Assignment 1

COMP 250 Fall 2019

posted:	Tuesday, Sept. 24, 2019
due:	Tuesday, Oct. 8, 2019 at 23:59

The teaching staff handling this assignment consists of Giulia Alberini (email), Sayantan Datta (email), Anmoljeet Gill (email), Alexander Orzechowski (email), Kelly Rombough (email), Rajveer Gandhi (email).

Their office hours is posted on mycourses. Additional office hours will be offered while assignment 1 is out. For all information, please refer to myCourses.

Learning Objectives

This assignment is meant for you to practice what we have learned in class in the past week. A lot of the design decision have been taken for you, but it is important for you to ask yourself why each choice has been made and whether there could be a better way of doing it. You'll soon realize that the classes you have to write are all closely related to one another. We hope that the assignment will help you appreciate the importance of class design. This is of course just a taste, you will learn much more about it in COMP 303. As mentioned in class, we suggest you take the time to draw out a class diagram. This should help you develop a clear picture of the relationship between all these classes.

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times but only the latest submission will be evaluated. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).

- Please store all your files in a folder called "Assignment1", zip the folder and submit it to myCourses. Inside your zipped folder, there must be the following files.
  - `Unit.java`
  - `Settler.java`
  - `Worker.java`
  - `MilitaryUnit.java`
  - `Warrior.java`
  - `Archer.java`
  - `ListOfUnits.java`
  - `Tile.java`

  Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks

- You will have to create all the above classes from scratch. The assignment shall be graded automatically. Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class (including for example `ArrayList` or `LinkedList`). In this assignment you are also not allowed to add any additional fields (neither `private` nor `public`). **Any failure to comply with these rules will give you an automatic 0.**
- We have included with these instruction a `SyntaxTester.java` to help you make sure that you followed the naming instructions correctly. The tester will check the class names, as well as the methods' headers. If your code fails the `SyntaxTester` it means that you have not followed the instructions correctly. **To run the tester place all the classes as well as `SyntaxTester.java` in the default package. Code that fails the `SyntaxTester` will receive a 0**.
- On Thursday, we will also release a second tester, which is a mini version of the final tester used to evaluate your assignment. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We therefore highly encourage you to modify the tester class and expand it.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on myCourses.

Toward a Strategy Game

For this assignment you will write several classes as a first step toward building a very simplified version of a Civilization-inspired strategy game. Make sure to follow the instructions below very closely. Note that in addition to the required methods, you are free to add as many other **private** methods as you want. You can also add `public toString()` methods in each class to help with the debugging process. No other additional non-`private` method is allowed. **No additional (`public` or `private`) fields are allowed.**

Let's start by creating a skeleton for some of the classes you will need. Note that later we will define a new data type called `Tile`. For the moment just go ahead and use it. It is normal that your code will not compile until you create a `public` class called `Tile`.

- Write an `abstract` class `Unit` which has the following `private` fields:
  - A `Tile` representing the position of the unit in the game.
  - A `double` representing the health points (hp) of the unit.
  - An `int` representing the available moving range of the unit.
  - A `String` representing the faction to which the unit belongs. The faction determines to which group of people this unit belongs. Examples of factions in civilization are: Barbarians, Aztecs, Sumerians, Romans, Greeks, etc.

  The class must also have the following `public` methods:
  - A constructor that takes as input a `Tile` indicating the position of the unit, a `double` indicating its hp, an `int` indicating its moving range, and a `String` indicating its faction. The constructor uses its inputs to initialize the corresponding attributes.
  - A `final getPosition()` method to retrieve the position of *this* `Unit`.
  - A `final getHP()` method to retrieve the health points of *this* `Unit`.
  - A `final getFaction()` method to retrieve the faction of *this* `Unit`.
- All of the following must be subclasses of the `Unit` class:
  - Write a class `Settler` derived from the `Unit` class. The `Settler` class has the following `public` method:
    - A constructor that takes as input a `Tile` which contains the position of the settler, a `double` indicating its hp, and a `String` indicating its faction. The constructor uses the inputs to create a `Settler` with the above characteristic and moving range equal to 2.
  - Write a class `Worker` derived from the `Unit` class. The `Worker` class has the following `private` field:
    - An `int` indicating the number of jobs already performed by the worker.

    The `Worker` class has also the following `public` method:

    - A constructor that takes as input a `Tile` which contains the position of the worker, a `double` indicating its hp, and a `String` indicating its faction. The constructor uses the inputs to create a `Worker` with the above characteristic, moving range equal to 2, and number of jobs performed equal to 0.

  - Write an `abstract` class `MilitaryUnit` derived from the `Unit` class. The `MilitaryUnit` class has the following `private` fields:

    - A `double` indicating the attack damage of the unit.
    - An `int` indicating the attack range of the unit.
    - An `int` indicating the armor of the unit.

