The Paintball Game

Object-Oriented Programming
1st Project, version 1.2, 2024-03-30
Deadline until 5pm on April 30, 2024

1 Introduction

This document describes the 1st OOP course project of the 1st year of the degree in Computer Science. It also describes the work methodology students must follow to develop the project.

The project must be carried out in groups of 2 students. Students can and should clarify any doubts with the teaching team, as well as discuss with their colleagues, always respecting the Code of Ethics principles available on the course page.

This project's goal is to program, in Java, "Paintball Game" described in this document.

2 Game Description

The goal is to build a turn-based strategy game, played by `teams`, consisting of a `map`, where you can find `bunkers`, and `players`. The game is played by alternating between different teams, with each team having multiple players. The goal of each team is to eliminate other teams, while avoiding being eliminated. The game ends when there is only one team left, the winner.

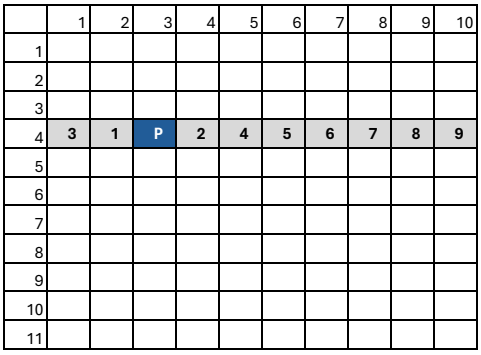
The map is a rectangular grid with given horizontal (rectangle's width) and vertical dimensions (rectangle's height), which is divided into *width* × *height* positions. Each position is identified by a pair (x_horizontal, y_vertical). The vertical coordinate is assigned from top to bottom, and the horizontal coordinate is assigned from left to right. Both the first vertical and the first horizontal coordinate are identified with the number 1. For example, the position (1, 1) is in the upper left corner of the grid. A given position may be empty or contain: (1) a bunker without a player, (2) a bunker with a player, or (3) a player.

Each bunker has a name, a location (cell) on the map, and a treasury. The treasury is a certain amount of coins. The bunker may belong to a team, or be `abandoned`. In a bunker there may or may not be a player of the team. On each turn of the game, the bunker generates another coin for your treasury.

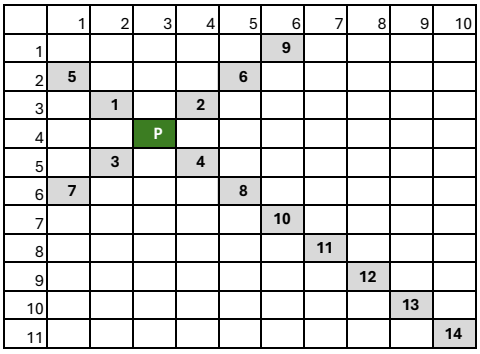
Each team has a name, can have several bunkers, and multiple players. To be active in the game a team must have at least one bunker or one player. The number of players in a team never exceeds the number of available positions on the map.

Players are **created** in a team's bunker, with a cost. To create a player, the bunker needs to have enough coins to cover the cost and, additionally, it must be free, i.e., it cannot have any players inside. The created player is automatically placed inside the bunker, i.e., in the same position as the bunker. There are 3 types of players each with its associated cost: `red` (4 coins), `green` (2 coins), and `blue` (2 coins).

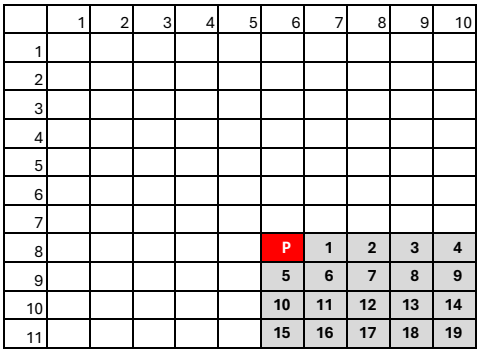
Players **move**. They can move `north`, `south`, `east`, or `west`, one position at a time, and cannot occupy the position of another player from the same team. Moving `north` corresponds to decreasing the position's vertical coordinate, going `south` corresponds to increasing this



(a) Blue player.



(b) Green player.



(c) Red player.

Figure 1: Players' movement order on command `attack`.

coordinate, going `east` increases the horizontal coordinate, and going `west` decrements the horizontal coordinate. Depending on the player's type, the movement may consist of a single move (`blue` and `green`) or 3 moves (`red`).

