Hacettepe University

Computer Engineering Department

Report of Programming Assignment 3

Programming Languages: Java

Subject: Polymorphism, Exceptions

Name-Surname: **Buse Orak**

Student number: **2200356005**

BBM104: Introduction to Programming Laboratory II (Spring 2021)

Bahar Gezici and Merve Özdeş

Due Date: 07.05.2021 (23:59)

Software Design Notes

Section 1: Problem

This experiment consists of the design of a game of war that goes on between the two sides, Zorde and Calliance. Each side has specific character types with specific abilities, functionalities, and properties. Character types Ork, Troll, and Goblin battle under Zorde, whereas Human, Elf, and Dwarf battle under Calliance.

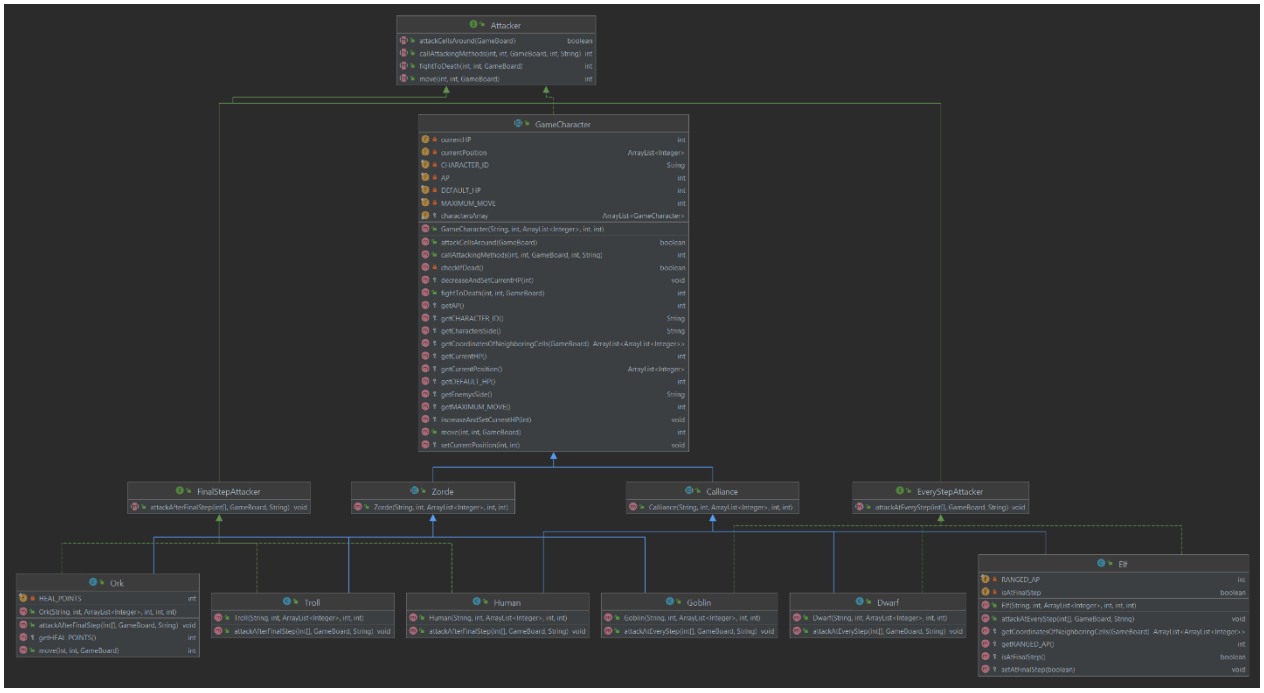
The game is played on a square board where each character occupies one cell with its ID written on it. As the characters from each side makes their moves one after another, since the aim of the game is to kill every opponent side's characters, characters fight by decreasing the defenders' Hit Points by their Attack Powers. According to the special Combat Rules of the game, the characters may attack neighboring cells or they can fight for a specific cell during their lifetime. The game ends when all the characters from one side are dead, which means that their Hit Points are zero or below. The program is exited as it reaches the end of the commands file.

Section 2: Solution

My Approach to the Problem:

- Throughout the project, my main goal was to create such a design that best fits the needs/necessities of the program. While planning the project, the most challenging part, and, in my opinion, the most important part, was to come up with this design so that the program will be
    1. supplied with the organized, effective, and well-contemplated class relationships and
    2. provided with the most important concepts of Object-Oriented Programming such as Encapsulation, Inheritance, Polymorphism, and Abstraction.
- Java does not support Multiple Inheritance; however, for instance, an Ork in this game is both a Zorde character and a final-step attacker, whereas a Dwarf is both a Calliance character and an every-step attacker. Consequently, a character must be an instance of both a `GameCharacter` (either a `Zorde` or `Calliance`) and `Attacker` (either `FinalStepAttacker` or `EveryStepAttacker`). To provide such relationships, I decided to use interfaces together with the usual class inheritance relationships. This is my main approach to the class design and the solution of the problem.

