade a village. Also, the village must be able to defend from attacks and perform attacks. The concept is very similar to several online mobile games.

Each village contains the following buildings:

- A **Village Hall**;
  - The village hall may have levels/stages that condition the upgrades available to all other buildings.
- Food production: **Farms**;
  - Each farm is responsible for feeding a population size. The number of farms restricts the population size;
  - Suggestion (extra): Farms may have levels that allow to have a larger population size fed per farm.
- Mining: **Gold Mines**, **Iron Mines**, and **Lumber Mills**;
  - Suggestion (extra): You may just assume that the mining/extraction buildings are placed in areas with plenty of ore and lumber;
  - Suggestion (extra): Even those buildings have costs to be built. You may define precedence of requirements: (i) lumber mill can be built from scratch; (ii) iron mines need lumber to be build; and (iii) gold mines need lumber and iron to be built;
  - Suggestion (extra): You can assume resources are unlimited, or you may define a limit for extracting resources – after that, the mine/mill depletes.
- Defences: **Archer Towers** and **Cannons**.
  - They can inflict damage to attacking armies, or army units. Depending on the level to details when simulating attacks. For instance, towers may have different damage than cannons do and may have larger defence range;
  - Suggestion (extra): Defence unities may be upgraded to inflict more damage;
  - Suggestion (extra): You may add dependencies for enabling defence unities or upgrading them, such as a blacksmith (likewise for army units).

  - Suggestion (extra): workers, miners, and collectors may help with defence but much lower damage and hitpoints.

Each village is limited by a maximum number of Inhabitants, and they can be the following:

- **Workers**: food production and construction;
  - A village must have idling builders for constructions and upgrades to start;
- **Miners/Collectors** (gold, iron, and wood);
  - Each miner and collector has an average or maximum "production" capacity;
  - Each mine or mill has a maximum number of miners and collectors.
- Army: **Soldiers**, **Archers**, **Knights**, and **Catapults**.
  - They can inflict damage to any building, including defence units.
  - Each army unit has a different damage and attack range.
  - Suggestion (extra): You may simulate the attack at a fine grain level – each army unit, depending on the level to details when simulating attacks. For instance, soldiers and archers may be more effective against defence units while catapults may inflict more damage against non-defence unities.;
  - Suggestion (extra): You may impose an army size or army composition limitations. For instance, an army can only be succeed in an attack if it is composed of at least one type of attack unit.

Every building, worker, miner, collector, and army unit has a production cost. All of them require feeding from farms (population size). Any construction, upgrade, and army unit requires amounts of Gold, Iron, and Wood - the quantity resources differ based on each building type and army unit type. We define those values in your implementation.

The game follows a real-time time pace, where building, producing, or upgrading take some game time to complete. You can define a guard time, in terms of ours, where your village is not exposed to attacks so that it can recover from a previous attack or freely upgrade without concerns.

When attacking, your army can pillage some loot from the village your attacked. If your village is attacked, it can be pillaged and loose gold, iron, and wood. Each member of army has hitpoints and inflict damage when attacking. All your buildings have hitpoints, and defence buildings inflict damage to attacking armies. The success in an attack depends on the size of the army, including the level of its units, against the defence capacity of the attacked village. The defence capacity comes from the number and level of defence units, the size of the village, level of its buildings, and number of workers, miners, and collectors.

- Suggestion (extra): an attack can be simulated in a fine grain level. That adds more realism to the attacks, turning the game interesting.

Suggestion (hint): when designing your class diagram, think of the extensibility of your application/game. Think about your game having new levels (home village extensions or new levels), new units, new buildings, and new upgrades.

Limits:

- **Map**: your village can grow up to an certain area or to certain number of buildings;
- **Upgrade**: All village elements can be upgraded to certain max level (damage and hitpoints);
- **Mining**: There is a maximum (upgradeable) capacity for gold, wood, and iron;
- **Loot**: There is a maximum loot (proportional to the size pillaged village). Loot also depends on the "success level" of an attack.

A **Game Engine** controls all game actions, deciding if upgrades, attacks, and production are allowed. It also determines the termination (time) of upgrades and production. When a user decides to attack another village, the Game Engine must "randomly" generate possible villages, define the success of attacks, and the loot of attacks. The Game Engine also "randomly" generates an army to attack the user's village whenever the village is in the non-guard period.