If a player tries to move to a position that is occupied by an opponent (i.e., a player from another team), a fight happens. The defeated player is eliminated. The fighting rules are as follows:

- A `red` player always wins over a `blue` player.
- A `blue` player always wins over a `green` player.
- A `green` player always wins over a `red` player.
- When two players of the same type (e.g., two `red` players) face each other, the attacker always wins, i.e., the player who moved wins.

If a team's player moves to a position containing a bunker, they seize the bunker for their team, regardless of the bunker being `abandoned` or belonging to another team.

The player may have to fight if the bunker is occupied by an opposing player. In this case, they only seize the bunker if they win the fight. Note that it may happen that one of the teams is left without players and bunkers, which means that team becomes inactive in the game. The game ends when there is only one active team left.

Players can **attack** without leaving their position. However, this only happens in coordination, meaning an attack is carried out by all the players in the team, by their creation order. Each player type moves differently:

- **Blue** players fight with all players from opposing teams who are in the same horizontal coordinate. The attack alternates between `west` and `east`, whenever possible. E.g, a `blue` player in position (3,4) of a grid with 10 × 11 positions, attacks the following positions, by this order: (2,4); (4,4); (1,4); (5,4); (6,4); (7,4); (8,4); (9,4), and (10, 4). This example is depicted in Figure 1a.

- **Green** players fights with all the players from opposing teams that are on the diagonals of the grid that go through the `green` plateyr's position. The attack alternates between the 2 diagonals from `north` to `south` and from `west` to `east`, whenever possible. E.g., a `green` player in position (3,4) of a grid with 10 × 11 positions, attacks the following positions, by the order: (2,3); (4,3); (2,5); (4,5); (1,2); (5,2); (1,6); (5,6); (6,1); (6,7); (7,8); (8,9); (9,10), and (10,11). This example is depicted in Figure 1b.

- **Red** players fight with all the players from opposing teams that are in the rectangular area, whose position represents the upper left corner of that rectangle. The attack is carried out from `west` to `east` and from `north` to `south` in the largest rectangle. E.g., a red player in position (6,8) of a grid with 10 × 11 positions, attacks the following positions, by the order: (7,8); (8,8); (9,8); (10,8); (6,9); (7,9); (8,9); (9,9); (10,9); (6,10); (7,10); (8,10); (9,10); (10,10); (6,11); (7,11); (8,11); (9,11), and (10,11). This example is depicted in Figure 1c.

In these attacks each player can carry out a sequence of fights. If they are eliminated, the attack immediately ends. Note that, in each fight the current team can become inactive (if the team looses all bunkers and all its players), or the game ends (only one team remains active). In these cases, the attack ends for all remaining players, if there are any.

2.1 Beginning of the Game

A new game starts by establishing the map's dimensions, the number of teams and the number of existing bunkers.

E.g., we can establish an horizontal dimension (width) of 10 and a vertical dimension (height) of 20, resulting in a map with 10 × 20 positions (i.e., the valid horizontal coordinates vary between 1 and 10 and the vertical vary between 1 and 20). The upper left corner of the map is identified by the pair (1, 1) and the lower right corner by (10, 20). The number of teams is between 2 and 8, so the number of bunkers is at least one per team, and at most one per position on the map. E.g., on a 10 × 20 map, if there are 3 teams, one can create from 3 to 200 bunkers. At the beginning of the game there can only be one bunker in each position.

Then, once the number of teams and bunkers is known, the bunkers will be created and added to the game. Finally, the teams will be added to the game. Each team is assigned an `abandoned` bunker and starts with no players in the army. There must be, at least, two teams for a game to be created.

2.2 Turns

Teams play alternately, by the order in which they were created. The first turn is played by the first team, the second turn by the second, and so on, alternating between the teams that are still active, until only one team remains. On each turn, a team can make an arbitrary number of queries, but can only perform one action that modifies the state of the game: **create** a player, **move** a player, and **attack** with all the players in the team. These actions may not change the game's state, however, this team's turn ends and it is the next active team's turn.

3 Application Specification

The application interface is intended to be simple, so that it can be used in different environments and allows the testing process to be automated. For these reasons, input and output must respect the precise format specified in this section. You can assume the input complies with the indicated values and formats, i.e., the user makes no mistakes, other than those provided in this description. The program reads lines from the standard input (`System.in`), writes lines to the standard output (`System.out`), and is case insensitive (for example, the words "quit", "quIT", "Quit" and "QUIT" they are the same).