    The `MilitaryUnit` class has also the following `public` method:

    - A constructor that takes as input a `Tile` which contains the position of the unit, a `double` indicating its hp, an `int` indicating its moving range, a `String` indicating its faction, and three additional fields: one `double` indicating its attack damage and two `int`s indicating attack range and armor respectively. The constructor uses the inputs to create a `MilitaryUnit` with the above characteristic.

  - Write a class `Warrior` derived from the `MilitaryUnit` class. The `Warrior` class has no fields, but it has the following `public` method:

    - A constructor that takes as input a `Tile` which contains the position of the warrior, a `double` indicating its hp, and a `String` indicating its faction. The constructor uses the inputs to create a `MilitaryUnit` with the above characteristic, moving range equal to 1, attack damage equal to 20.0, attack range equal to 1, and armor equal to 25.

  - Write a class `Archer` derived from the `MilitaryUnit` class. The `Archer` class has the following `private` field:

    - An `int` indicating the number of arrows available to the archer.

    The `Archer` class has also the following `public` method:

    - A constructor that takes as input a `Tile` which contains the position of the archer, a `double` indicating its hp, and a `String` indicating its faction. The constructor uses the inputs to create an `Archer` with the above characteristic, moving range equal to 2, attack damage equal to 15.0, attack range equal to 2, armor equal to 0, and 5 available arrows.

You can now leave these classes as they are, we will come back to them later. Let's instead focus on the two other classes:

- Write a class `ListOfUnits`. The purpose of this class is to implement your own arraylist of Unit. We want to stress once again that you are not allowed to import any class for this assignment! The class `ListOfUnits` must have the following `private` fields:

  - An array of `Unit`s which will be used to store the units that are part of the `ListOfUnits`.

- An `int` indicating the size of the `ListOfUnits`, i.e. how many units have been stored in this list.

The class must also have the following `public` methods:

- A constructor that takes no inputs and creates an empty `ListOfUnits`. To do so, the fields should be initialized with an array of size 10. No units should be stored in the array at this moment. The size of the `ListOfUnit` should reflect this.
- A `size()` method that takes no inputs and returns the number of units that are part of *this* `ListOfUnits`.
- A `getUnits()` method which takes no inputs and returns an array containing all the units that are part of *this* `ListOfUnits`. This array should not contain any `null` elements.
- A `get()` method which takes as input an `int` and returns the unit at the specified position in *this* list. If the integer received is out of range, i.e. negative or greater than or equal to the number of units in the list, then the method should throw an `IndexOutOfBoundsException`.
- An `add()` method which takes as input a `Unit` and does not return any value. The method adds the `Unit` at the end of *this* list. If not enough space is available, then the method should resize the array of units. The capacity of the array should increase by a factor of 1.5 using the following formula:

  ```
  new_capacity = old_capacity + old_capacity/2 + 1
  ```

  This is a great place to create your own private method to performs such operation. Warning: make sure to not "lose" any of the units from the list.
- An `indexOf()` method which takes as input a `Unit` and returns an `int` indicating the position of the **first** occurrence of the specified element in *this* list. If no such unit exists, then the method returns -1. Remember that when comparing reference types you do not want to use '`==`'.
- A `remove()` method which takes as input a `Unit` and returns a `boolean`. The method removes the **first** occurrence of the specified element from the array of units of *this* list. If no such unit exists, then the method returns `false`, otherwise, after removing it, the method returns `true`. Note that this method **removes a unit from the list if and only if such unit is *equal* to the input received**. For example, it is not possible to remove an Aztect settler in place of a Sumerian settler. After the unit has been removed from the array, the subsequent elements should be shifted down by one position, leaving no empty slot in the array. No resizing of the array is necessary!
- A `getArmy()` method that takes no inputs and returns an array of `MilitaryUnit`s. The array should contain all `MilitaryUnit`s that are part of *this* list. Note that not all units in the list are military units. A `ListOfUnits` may include also setters and workers.

- Write a class `Tile`. You can think of a Tile as a square on the board on which the game will be played. A Tile must have the following `private` fields:

- An `int` indicating the x-coordinate of the tile.
- An `int` indicating the y-coordinate of the tile.
- A `boolean` indicating whether or not a city has been built on the tile.
- A `boolean` indicating whether or not the tile received some "improvements".
- A `ListOfUnits` containing all the units positioned on the tile. This is where you will be using your own implementation of an arraylist of units.

