Assignment 1

ECSE 250 Winter 2024

Posted:	Monday, Jan. 29, 2024
Due:	**Tuesday, Feb. 13, 2024 at 16:59 (4:49 pm)**
GitHub Invitation:	https://classroom.github.com/a/a5bsI3JF

Learning Objectives

This assignment is meant for you to practice what we have learned in class in the past few weeks. 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. 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 3 days late with a 20% penalty. Please note that submitting one minute late is equivalent to submitting 23 hours late. Resubmissions after the due date, regardless of the reason (e.g., wrong file submitted, incorrect file format, or any other reason), will be considered late submissions. This policy applies regardless of whether or not the student can provide proof that the assignment was completed on time.
  - All your work must be submitted on GitHub. Similar to the previous weekly assignments, you can begin by accepting the assignment at https://classroom.github.com/a/a5bsI3JF. Subsequently, a repository naming `assignment-1-***` will be created for you (`***` is your GitHub username). You can then clone the repository, and work on it. If you want to recap the instructions to setup GitHub, clone your repository and submit your code, you can refer to the instructions of weekly assignment 1 (https://classroom.github.com/a/8qHfdL6n).
  - Don't worry if you realize that you made a mistake after you submitted: you can git push multiple times but only the latest submission will be graded. Regularly pushing your work with well-documented commits is an encouraging practice to stay on track with your changes. You will be using the following commands.
    - git add *
    - git commit –m "update yyyy method in xxxx.java."
    - git push

- Only the code submitted to GitHub will be graded. Please ensure that your repository contains the work that you want to be assessed. If you can see the latest version of your code in your repository on GitHub, it means that you successfully submitted your code.
- These are the files you should be submitting on GitHub:
  * `Insect.java`
  * `Hornet.java`
  * `HoneyBee.java`
  * `BusyBee.java`
  * `AngryBee.java`
  * `FireBee.java`
  * `SniperBee.java`
  * `Tile.java`
  * `SwarmOfHornets.java`

  Do not submit any other files, especially .class files.
- Please make sure that all the files you submit are part of a package called `assignment1`.
- **Your `assignment1` package should be put into the `src` folder when you push to GitHub. In other words, you will have your `.java` files in folder `assignment1` and your `assignment1` folder should be in folder `src`. Failing to do so will result in 10% penalties.**

- 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). **Any failure to comply with these rules will give you an automatic 0.**

- **You will automatically get 0 if your code does not compile.**

- With this assignment, a class called MiniTester will also be posted. 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. We will test your code on a much more challenging set of examples. Passing the exposed tests assures you that your submission will not receive a grade lower than 40/100. We highly encourage you to test your code thoroughly before submitting your final version.

- We encourage you modify and expand the MiniTester. *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.

- 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.

- If you have any questions related to your grade, you can request a regrade within 7 days after the release of the grade by posting a private post on Ed and including your McGill ID.

- **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.** Start testing and debugging while you are implementing your classes. 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.

Toward a Tower Defense Game

For this assignment you will write several classes as a first step toward building a very simple Tower Defense game. Tower Defense games are strategy games where the goal is to defend the player's territories by obstructing the advance of the enemies troops. Note that you do not need to be familiar with Tower Defense games in order to successfully complete the assignment. 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. Unless otherwise specified, never make copies (neither shallow, nor deep) of any of the objects the methods return or receive as input.

STEP 1: The skeleton

Let's start by creating a skeleton for some of the classes you will need.

- Write a class called `Tile`. You can think of a tile as a square on the board on which the game will be played. We will come back to this class later. For the moment you can leave it empty while you work on creating classes that represents characters in the game.
- Write an `abstract` class `Insect` which has the following `private` fields:
  - A `Tile` representing the position of the insect in the game.
  - An `int` representing the health points (hp) of the insect.

  The class must also have the following `public` methods:
  - A constructor that takes as input a `Tile` indicating the position of the insect, an `int` indicating its hp. The constructor uses its inputs to initialize the corresponding fields.
  - A `final getPosition()` method to retrieve the position of *this* insect.
  - A `final getHealth()` method to retrieve the health points of *this* insect.
  - A `setPosition()` method that takes a `Tile` as input and updates the value stored in the appropriate field.
- All of the following must be subclasses of the `Insect` class:
  - Write a class `Hornet` derived from the `Insect` class. The `Hornet` class has the following `private` field:
    * An `int` indicating the attack damage of the hornet.