User interaction is done through commands. When starting the application, the game is not active, and the only commands available are:

- `game` - start a new game.
- `help` - view the available commands.
- `quit` - quit the application.

At this point, the entry prompt is the symbol >. While there is an active game the prompt is teamName>, with `teamName` being the name of the team that will perform the next move. In the remaining of this document, this team is referred to as the "current team". If there is a game running, all the following commands are available:

- `game` - Start a new game.
- `move` - Move a player.
- `create` - Create a player in a bunker.
- `attack` - Attack with all players of the current team.
- `status` - List the current state of the game.
- `map` - List the map of the current team.
- `bunkers` - List all bunkers of the current team, by the order they were seized.
- `players` - List all active players of the current team, by the order they were created.
- `help` - Display all available commands.
- `quit` - Exit the application.

Next, the commands offered by the application will be described. In the following examples, the symbol ↵ denotes the newline and, when executing a command, we differentiate between `text written by the user`, and feedback written by the program on the console. If the user enters a command that does not exist, or is not available at the time, the program writes the message "Invalid command." in the console.

3.1 Command `quit`

Ends the program execution, even if the game gets interrupted. It does not require any arguments. The program ends by writing a line to the console with the message "Bye.".

The following examples illustrates the two scenarios where this command can be invoked:

1. No game in progress:

```
> quit↵
Bye.↵
```

2. Game in progress, with the current team being `teamName`:

```
teamName> quit↵
Bye.↵
```

3.2 Command `help`

Informs the user about the available commands. It does not require any arguments and always succeeds. The command's results depend on usage context, two alternative scenarios should be considered: (1) when the command is invoked and there is no game in progress, and (2) when the command is invoked during a game. These scenarios are illustrated in the following examples:

1. No game in progress:

```
> help↵
game - Create a new game↵
help - Show available commands↵
quit - End program execution↵
>
```

2. A game is in progress and `teamName` is the current team's name:

```
teamName> help↵
game - Create a new game↵
move - Move a player↵
create - Create a player in a bunker↵
attack - Attack with all players of the current team↵
status - Show the current state of the game↵
map - Show the map of the current team↵
bunkers - List the bunkers of the current team, by the order they were seized↵
players - List the active players of the current team, by the order they were created↵
help - Show available commands↵
quit - End program execution↵
teamName>
```

3.3 Command **game**

Starts a new game. A new game can be started at any time. If you attempt to start a new game while there is a game in progress, this action will cancel the game in progress.

At the start of a new game the map dimensions are provided (width and height). These values must be integers greater or equal than 10. The smallest possible map should have 10×10 positions, with the upper left corner at position (1,1). Next, the number of teams are provided (an integer in the range $[2, \cdots, 8]$), and the number of bunkers to be created (an integer in the range $[numberOfTeams, \cdots, (width \times height)]$). Next, there should be one loop reading the bunker data, and another reading the team data. For each bunker, we provide the horizontal (x axis) and vertical (y axis) coordinates, the initial treasury (a natural number), and the name of the bunker (a string of characters that can have spaces and ends at the end of the line). For each team, we provide the name of the team (a word with no spaces) and the name of a bunker.

Both bunker and team's names should be read as to preserve upper and lower case letters, as stated by the user when creating these entities. All coordinates should be integer values. The overall command format is as follows (parameters have the style **bold**):

```
> game width height teamsNumber bunkersNumber↵
bunkersNumber bunkers:↵
xCoordinate yCoordinate treasure bunkerName↵
xCoordinate yCoordinate treasure bunkerName↵
...↵
xCoordinate yCoordinate treasure bunkerName↵
teamsNumber teams:↵
teamName bunkerName↵
teamName bunkerName↵
...↵
teamName bunkerName↵
```

If the game is successfully created the prompt becomes teamName>, with `teamName` being the name of the first created team.

E.g., suppose you want to start a game with a 20×20 map, with 3 teams called *Lancaster*, *York*, and *Tudor* and 4 bunkers called *Pontefract*, *Sandal*, *Windsor*, and *Thornbury*. In this scenario, the game is created and the first to play is the *Lancaster* team, as shown in the following example:

```
> game 20 20 3 4↵
4 bunkers:↵
10 15 4 Pontefract↵
12 20 1 Middleham↵
1 3 10 Windsor↵
15 16 2 Thornbury↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster>
```

To simplify our program, we will assume that the first line of the `game` command is correct: (1) the horizontal (x) and vertical (y) coordinates that define the map are positive integers greater than or equal to 10, (2) the number of teams is a positive number between 2 and 8, and (3) the number of bunkers is greater than or equal to the number of teams.

