

---

# Assignment 1

ECSE 250      Fall 2023

posted:      Sept. 22nd, 2023  
due:          Oct. 13th, 2023

## Learning Objectives

This assignment is meant for you to practice what we have learned in class in the first four weeks of the semester. A lot of the design decisions 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. Some of our choices were influenced by the intention of having you practice most of what you have learned in class. Others were dictated by our need to be able to fully test your assignments. Finally, we will not be grading your work on efficiency for the first assignment, but your implementation decisions can affect how efficient the code for this assignment is. Try to keep this in mind while coding. Make sure to ask yourselves why certain modifiers (**private**, **public**, **static**,...) were used and if you would have made the same decisions. We hope that the assignment will help you appreciate the importance of class design. We suggest you take the time to draw out a **class diagram** or a UML diagram. This should help you develop a clear picture of the relationship between all these classes.

## General Instructions

- **Submission instructions**

- Late Submission: Assignments are due at 5 pm Montreal time unless stated otherwise. Late work will be automatically subject to a 20% penalty and can be submitted up to 3 days after the deadline. Note that submitting one minute late is the same as submitting 3 days late. Late submission will be counted 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 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 Ed Lessons may be overloaded during rush hours).
- These are the files you must submit on Ed:
  - \* `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 and the minitester.** Any deviation from these requirements may lead to lost marks.

- Please make sure that all the files you submit are part of a package called `assignment1`.
- 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 instructions 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`). **Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to Ed, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. 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. Please note that these tests are only a subset of what we will be running on your submissions, we will test your code on a more challenging set of examples. **Passing the exposed tests assures you that your submission will not receive a grade lower than 40/100 (assuming that you follow the submission instructions correctly)** We highly encourage you to test your code thoroughly before submitting your final version.
- With this assignment, a class called `Minitester` will also be posted. The tests included in this class are equivalent to those exposed on Ed. We encourage you to modify and expand this class. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- Your submission will automatically get a 0 if the 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 Ed.
- **IMPORTANT:** Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program otherwise. If you need help debugging, feel free to reach out to the teaching staff. When doing so, make sure to mention what is the bug you are trying to fix, what have you tried to do to fix it, where have you isolated the error to be.

---

*The Civilization 250 is a strategy game that immerses players in the roles of historical leaders. Each player takes turns, making decisions like move their people, build cities, or improve their territories, all with the ultimate goal of guiding their civilization to glorious victory.*

*In Civilization 250, the term “unit” encompasses any entity capable of movement on the game map. There are four distinct types of units in the game: settlers, workers, warriors, and archers. The military units, such as warrior and archer units, possess the capability to engage in combat with other units. On the contrary, non-military units, specifically settlers and workers, have different capabilities. The settlers can establish new cities, while the workers are tasked with enhancing the terrain.*

*A significant portion of the game revolves around unit movement within the game world. The world is depicted as a two-dimensional grid, and each individual cell within the grid is referred to as a “tile”. You will march your military units to engage in conflicts with neighboring civilizations or guide your non-military units to discover new lands and cultivate new territory.*

## 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 methods is allowed. **No additional (public or private) fields are allowed unless otherwise specified.**

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, i.e. how far can this unit move in one single turn.
  - A **String** representing the faction to which the unit belongs. The faction determines to which group of people this unit is part of. 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 `ints` 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 list of Units. Note that you will not be tested on the efficiency of your code, but we highly encourage you to think about how you can make your code more efficient when implementing a data structure that does not have a fixed size of elements. 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 **Units** 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 list of units. To do so, the fields should be initialized to reflect the fact that at the moment there are **no units** in the list.
  - A **getSize()** method that takes no inputs and returns the number of units that are part of this list.
  - A **getList()** method which takes no inputs and returns an array containing all the units that are part of this list. The units should appear in the order in which they have joined the list. This array must contain as many elements as the number of units in the list, and it should not contain any **null** elements.
  - A **getUnit()** method which takes as input an **int** and returns the reference of 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 **addUnit()** method which takes as input a **Unit** and does not return any value. The method adds the **Unit** at the end of this list. Make sure to handle the case in which there might not be enough space for this unit to join the list. In such case, you need to make sure to create additional space. No unit should be rejected from joining the list.

---

If need be, this is a great place to create your own private method to help you with the implementation. Warning: make sure that no unit is removed from the list as a result of adding the new one.

- 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 `removeUnit()` 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 Aztec settler in place of a Sumerian settler.
- A `getArmy()` method that takes no inputs and returns an array of `MilitaryUnits`. The array should contain all `MilitaryUnits` that are part of this list. Note that not all units in the list are military units. A `ListOfUnits` may include also settlers and workers. The array returned by the method should not contain any `null` elements.
- 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.

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

- A constructor that takes as input two `ints` 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 `buildCity()` 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 reference of 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 if the unit is allowed to be stationed on such tile (i.e. no military unit of different faction should ever be stationed on the same tile). Note that it is possible for non military units to be stationed on the same tile with other units from different factions. 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 **or equal to** the unit's moving range. 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. If positive, 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. If the input is a negative number, then the method should not do anything and you can simply return.

- 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**. Be careful when comparing doubles! You should not be comparing them directly, instead consider the health of the two units to be equal if and only if the square of their difference is less than 0.001 (i.e.  $(HP_1 - HP_2)^2 < 0.001$ )
- 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. If the conditions on the input value are not met, then the method simply terminates.
  - Similar to 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, 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  $100/(100 + \text{armor})$ . 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
  - Similar to 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. **Note that the archer should use an arrow as soon as they try to attack (i.e. even in the case in which they are attacking a tile out of range or without any enemy unit)**
  - Similar to 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.