    The `Hornet` class has also the following `public` method:
    * A constructor that takes as input a `Tile` which represents the position of the hornet, an `int` indicating its hp, and an `int` indicating its attack damage. The constructor uses the inputs to create a `Hornet` with the above characteristic.

- Write an `abstract` class `HoneyBee` derived from the `Insect` class. The `HoneyBee` class has the following `private` field:
    - An `int` indicating how much this bee costs in food. Whenever a player wants to place a bee in the game, they will need to have a certain amount of food available. Larvae need to eat to become strong bees!

  The `HoneyBee` class has also the following `public` methods:
    - A constructor that takes as input a `Tile` representing the position of the bee, an `int` indicating its hp, an `int` indicating its cost in food respectively. The constructor uses the inputs to create a `HoneyBee` with the above characteristic.
    - A `getCost()` method which returns how much this bee costs in food.
- Write a class `BusyBee` derived from the `HoneyBee` class. The `BusyBee` class is used to represent bees that collect pollen. This class has two `public static` fields:
    - An `int` called `BASE_HEALTH` indicating the base health of busy bees.
    - An `int` called `BASE_COST` indicating the their base food cost.

  This class has also the following `public` method:
    - A constructor that takes as input a `Tile` which represents the position of the bee in the game. The constructor uses the input to create a bee with the given position, base health and base food cost.
- Write a class `AngryBee` derived from the `HoneyBee` class. The `AngryBee` class is used to represent our private soldiers. These bees fly around stinging single enemies at melee range. The class has the following fields:
    - A `private int` indicating the attack damage of the bee.
    - A `public static int` called `BASE_HEALTH` indicating the base health of angry bees.
    - A `public static int` called `BASE_COST` indicating the their base food cost.

  The `AngryBee` class has also the following `public` method:
    - A constructor that takes as input a `Tile` representing the position of the bee, and an `int` indicating its attack damage. The constructor uses the inputs to create a bee given the position, the attack damage, base health, and base food cost.

STEP 2: Set up of the main data structures

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

- Write a class `SwarmOfHornets`. This class represents a group of hornets together. Despite what the name 'swarm' might let you think, these hornets group together forming a very neat line. The purpose of this class is to implement your own queue (or list) of hornets. 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 and for which the most common operations are removing the first element of the queue and adding an element to the end of the queue. We want to stress once again that you are not allowed to import any class for this assignment! The class `SwarmOfHornets` must have the following `private` fields:

– An array of `Hornets` which will be used to store the hornets that are part of the swarm.

– An `int` indicating the size of the swarm, i.e. how many hornets are part of the swarm.

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

– A constructor that takes no inputs and creates an empty swarm. To do so, the fields should be initialized to reflect the fact that at the moment there are **no hornets** in the swarm.

– A `sizeOfSwarm()` method that takes no inputs and returns the number of hornets that are part of *this* swarm.

– A `getHornets()` method which takes no inputs and returns an array containing all the hornets that are part of *this* swarm. The hornets should appear in the order in which they have joined the swarm. This array must contain as many elements as the number of hornets in the swarm, and it should not contain any `null` elements.

– A `getFirstHornet()` method which takes no inputs and returns the hornet who joined the swarm first, between those that are currently part of it. If there's no hornet in this swarm, then the method returns `null`. Note that this method should not remove the hornet from the swarm.

– An `addHornet()` method which takes as input a `Hornet` and does not return any value. The method adds the hornet at the end of the queue of hornets in *this* swarm. Make sure to handle the case in which there might not be enough space for this hornet to join the swarm. In such case, you need to make sure to create additional space. No hornet should be rejected from the swarm. 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 hornet is removed from the swarm as a result of adding the new one.

– A `removeHornet()` method which takes as input a `Hornet` and returns a `boolean`. The method removes the **first occurrence** (remember the hornets should appear in the swarm based on the order in which they joined it) of the specified element from *this* swarm. If no such hornet exists, then the method returns `false`, otherwise, after removing it, the method returns `true`. Note that this method should **compare hornets using directly their reference**[1].

ATTENTION: Before proceeding further with the assignment, ensure that your implementation of this class functions as intended. Much of the subsequent code you'll be writing depends on methods implemented in this class. Testing its functionality at this point is crucial!

[1]The reason for this is to remove any dependency between this method and your implementation of `equals()` (see later).