Reading bunker data: when reading bunker data, the bunker is created only if: (1) the location is a valid position on the map, (2) the treasury is a positive integer, (3) the name of the bunker does not exist in the game, and (4) the indicated location is not occupied. If any of these requirements are not met, the error message "Bunker not created." is displayed.

Reading team data: when reading team data, the team is created only if: (1) there is no other team with the same name in the game, (2) the intended bunker exists in the game, and (3) it does not belong to another team in the game. If any of these requirements are not met, the error message "Team not created." is displayed. Note that only teams occupying `abandoned` bunkers are created.

The following two examples show a game being created, even though the creation of teams and bunkers do not comply with the previous requirements:

1. Three bunkers and two teams are created:

```
> game 20 20 3 5↵
5 bunkers:↵
10 15 4 Pontefract↵
12 20 -1 Windsor↵
Bunker not created.↵
1 3 10 Middleham↵
7 -2 8 Hampton Court↵
Bunker not created.↵
15 16 2 Thornbury↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Team not created.↵
Lancaster>
```

2. Two bunkers and two teams are created:

```
> game 20 20 3 5↵
5 bunkers:↵
10 15 4 Middleham↵
12 20 -1 Windsor↵
Bunker not created.↵
1 3 10 Pontefract↵
7 -2 8 Thornbury↵
Bunker not created.↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
Tudor Windsor↵
Team not created.↵
York Middleham↵
Lancaster>
```

However, there is a scenario where errors in creating bunkers and teams prevent the game from being created. This scenario happens when the number of successfully created teams is less

than 2. In this case, the error message "FATAL ERROR: Insufficient number of teams." should be displayed. If there was a game running and the new game creation fails, the previous game should be cancelled.

The following examples show two scenarios where the game is not created due to a fatal error in bunker and team creation:

1. Two bunkers and one team are created, therefore the game is not created:

```
> game 20 20 3 5↵
5 bunkers:↵
10 15 4 Middleham↵
12 20 -1 Windsor↵
Bunker not created.↵
1 3 10 Pontefract↵
7 -2 8 Thornbury↵
Bunker not created.↵
15 16 2 Middleham↵
Bunker not created.↵
3 teams:↵
Lancaster Thornbury↵
Team not created.↵
York Middleham↵
Tudor Windsor↵
Team not created.↵
FATAL ERROR: Insufficient number of teams.↵
>
```

2. Five bunkers and one team are created, therefore the game is not created:

```
> game 20 20 3 5↵
5 bunkers:↵
10 15 4 Middleham↵
12 20 1 Windsor↵
1 3 10 Pontefract↵
7 2 21 Thornbury↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Tutbury↵
Team not created.↵
York Middleham↵
York Pontefract↵
Team not created.↵
FATAL ERROR: Insufficient number of teams.↵
>
```

3.4 Command `status`

Displays information on the current state of the game. This command is only available when there is a game in progress. It does not require any arguments. Describes the current state of the game with respect to: (1) the size of the map, (2) the bunkers on the map and their

owners, including `abandoned` ones, by the order they were created, and (3) the teams that are still playing, by their turn order.

With **owner_i** being the name of the team to which the bunker with name **bunkerName_i** belongs. If the bunker is `abandoned` it is displayed as **without owner**. The game description format should be as follows:

```
teamName> status↵
xDimension yDimension↵
bunkersNumber bunkers:↵
bunkerName_1 (owner_1)↵
bunkerName_2 (owner_2)↵
...↵
bunkerName_bunkersNumber (owner_bunkersNumber)↵
activeteamsNumber teams:↵
teamName_1; teamName_2; ...  teamName_activeteamsNumber↵
teamName>
```

The following example shows the game's description, right after its creation:

```
> game 20 20 3 4↵
4 bunkers:↵
10 15 4 Middleham↵
12 20 1 Windsor↵
1 3 10 Pontefract↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster> status↵
20 20↵
4 bunkers:↵
Middleham (York)↵
Windsor (Tudor)↵
Pontefract (Lancaster)↵
Hampton Court (without owner)↵
3 teams:↵
Lancaster; York; Tudor↵
Lancaster>
```

3.5 Command `map`

Current team's map view of players and bunkers. This command is only available when there is a game in progress. It does not require any arguments, and always succeeds. It shows the horizontal and vertical dimension of the game map in the first line, followed by all the map's positions (line by line). Each position can have one of the following characters:

- `.` when there is neither a bunker nor a player in that position;
- `B` when there is a free bunker (without players) in that position;
- `O` when there is an occupied bunker (with a player) in that position;

- `P` when there is a player from the current team in that position.