The class must also have the following `public` methods:

- A constructor that takes as input two `int`s indicating the x and the y coordinate respectively. The constructor creates a Tile with the specified coordinates. A new tile is not a city, nor it has ever been improved. It also hosts no units. You must represent this by initializing its corresponding field with an empty `ListOfUnits`.
- A `getX()` and a `getY()` method which return the x and the y coordinate of the tile respectively.
- A `isCity()` method which returns whether or not the tile is a city.
- A `isImproved()` method which returns whether or not the tile has been improved.
- A `foundCity()` method which turns the tile into a city if it wasn't already.
- A `buildImprovement()` method which improves the tile if it wasn't already.
- An `addUnit()` method which takes as input a unit and adds it to the tile's `ListOfUnits`. Note that a military unit can be added to the tile if and only if no military unit of a different faction (the enemies' army!) is stationed here. Non-military units can always be added to the tile. The method returns `true` if the unit was successfully added to the list, `false` otherwise.
- A `removeUnit()` method which takes as input a unit and removes it from the tile's `ListOfUnits`. The method should also return a `boolean` indicating whether or not the operation was successful.
- A `selectWeakEnemy()` method which takes as input a `String` representing a faction. The method should return the enemy unit (i.e. any `Unit` with a faction different than the one specified) stationed on the tile with the lowest health. If no enemy unit is present, then the method returns `null`. If more than one enemy unit has the lowest health, then the one that appears first in the list should be returned.
- A `static` method called `getDistance()` which takes as input two tiles and returns a `double` indicating the distance between the two. Remember that given two points (x_1, y_1) and (x_2, y_2), the distance can be computed with the following formula:

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

We are now ready to go back to the classes we created at the beginning.

- In the class `Unit` go back to the constructor and make sure that when a unit with a specified position is created, such unit is also added to the `ListOfUnits` of the corresponding tile. Note that it is not always possible to do that (we cannot have enemy military units on the same tile!). If it is not possible to add the unit to the specified tile, then the constructor should throw an `IllegalArgumentException`.

  To the `Unit` class add the following `public` methods:

  - A `moveTo()` method which takes as input a `Tile` representing where the unit should be deployed. Note that a unit can be moved only if the target tile is within the moving range of the unit and no enemy military units is already stationed there. The method returns `true` if the unit is successfully moved, `false` otherwise. Note that a target tile is within the unit range if the distance to the tile is less than the unit range plus one. Note also that when a unit is moved to a different tile this should be reflected both on the unit's position as well as the old tile's `ListOfUnits`.
  - A `receiveDamage()` method which takes as input a `double` indicating the damage received by the unit. The method applies the damage to the unit by modifying the unit's health. To do so, subtract the damage from the unit's health points. Note that if the unit is stationed on a city, then the damage should be reduced by 10% before being applied. Moreover, if after the damage is applied the unit ends up having a non-positive health (0 or below), then the unit has been killed. In such a case, the unit should be removed from the game. To do that, remove the unit from the tile.
  - An `abstract` method `takeAction()` which takes as input a Tile and does not return any value. This method should be abstract (thus, not implemented) because the action to take depends on the type of the unit.
  - An `equals()` method which takes an `Object` as an input and returns `true` if the input matches `this` in type, position, health and faction. Otherwise the method returns `false`.

- In the `Settler` class do the following:

  - Implement the `takeAction()` method. The method takes a `Tile` as input. If this is the tile where the unit is stationed and it is not a city yet, then a city should be built on the specified tile. In such a case the unit is expended. To represent this, remove the unit from the tile. On the other hand, if a city is not built the method simply terminates.
  - Override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health and faction. Otherwise the method returns `false`. Note that you do not want to rewrite code that you have already written in the superclass. How can you access methods from the superclass that have been overridden?

- In the `Worker` class do the following:

  - Implement the `takeAction()` method. The method takes a `Tile` as input. If this is the tile where the unit is stationed and it has not been improved yet, then an improvement should be built on the specified tile. If an improvement is built, then the number of jobs

this worker has already performed should increase by one. If the worker reaches 10 jobs, then the unit is expended, i.e. the unit is removed from the tile.

  - As for the `Settler`, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, faction, and number of jobs performed.

- In the `MilitaryUnit` class do the following:
  - Implement the `takeAction()` method. The method takes a `Tile` as input. If the distance between the target tile and the position of the unit is greater than or equal to the unit's attack range plus one, then an attack is not possible and the method terminates. Otherwise, the weakest enemy (a unit of a different faction) on the target tile is selected and receives a damage equal to *this* unit's attack damage. Note that, if *this* unit is stationed on a tile which has been improved, then the damage inflicted should be increased by 5%. If no enemy unit is stationed on the specified tile, then the method does not do anything.
  - Override the `receiveDamage()` method. The method receives a `double` as input indicating the damage the unit should receive. Before applying the damage to the unit (in the same way as `receiveDamage()` from `Unit` does), the damage should be multiplied by a multiplier which depends on this unit's armor. The multiplier is a double equal to $100/(100 + armor)$. Once again, you do not want to rewrite code that you already have written in the superclass. How can you access methods from the superclass that have been overridden?

- In the `Warrior` class do the following
  - As for the `Settler` and `Worker`, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, and faction.

- In the `Archer` class do the following
  - Override the `takeAction()` method. If the archer does not have any more arrows, it cannot attack. The method will instead take this time to reset the number of arrows to 5. On the other hand, if the archer has still available arrow it attacks like a `MilitaryUnit` and uses an arrow to do so.
  - As for the `Settler`, `Worker` and `Warrior`, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, faction, and available arrows.