- Go back to the class `Tile` which you have created before. To this class add the following `private` fields:
  - An `int` indicating the food present on the tile. Bees collect food to feed their larvae. This is how strong bees are raised. If a player wants to add a new bee to the game, they will need to have enough food to do so.
  - A `boolean` indicating whether or not a bee hive has been built on the tile.
  - A `boolean` indicating whether or not a hornet nest has been built on the tile.
  - A `boolean` indicating whether or not this tile is part of the path that leads from the hornet nest to the bee hive. You can assume that in the game there is only one such path.
  - A `Tile` containing a reference to the next tile on the path from the hornet nest toward the bee hive. It contains `null` if the tile is not on the path or at the end of it.
  - A `Tile` containing a reference to the next tile on the path from the bee hive toward the hornet nest. It contains `null` if the tile is not on the path or at the end of it.
  - A `HoneyBee` indicating the bee positioned on the tile.
  - A `SwarmOfHornets` containing all the hornets positioned on the tile.

  Note that both the tile with the hive and the one with the nest will be part of the path leading from one to the other. *They become part of the path only when a connection to the other tiles is establishes, not when the hive/nest is built.*
  The `Tile` class must also have the following `public` methods:
  - A constructor that takes no inputs and creates a new tile. A new tile does not have a bee hive, a hornet nest, nor is on the path that leads from one to the other. On a new tile there's no food, no bee, and no hornets. Initialize the fields to represent this.
  - A second constructor that takes all the inputs needed to initialize the fields from this class. The method takes the inputs in the same order as the fields are listed above.
  - An `isHive()` method which returns whether or not a bee hive has been built on this tile.
  - An `isNest()` method which returns whether or not a hornet nest has been build on this tile.
  - A `buildHive()` method which builds a bee hive on this tile. This is a `void` method that simply updates the field indicating whether or not there's a bee hive on the tile.
  - A `buildNest()` method which builds a hornet nest on this tile. This is a `void` method that simply updates the field indicating whether or not there's a hornet nest on the tile.
  - An `isOnThePath()` method which returns whether or not this tile is part of the path leading from the hornet nest to the bee hive.

- A `towardTheHive()` method which returns the next tile on the path from the hornet nest to the bee hive. If this tile is not on the path or it is the tile with the hive, then the method returns `null`.
- A `towardTheNest()` method which returns the next tile on the path from the bee hive to the hornet nest. If this tile is not on the path or it is the tile with the nest, then the method returns `null`.
- A `createPath()` method which takes two `Tile`s as input representing the next tile on the path toward the hive, and the next tile on the path toward the nest respectively. The method updates the respective fields, making this tile become a tile that is part of the path that leads from the hornet nest to the bee hive. Note that it is possible for this method to receive `null` as one of the two inputs. This should only happen when the tile has a hive or a nest respectively and it is then placed at the extremities of the path that leads from one to the other. If this is not the case, the method should raise an `IllegalArgumentException`. To do so, write the following statement:

  ```
  throw new IllegalArgumentException(errMsg);
  ```

  where `errMsg` is a `String` containing an error message of your choice.
- A `collectFood()` method with takes no inputs and returns an `int` representing the food stored on the tile. The tile is then left with no food.
- A `storeFood()` method which takes an `int` as input representing the amount of food received and it adds it to the food stored on the tile. This method does not return anything.
- A `getNumOfHornets()` method which returns the number of hornets positioned on this tile.
- A `getBee()` method which returns the bee positioned on this tile. You should not make a copy of this object.
- A `getHornet()` method which returns the hornet who joined the swarm on this tile first, between those that are currently part of it. You should not make a copy of this object.
- A `getHornets()` method that returns an array containing all the hornets that are positioned on this tile. The hornets should appear in the order in which they have joined the swarm.
- An `addInsect()` method which takes as input an `Insect` and adds it to the tile. Note that a bee can be added to the tile only if there's no other bee positioned on this tile. More over, no bee can be positioned on the hornet nest! On the other hand, there's no limit to the number of hornets that can be positioned on a tile. But hornets can only be positioned on a tile that is on the path from the nest to the hive (including both the nest and the hive). The method returns `true` if the insect was successfully added to the tile, `false` otherwise. Note that adding an insect to a tile, not only changes the properties

of the tile, but also the properties of the insect. That is, the insect should now result as positioned on this tile. Once again, please do not make a copy of the input object.

- A `removeInsect()` method which takes as input an `Insect` and removes it from the tile. The method should also return a `boolean` indicating whether or not the operation was successful. Note that removing an insect from a tile, not only changes the properties of the tile, but also the properties of the insect. That is, the insect should now result as not positioned on a tile. You can indicate this by updating the position to be `null`. This method should **compare insects using directly their references**.

