German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2018*
Super Heroes Chess: **Milestone 2**

Deadline: 06.04.2018 @ 23:59

This milestone is a further *exercise* on the concepts of **object oriented programming (OOP)**. By the end of this milestone you should have a working game engine with all its logic, that can be played on the console if needed. The following sections describe the requirements of the milestone. Refer to the **Game Description Document** for more details about the rules.

- Please note that you are not allowed to change the hierarchy provided in milestone 1. You should also conform to the method signatures provided in this milestone. However, you are free to add more helper methods and instance variables.
- All methods mentioned in the document should be `public`. All class attributes should be `private` with the appropriate access modifiers and naming conventions for the getters and setters as needed.
- You should always adhere to the OOP features when possible. For example, always use the `super` method or constructor in subclasses when possible.
- The model answer for M1 is available on the MET website. It is recommended to use this version. A full grade in M1 doesn't guarantee a 100 percent correct code, it just indicates that you completed all the requirements successfully :)
- All exceptions that could arise from invalid actions should be thrown and handled in this milestone, as will be detailed below.
- The orientation of the board we will be using can be found in Fig. 1.

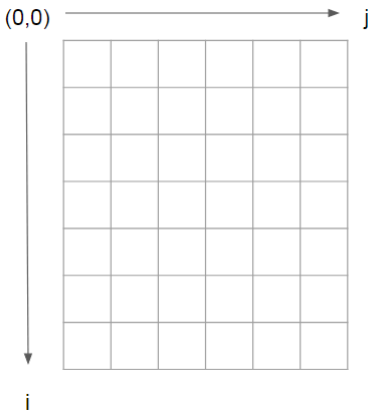

Figure 1: Board orientation with the indices.

1 Game Logic

1.1 `Game` Class

You should add the following methods to the previously built class.

1. `void assemblePieces()`: This method is responsible for assembling the pieces of each player on the game board according to the setup explained in the **Game Description Document**. All pieces that will be added to the board are created in this method. Remember to set the position of the piece itself.
2. `Cell getCellAt(int i, int j)`: This method returns the cell located in position `[i,j]` in the board.
3. `void switchTurns()`: This method is responsible for switching the turns between the two players throughout the game play. You should be careful to call this method after all the turn actions, checks and updates have been performed.
4. `boolean checkWinner()`: This method is responsible for checking the position of the payload to determine whether the current player has won the game. In case the game is won, it should return true and it should return false otherwise. This method will be modified in **Milestone 3** to handle ending the game and notifying the winner in this case.

1.2 `Piece` Class

You should add the following methods to the previously built class.

1. `void attack(Piece target)`: This method is responsible for handling the case where a piece is attacking another `target` piece. This can occur as a result of a movement or a special ability application. It only performs the attacking action and the possible resulting removal of the `target` piece. The moving of the `piece` performing the attack does not occur here. The method should be correctly implemented and overridden in the subclasses. You should take care to update the counters (`sideKilled`, `deadCharacters` and `payloadPos`) accordingly. Consider the following special cases while implementing the method:
   - Attacking an `Armored` with a raised armor.
   - Attacking a `Sidekick`.
   - A `Sidekick` attacking a `Hero` piece.

   This method is also responsible for checking if the game has ended by calling the `checkWinner` method.

1.3 `Movable` Interface

You should add the following methods to the previously built class.

1. `void move(Direction r)`: This method performs any movement action of a movable piece in the give direction `r`. It should be implemented in the corresponding classes accordingly. It is suggested that you utilize the following helper methods to perform the various movements in the various directions and be able to give each piece its allowed movements:
   (a) `moveDown()`
   (b) `moveDownLeft()`
   (c) `moveDownRight()`

(d) `moveLeft()`

(e) `moveRight()`

(f) `moveUp()`

(g) `moveUpLeft()`

(h) `moveUpRight()`

The implementation of the movement methods should handle all of the following:

(a) performing the actual movement.

(b) wrapping the movement if needed.

(c) piece elimination resulting from movements by calling the `attack()` method, as illustrated further in Fig. 2.

(d) ending the turn by calling the `switchTurns()` method.

(e) updating all affected the counters and flags.

Carefully consider where the method will be implemented, as well as where and how it will be overridden given the classes of the various piece types. You should implement helper methods to include repetitive pieces of code. **Carefully consider all the special cases of the superhero types when implementing and overriding this method.**