As the **main objective** of the game, **users** need to max-upgrade their village, keep an army to attack other villages, and rank up.

- It is up to you to define the ranking of a village. It might be related to the amount of attack wins, defence wins, and/or accumulated loot.

Assignment Details

Redesign of the Strategy Game so that it contemplates a series of Design Patterns. All the game functionalities, requested from Assignment 2, should be still provided in the redesigned code of Assignment 3.
The Game should allow the following game functionalities:

- **Building** – the player should be able to construct any new building, following the allowed limits of the game;
- **Training** – the player should be able to train/produce habitants provided that they have the supporting structures and up to a maximum limit.
- **Upgrading** – the player should be able to upgrade any building and habitant up to a maximum level limit;
- **Generating Village** – the should be able to generate a complete village for the player to attack. This village should in similar overall compatible level as the attackers' village and army;
- **Attack Exploring** – the player should be able to choose to attack a village that the game provides. The player can keep choosing until a suitable village is found.
- **Attacking** – the game should be able to generate overall scores from the attacker's army and the attacked village. The score can be used to determine if the attack succeeds, which can be in terms of percentage; it can also give a hint for a proportional pillage loot.

Since the Strategy Game is by definition an interactive system that involve the user, at least one player, it can greatly benefit from an Architectural Design Pattern: **Model View Controller** (MVC). Thus, ***(i)*** students will have modify and ensure that their program design follows the MVC Pattern.
Also, students will have to ***(ii)*** apply either **Factory** or **Abstract Factory Creational Design Pattern** in some parts of their code so that there is a more transparent handling of object instantiation. They will have to ***(iii)*** apply the **Adapter Structural Design Pattern** in their code (either Object or Class Adapter). The Adapter will enable the code of the student to make use of a set of classes that are provided for deciding the outcome of an attack. The student cannot modify the provided classes at all.
As a plus in this assignment (additional marks – +15), the student can incorporate functionalities in their program through the use of XML or JSON. The state of a Village can be saved and loaded up whenever the user executes game again. The Village state should be loaded from file and created/instantiated before the game starts. The elements of the Village must be properly stored in an ***(iv)*** **XML or JSON file**, created by the student. The file can follow a **XSD of JSON Schema**, which is defined by the student.
Important Notes

1. Applying the ***(i)*** MVC Pattern on the code may require modifications in the code. The amount of modifications depends on how the design and code have been performed in Assignment 2.
   Note: the student should explicitly and objectively explain in the **Description File** how their updated

design now follows the MVC Pattern, pointing out how interactions and which classes below to the parts of MVC.

2. The student may choose to use either ***(ii)*** Factory or Abstract Factory to implement certain transparency when instantiating classes for the simulator. Please note that it is not necessary to apply such Creational Design Patterns all over the code. The student can choose which parts of the code are more suitable to use these Patterns; the patterns should be employed at least once in the code.
   Note: again, the student should indicate which parts of the code (tell which classes – the client requesting instantiation and classes being instantiated) are using Factory of Abstract Factory Patterns. The student should also explicitly and objectively explain in the **Description File** how part(s) of their updated design now follow(s) such Pattern, pointing out how interactions and which classes follow now the Pattern.
3. The student will need to modify some parts of the code in order to apply ***(iii)*** Adapter Structural Design Pattern. The student will have to apply the pattern only for making a seamless translation of the what the current Assignment-2 code is doing in terms of deciding the outcome of Army attacks. A Package with a series of classes is provided; these classes accept an inpput through an specific API and calculate the outcome of the attack. The student will have to employ these classes without modifying them at all. The student can choose to utilize an Object Adapter of a Class Adapter.
   Note: again, the student should indicate which parts of the code (tell which classes – the client requesting instantiation and classes being instantiated) are using Adapter Structural Design Pattern. The student should also explicitly and objectively explain in the **Description File** how part(s) of their updated design now follow(s) such Pattern, pointing out how interactions and which classes follow now the Pattern.
4. **ADDITIONAL.** The student will need to modify some initialization parts of the code in order to have the status of a Village, if not a new one, to be ***(iv)*** loaded up and parsed from an XML or JSON file. The file must follow an Schema (XSD or JSON) that must also be defined.
   Note: again, the student should indicate which parts of the code (tell which classes – the client requesting instantiation and classes being instantiated) are loading the XML/JSON file and conducting the parsing. The student should also explicitly and objectively explain in the **Description File** how part(s) of their updated design now follow(s) such modifications, Pattern, pointing out how interactions and which classes interact to load the map.
5. **Reiterating** – Recall that this course is about advanced object-oriented programming. Thus, you will want to ensure that your implementation makes use of object-oriented constructs, such as interfaces, inheritance, enumerations, and generics, where appropriate. Such constructs should have been identified during design.
6. **Reiterating** – Be sure to include sufficient comments which should consist of, at minimum, an appropriate comment for each file and method. Consider using Javadoc style comments. If you use NetBenas, Javadoc comments can be auto-created in NetBeans by placing the cursor above a method or a class that currently has no Javadoc comment, typing "/**", then pressing Enter. This will create a template Javadoc comment with some information automatically completed for you (e.g., parameters for a method).
   - Your comments, if done correctly, should facilitate writing the document that explains your code. The comments should match with what you have in your description document.
7. Besides the Object Orientation concepts, you must be able to show complete understanding of the following concepts and employ them in your code. **Adequately use them whenever possible and explain them, pointing out in your description file** (in other words, marking will consist checking the existence and proper use of each of these elements):
   (a) Redesign of the code following the MVC Architectural Pattern;