STEP 3: Basic game play mechanics

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

- In the class `Insect` go back to the constructor and make sure that when an insect with a specified position is created, such insect is also added to the corresponding tile. Note that it is not always possible to do that (for example there cannot be more than one bee on the same tile). If it is not possible to add the insect to the specified tile, then the constructor should *throw* an `IllegalArgumentException`.

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

  - A `takeDamage()` method which takes as input an `int` indicating the damage received by the insect. The method applies the damage to the insect by modifying its health. To do so, subtract the damage from the insect's health points. If, after the damage is applied, the insect ends up having a non-positive health (0 or below), then the insect has been killed. In such a case, it should be removed from the game. To do that, remove it from the tile. This method does not return any value.
  - An `abstract` method `takeAction()` which takes no inputs and returns a `boolean`. This method should be abstract (thus, not implemented) because the action to take depends on the type of the insect.
  - Override the `equals()` method. It takes as input an `Object` and returns `true` if it matches `this` in type, position and health. Otherwise, the method returns `false`.

- In the `Hornet` class do the following:

  - Implement the `takeAction()` method. If there's a bee positioned on the same tile as this hornet, the hornet will sting it inflicting on the bee an amount of damage equal to this hornet's attack damage. If on the other hand there's no bee on the same tile as this hornet, then the hornet will take a step toward the bee hive by moving to the next tile on the path. Note that this means that the tiles and the position of this hornet must all be updated accordingly. Whether the hornet stings a bee or moves to the next tile, the method returns `true`. If hornet on the other hand is already on the bee hive, and there's no bee left on the tile, then there's nothing for this hornet to do (the hornets have won the game!) and the method should return `false`. Note that you can assume that the hornet will never be positioned on a tile that is not on the path from the nest to the hive. Note also that the hornet either stings or moves. That is, even if as a consequence of the hornet's attack, the bee dies, the hornet cannot move forward until its next turn.

  - Override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health and attack damage. 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 `HoneyBee` class do the following:
  - Add a `public static double` called `HIVE_DMG_REDUCTION` indicating the percentage of damage reduction bees should received if positioned on the tile with the bee hive.
  - Override the `takeDamage()` method. If a bee is positioned on the tile with the bee hive, then the damage received should be reduced by `HIVE_DMG_REDUCTION` (the damage should be rounded toward 0).
- In the `BusyBee` class do the following
  - Add a `public static int` called `BASE_AMOUNT_COLLECTED` indicating the amount of food busy bees collect.
  - Override the `takeAction()` method. This type of bees collect pollen, so excuting an action results into `BASE_AMOUNT_COLLECTED` of food being added to the tile where the bee is positioned. The method returns `true`.
- In the `AngryBee` class do the following:
  - Implement the `takeAction()` method. An angry bee positioned on the path that leads from the nest to the hive will attempt to sting a hornet. If on the other hand, it is not positioned on the path, then it won't do anything and the method will return `false`. Note that if the bee tries to sting an hornet, it is not just simply stinging a random one! It will first look for a non-empty swarm, either on the on the same tile as the bee itself or on the next tile towards the nest. If it finds it, it will then sting the first hornet in the swarm by inflicting it an amount of damage equal to this bee's attack damage. Note that bees cannot sting a hornet that is positioned on its nest! If there's no hornet to be stung, then the bee won't do anything and the method returns `false`. If the bee stings a hornet, then the method returns `true`. Note that no matter what the bee does, the bee ends its action on the same tile that it started on. That is, this method does not modify the tile on which the bee is positioned.

STEP 4: Adding special units

Now that we have a working game, we can let our fantasy run and start adding some special units to make the game more interesting. For this assignment you should add the following:

- ***Fire Bees***. These bees have the ability to launch fire projectiles at tiles within a specified range from their position, causing those tiles to catch fire. This results in ongoing damage affecting the entire swarm situated on those tiles. In order to add this special unit to the game you need to make the following updates:

- In the `Tile` class do the following:
  - Add a `private boolean` field indicating whether or not the tile is on fire. This should be set to `false` by default.
  - Add a `public` method called `setOnFire()` that updates the corresponding field to `true`.
  - Add a `public` method called `isOnFire()` that returns the value stored in the corresponding field.
- In the `Hornet` class do the following:
  - Add a `public static int` called `BASE_FIRE_DMG` as a field. This field stored the amount of damage hornets take from being positioned on a tile that is on fire.
  - Modify the `takeAction()` method. Hornets positioned on a tile that is on fire receive `BASE_FIRE_DMG` damage each time they begin taking an action.
- Write a class `FireBee` derived from the `HoneyBee` class. This class should have the following fields:
  - A `private int` indicating the maximum attack range of the bee.
  - A `public static int` called `BASE_HEALTH` indicating the base health of fire bees.
  - A `public static int` called `BASE_COST` indicating the their base food cost.