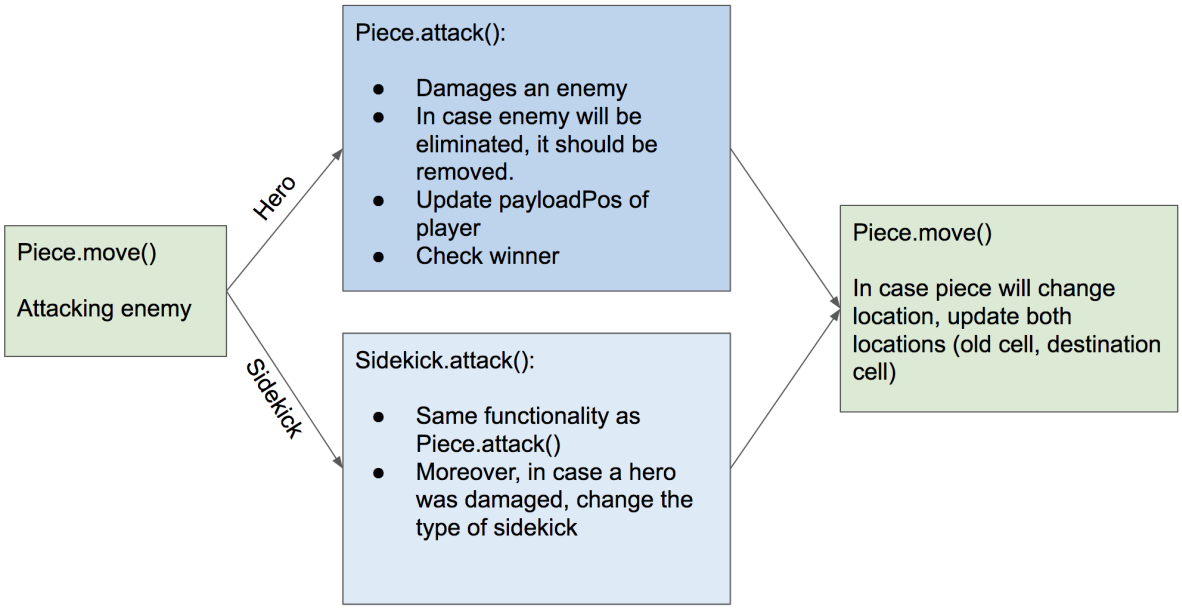

Figure 2: A diagram illustrating the interaction between the `move()` and the `attack()` methods.

1.4 `ActivatablePowerHero` Class

You should add the following methods to the previously built class.

1. `void usePower(Direction d, Piece target, Point newPos)`: This method is responsible for using the power of a hero with activate-able super powers. Depending on the type of power, the `direction`, `target` and `newPos` should either be given as input or as `null` if they are not needed for

defining the power. Be careful to use the `Point` from the `java.awt` package. The method should be correctly implemented and overridden in the subclasses. The common functionality should be implemented in the superclass. To revise the functionality of each of the heroes' super powers you should consult the **Game Description Document**. Remember that the effect of applying a power does not wrap over the board boundaries (unlike the movement). **Carefully consider all the special cases of the superhero types when implementing and overriding this method.**

2 Exception Handling

In this milestone you are required to deal with all the special cases and invalid actions that can arise during gameplay. All exception classes where implemented in milestone 1. In this milestone you are required to throw the correct exceptions in the appropriate situations. You should update the methods implemented above by throwing the necessary exceptions preventing any illegal move from happening. The handling of the exceptions will be done in milestone 3 when integrating the engine with the GUI of the game. Always be sure to make the exception messages as descriptive as possible, as these messages should be displayed whenever any exception is thrown. The following section will aid you in determining where each exception should be thrown. It should be used alongside the the **Game Description Document** to revise the rules as well as the **Milestone 1 Description** to revise the role of each of the implemented exceptions.

1. Before performing any game action (movement or special ability use) the turn needs to be checked. If it's not the correct player's turn a `WrongTurnException` should be thrown.
2. When moving any piece, the direction of movement needs to be checked to ensure it is an allowed movement direction for the piece type. If it is not an allowed move, an `UnallowedMovementException` should be thrown.
3. If the destination of the move of any piece contains a friendly piece then an `OccupiedCellException` should be thrown.
4. After checking the turn, in case of attempting to use the power of any type of hero, it needs to be checked whether the power has already been used. If the power has already been used then a `PowerAlreadyUsedException` needs to be thrown.
5. Medic power use:
   - If a Medic tries to revive an enemy piece an `InvalidPowerTargetException` should be thrown, with a message saying that the target piece belongs to the enemy team.
   - If a Medic tries to revive an alive friendly piece an `InvalidPowerTargetException` should be thrown, with a message saying that the target piece has not been eliminated before, so it cannot be revived.
   - If a Medic tries to revive a dead friendly piece in a direction which would result in an occupied cell (by either a friendly or an enemy piece) then an `InvalidPowerTargetException` should be thrown, with a message saying that the target location is occupied.
6. Ranged power use:
   - If a Ranged tries to use the power in a direction which is not allowed (diagonal) an `InvalidPowerDirectionException` is thrown, with a message saying that this direction is not allowed.

- If a Ranged tries to use its power in a direction with no pieces in that direction an `InvalidPowerDirectionException` is thrown, with a message saying that this direction will result in hitting no enemies.
- If a Ranged tries to use its power in a direction that will hit a friendly piece, an `InvalidPowerDirectionException` is thrown, with a message saying that this direction will result in hitting a friendly piece.

7. Super power use:
   If a Super tries to use the power in a direction that is not allowed by the game rules (diagonal) an `InvalidPowerDirectionException` is thrown, with a message saying that this direction is not allowed by the game rules.
8. Tech power use:
   - If a Tech tries to use teleport on a friendly piece but the target location is occupied by either a friendly or an enemy piece, an `InvalidPowerTargetException` is thrown, with a message saying that the target location is occupied.
   - If a Tech tries to use teleport on an enemy piece, an `InvalidPowerTargetException` is thrown, with a message saying that the target piece doesn't belong to the same team.
   - If a Tech tries to hack a target that already used its power, an `InvalidPowerTargetException` is thrown, with a message saying that the enemy has already used its power and cannot be hacked.
   - If a Tech tries to hack an Armored that already dropped its shield, an `InvalidPowerTargetException` is thrown, with a message saying that the enemy has already used its power and cannot be hacked.
   - If a Tech tries to restore the ability of a friendly piece that did not use its power yet, an `InvalidPowerTargetException` is thrown, with a message saying that the target piece did not use its power yet.
   - If a Tech tries to restore the shield of a friendly Armored that did not drop its shield yet, an `InvalidPowerTargetException` is thrown, with a message saying that the target piece did not use its power yet.