UML Diagrams of the Classes that Have Is-A Relationships:



Explanations of the Classes:

A) The Interface Inheritance Relationships (with the keyword `extends`):

- `interface Attacker`
  - `interface FinalStepAttacker`
  - `interface EveryStepAttacker`

1. `interface Attacker`
   - This interface contains the declarations of four methods that are common for all `Attacker` instances, which are *move*, *callAttackingMethods*, *attackCellsAround*, *fightToDeath*.
2. `interface FinalStepAttacker`
   - This interface extends the interface `Attacker` since all instances of `FinalStepAttacker` are instances of `Attacker` as well.
   - It contains the method declaration *attackAfterFinalStep*.
3. `interface EveryStepAttacker`
   - This interface extends the interface `Attacker` since all instances of `EveryStepAttacker` are instances of `Attacker` as well.
   - It contains the method declaration *attackAtEveryStep*.

Through this choice of design, I managed to separate the characters with different types of attack times from each other and distinguish them. In addition, I gained the opportunity of using these two subclasses as reference types, by this way, I was able to write more general, clean and reusable code in the methods of the class `GameCharacter`, without being have to write the specifications of final-step attackers and every-step attackers in each 6 subclass.

B) <u>The Inheritance Relationships (with the keyword `extends`) and the Implementation Relationships (with the keyword `implements`)</u>:

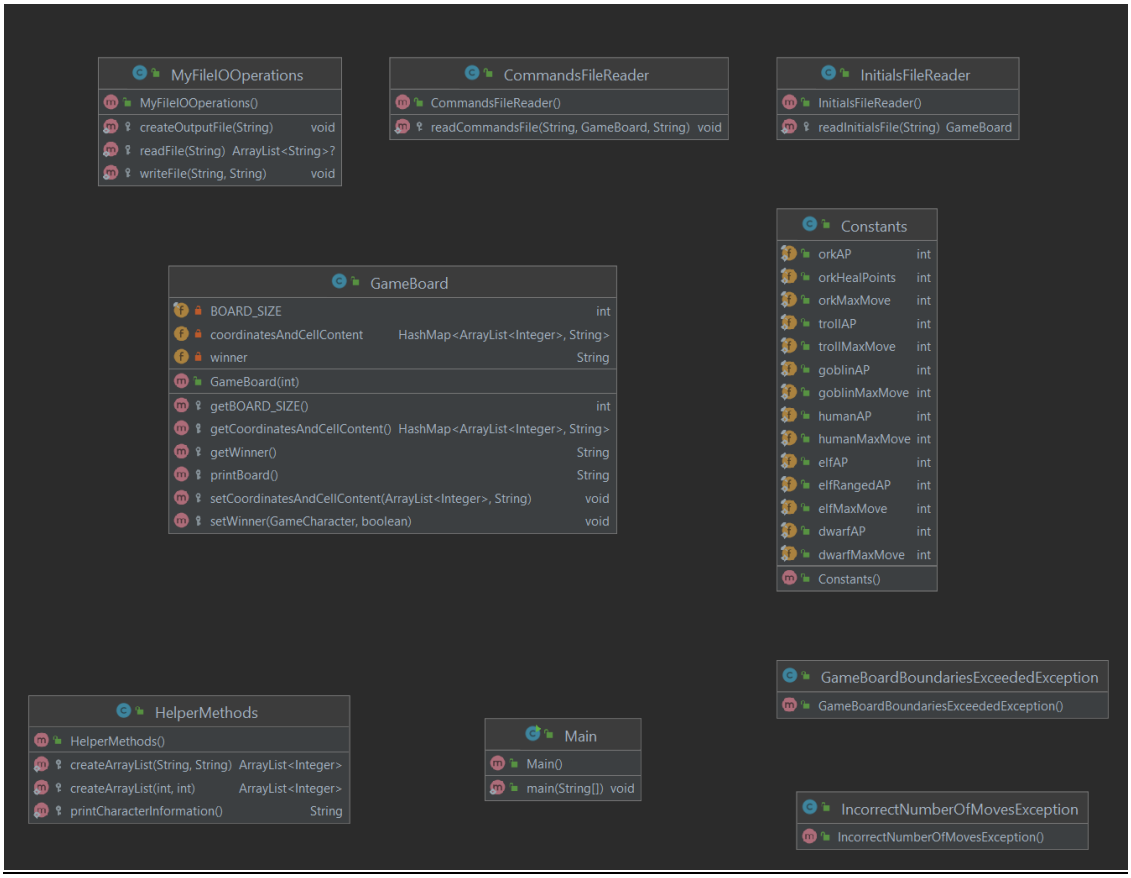
- `abstract class GameCharacter implements Attacker`
  - `abstract class Zorde extends GameCharacter`
    - `class Ork extends Zorde implements FinalStepAttacker`
    - `class Troll extends Zorde implements FinalStepAttacker`
    - `class Goblin extends Zorde implements EveryStepAttacker`
  - `abstract class Calliance extends GameCharacter`
    - `class Human extends Calliance implements FinalStepAttacker`
    - `class Elf extends Calliance implements EveryStepAttacker`
    - `class Dwarf extends Calliance implements EveryStepAttacker`