  The `FireBee` class has also the following `public` methods:
  - A constructor that takes as input a `Tile` representing the position of the bee, and an `int` indicating its maximum attack range. The constructor uses the inputs to create a bee given the position, the attack range, base health, and base food cost.
  - The `takeAction()` method. If positioned on the path, fire bees will exclusively target tiles occupied by hornets within their range, and that have not been already previously attacked by a fire bee. More precisely, between all of the available targets, they will set on fire the first encountered tile on the path leading from the bee's position to the hornet nest. If the bee is not positioned on the path, then it does not do anything. Note that a fire bee can target tiles with other bees on it, but it should not target the tile on which it is itself positioned. Finally, remember that bees cannot attack hornets positioned on their nest.

- ***Sniper Bees***. These bees are highly skilled marksbees, armed with a long-range stinger enabling them to eliminate hornets from a distance with precision. Their two-step attack involves aiming before releasing a potent shot that pierces through multiple hornets in a straight line, effectively reducing approaching swarms. While incredibly powerful, their slower rate of fire and vulnerability at close range necessitate strategic placement to optimize their effectiveness. To add this special unit to the game, do the following:

- Write a class `SniperBee` derived from the `HoneyBee` class. This class should have the following fields:
    - A `private int` indicating the attack damage of the bee.
    - A `private int` indicating the piercing power of the bee.
    - A `public static int` called `BASE_HEALTH` indicating the base health of sniper bees.
    - A `public static int` called `BASE_COST` indicating the their base food cost.

  The `SniperBee` class has also the following `public` methods:
    - A constructor that takes as input a `Tile` representing the position of the bee, an `int` indicating its attack damage, and another `int` indicating its piercing power. The constructor uses the inputs to create a bee given the position, attack damage, piercing power, base health, and base food cost.
    - The `takeAction()` method. If not positioned on the path that leads from hive to nest, sniper bees do not do anything and the method returns `false`. On the contrary, if situated on the path, the bee scans for the first non-empty swarm along the route to the hornet nest from its tile. The sniper bee's shot inflicts damage equivalent to its attack power on the first n hornets in the swarm, where n is either equal to the bee's piercing power or the size of the swarm (whichever is smaller). But remember, sniper bees take two turns to shoot. This means that `takeAction()` should alternate between *aiming* (i.e. not doing anything), and shooting. The method returns `true` only when the shot has been released. As mentioned before, remember that bees cannot attack hornets positioned on their nest.

- ***Hornet Queen***. Almost twice the length of an average hornet, the leader of the hornets commands fear. With astonishing speed, she can execute two attacks within the time it takes a regular hornet to launch just one. Furthermore, when this formidable leader enters the battlefield, the entire swarm experiences a significant morale boost, resulting in rapid health regeneration. To add this special unit to the game do the following:
    - Inside the `Insect` class add a `public void` method called `regerateHealth()`. The method takes a `double` as input indicating the percentage of the health to be regenerated. it adjusts the insect's current health based on the provided percentage. The result should be rounded toward 0. For instance, if the insect has health equal to 43 and the method should regenerate its health by 5%, upon execution, the insect's health will be updated to 45.
    - Inside the `Hornet` class do the following:
        - Add two fields: a `private boolean` indicating whether or not this hornet is a queen. And a `private int` that keeps count of how many queens have been created. These fields should be initialized by default to `false` and `0`, respectively. Should they be `static` or non-`static`?
        - Add a `public` method called `isTheQueen()` that returns whether or not this hornet is the queen.

    - Add a `public void` method called `promote()` that makes this hornet a queen if no other queen has been created yet.
    - Modify the `takeAction()` method to reflect the fact that the queen is able to act twice within the time of a single action. Note that this means that, if on a tile that is on fire, the queen should receive the damage twice (unless with its first act it moves to a tile without fire).
  - Finally, inside the `SwarmOfHornets` class do the following:
    - Add a field: a `public static double` called `QUEEN_BOOST`. This denotes the percentage of health regeneration received by all the hornets in the swarm when the queen joins them.
    - Modify `addHornet()` to reflect the fact that when the queen joins the swarm, all the other hornets receive a `QUEEN_BOOST` health regeneration. Note that the queen's health should not regenerate!