(b) Factory or Abstract Factory Creational Design Pattern;
(c) Adapter Structural Design Pattern;
(d) **ADDITIONAL** - Loading and Parsing XML/JSON files;
(e) **ADDITIONAL** - Defining the XML/JSON Schema in a file.

Submission Material

The submission for this assignment will consist of ***TWO PARTS***:

1. A ***Description document (PDF)***. A document succinctly describing your design and implementation decisions is necessary. Also, you will find it beneficial to justify your design choices such that the marker does not have to reason about why you have designed your system as you have. Make sure that the description and reasoning of your design decisions are consistent.
    - ***Latex template - a must for writing your assignment***. For writing your description file, use the Latex template enclosed in this assignment (update it accordingly!). You do not need to install Latex software in your computer. You can write it through Overleaf on your browser (it is a free tool). Just upload the latex template to your Overleaf project; it should compile/render the tex file gracefully.
2. The **respective Java code of your designed classes (in a zip file)**. It consists of the implementation of the design created in the previous assignment or the one provided. The code should follow the UML class diagram. The code also must address the OOP concepts listed above. The classes and methods must be documented (commented). The code should compile and run properly. The compilation and execution of your code should not rely on any IDE.
    - ***Compilation***. Provide the command for compiling your source code from the command line.

You must guarantee that your code is legible, clear, and succinct. Keep in mind that any questionable implementation decision or copy from any source might have a negative effect on your mark. If you still have any questions regarding which file types are acceptable, please inquire prior to submission. Note that it is not the fault of the marker if they are unable to mark your assignment due to submitting an unreasonable or uncommon file format.

Marking Scheme

Marks will be awarded for completeness and demonstration of understanding of the material. It is important that you fully show your knowledge when providing solutions in a concise manner. Quality and conciseness of solutions are considered when awarding marks. Every code added to the originals should be well commented and explicitly indicated in the Java files; lack of clarity may lead you to loose marks, so keep it simple and clear.

Submission

Submission is to be a PDF (description document), and text Java files. All the submission should be performed electronically through D2L.

You must guarantee that your code is legible, clear, and succinct. Keep in mind that any questionable implementation decision or copy from any source might have a negative effect on your mark. If you still have any questions regarding which file types are acceptable, please inquire prior to submission. Note that it is not the fault of the marker if they are unable to mark your assignment due to submitting an unreasonable or uncommon file format.

All code files should be organized and put together in a ZIP for the submission through D2L. The description document should be submitted separately as a PDF file in D2L.

* **Do not forget to include the names and student IDs of group members.**
** **Only one student needs to submit the assignment on behalf of a group.**

Plagiarism

Students are expected respect academic integrity and deliver evaluation materials that are only produced by themselves. Any copy of content, text or code, from other students, books, web, or any other source is not tolerated. If there is any indication that an activity contains any part copied from any source, a case will be open and brought to a plagiarism committee's attention. In case plagiarism is determined, the activity will be cancelled, and the author(s) will be subject to the university regulations.
For further information on this sensitive subject, please refer to the document below:

`https://brocku.ca/academic-integrity/`