1. `abstract class GameCharacter`
   - This class is the superclass of all the character classes, including the fields and methods that are common for all characters. Since it is not reasonable to instantiate it, it is declared `abstract`. It can be used as a reference type.
   - This class extends and implements the methods (*move*, *callAttackingMethods*, *attackCellsAround*, *fightToDeath*) of the interface `Attacker` since all `GameCharacter` objects are attackers, even though they differ in when and how they attack (which is where subclass specifications are needed, as explained in section A).
2. `abstract class Zorde` and `abtract class Calliance`
   - These classes are the subclasses of `GameCharacter` class. They are declared `abstract` since it is not reasonable to instantiate them; however, they are used as reference types still.
3. `class Ork`
   - Since the Ork characters attack at their final step, class `Ork` extends interface `FinalStepAttacker`, which extends `Attacker`. Through this design choice, I had the capability of treating an instance of `Ork` both as a `GameCharacter`, specifically, a `Zorde` character, and an `Attacker`, specifically, a `FinalStepAttacker` (but never as an `EveryStepAttacker`. The method *attackAtEveryStep* is never accessible to the instances of the class `Ork`.).

- This class overrides the method *move* (inherited from class `GameCharacter`, which was containing it as the implementation of the `Attacker` method) because `Ork` characters heal their allies and themselves, which is a specific functionality that other classes do not have.
- This class implements the method *attackAfterFinalStep*, taken from the interface `FinalStepAttacker` that it implements.

4. `class Troll`
   - Due to the same reasons, class `Troll` extends class `Zorde` and implements the interface `FinalStepAttacker`.
   - It implements the method *attackAfterFinalStep*, taken from the interface `FinalStepAttacker` that it implements.
5. `class Goblin`
   - Class `Goblin` extends class `Zorde` and implements the interface `EveryStepAttacker`.
   - It implements the method *attackAtEveryStep*, which is taken from the interface `EveryStepAttacker` that it implements.
6. `class Human`
   - Class `Human` extends class `Calliance` and implements the interface `FinalStepAttacker`.
   - It implements the method *attackAfterFinalStep*, taken from the interface `FinalStepAttacker` that it implements.
7. `class Elf`
   - Class `Elf` extends class `Calliance` and implements the interface `EveryStepAttacker`.
   - It implements the method *attackAtEveryStep*, which is taken from the interface `EveryStepAttacker` that it implements.
   - Additionally, this class overrides the method *getCoordinatesOfNeighboringCells* (inherited from class `GameCharacter`) since how it performs its attacks is different than others and needs specification. If the `Elf` character will attack the cells around it, if it is its final attack, this overridden method makes it attack the cells in a range of 2 cells, if it is its not final attack, the superclass version of the method is called and its result is returned.
8. `class Dwarf`
   - Class `Dwarf` extends class `Calliance` and implements the interface `EveryStepAttacker`.
   - It implements the method *attackAtEveryStep*, which is taken from the interface `EveryStepAttacker` that it implements.

UML Diagrams of the Classes that Do Not Have Any Is-A Relationships:



Where I Have Achieved the Use of Polymorphism:

- The object creation while reading the initials file (in method *readInitialsFile*)
    - All the objects are created with reference type `GameCharacter` although their object types are different.
- The `ArrayList` of type `GameCharacter` in the class `GameCharacter`
    - Since the inheritance relationship is defined, I have stored the `GameCharacter`-referenced objects that are instances of either the class `Ork`, `Troll`, `Goblin`, `Human`, `Elf`, or `Dwarf`.
- Overriding the inherited methods
    - In the class `Ork` the method *move* inherited from the indirect superclass `GameCharacter` is overridden in order to provide the instances of `Ork` the functionality of healing themselves and their allies.
    - In the class `Elf` the method *getCoordinatesOfNeighboringCells* inherited from the indirect superclass `GameCharacter` is overridden in order to provide the instances of `Elf` the functionality of making a ranged attack in their final step of their move sequence.
- The use of abstraction with interfaces
    - As explained in detail above, I have used interfaces with inheritance relationships in order to achieve abstraction and polymorphic relationships with the classes that implement them. Consequently, I have achieved interface-based compatibility of class types.

Where I Have Used Custom Exception Classes:

- I have created two classes named `GameBoardBoundariesExceededException` and `IncorrectNumberOfMovesException` that extend the class `Exception` in order to handle the exceptional events that may occur over the course of the game.

Works Cited

Deitel, Paul, and Harvey Deitel. *Java - How to Program*. Pearson Education, 2012.

University of Texas at Austin Computer Science Department. www.cs.utexas.edu/~mitra/csFall2011/cs312/lectures/interfaces.html. Accessed 2 May 2021.

Hacettepe University BBM102 Lecture Notes