The following example shows a map right after creating a new game. In this scenario, there is only one bunker in the map, since all teams are created with one bunker without players.

```
> game 10 15 3 4↵
4 bunkers:↵
10 12 4 Middleham↵
1 12 1 Windsor↵
1 3 10 Pontefract↵
1 6 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster> map↵
10 15↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . .↵
 2 . . . . . .↵
 3 B . . . . . .↵
 4 . . . . . .↵
 5 . . . . . .↵
 6 . . . . . .↵
 7 . . . . . .↵
 8 . . . . . .↵
 9 . . . . . .↵
10 . . . . . .↵
11 . . . . . .↵
12 . . . . . .↵
13 . . . . . .↵
14 . . . . . .↵
15 . . . . . .↵
Lancaster>
```

3.6 Command `bunkers`

Informs about the bunkers in the current team. This command is only available when there is a game in progress. It does not require arguments. Lists the name, treasury and location of all bunkers in the current team, by the order in which they were seized or occupied (this also includes the initial occupation of a bunker by a team, at the beginning of the game).

The listing has the following format, with **N** being the number the team's bunkers, and **teamName** being the name of the team:

```
teamName> bunkers↵
N bunkers:↵
bunkerName_1 with treasure_1 coins in position (xCoord_1, yCoord_1)↵
bunkerName_2 with treasure_2 coins in position (xCoord_2, yCoord_2)↵
...↵
bunkerName_N with treasure_N coins in position (xCoord_N, yCoord_N)↵
teamName>
```

If the current team (whose name is `teamName`) does not have bunkers, it should write the message **"Without bunkers."**, as follows:

```
teamName> bunkers↵
Without bunkers.↵
teamName>
```

The following example illustrates the creation of a game and the list of bunkers in the *Lancaster* team, right after its creation:

```
> game 20 20 3 4↵
4 bunkers:↵
10 15 4 Middleham↵
12 20 1 Windsor↵
1 3 10 Pontefract↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster> bunkers↵
1 bunkers:↵
Pontefract with 10 coins in position (1, 3)↵
Lancaster>
```

Suppose that, after a few turns, the *Lancaster* team seized *Windsor* bunker with a treasury of 3, and that *Pontefract's* current treasury is 12. The result of the `bunkers` command would be as follows:

```
Lancaster> bunkers↵
2 bunkers:↵
Pontefract with 12 coins in position (1, 3)↵
Windsor with 3 coins in position (12, 20)↵
Lancaster>
```

3.7 Command `players`

Informs about the players of the current team. This command is only available when there is a game in progress. It does not require arguments. Lists all active players in the current team, by the order in which they were recruited. The listing has the following format, where **playerType_i** is the player's type (`red`, `blue`, or `green`):

```
teamName> players↵
N players:↵
playerType_1 player in position (xCoord_1, yCoord_1)↵
playerType_2 player in position (xCoord_2, yCoord_2)↵
...↵
playerType_N player in position (xCoord_N, yCoord_N)↵
teamName>
```

If the active team does not have players, the message **"Without players."** should be written. This always happens after creating a game, as you can see in the following example:

```
> game 20 20 3 4↵
4 bunkers:↵
10 15 4 Middleham↵
12 20 1 Windsor↵
1 3 10 Pontefract↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster> players↵
Without players.↵
Lancaster>
```

Suppose that, after some time, the *Lancaster* team successively recruited a `red` player, two `blue` players, a `green` player, and another `red` player. The command result should be presented as follows:

```
Lancaster> players↵
5 players:↵
red player in position (4, 3)↵
blue player in position (3, 3)↵
blue player in position (2, 2)↵
green player in position (1, 4)↵
red player in position (2, 3)↵
Lancaster>
```

Suppose that, after a few turns, the *Lancaster* team loses both the `blue` player in position (3, 3) and the `green` player in position (1, 4), while the other players remained in the same place. Next, the *Lancaster* team creates another `green` player. The result should be as follows:

```
Lancaster> players↵
4 players:↵
red player in position (4, 3)↵
blue player in position (2, 2)↵
red player in position (2, 3)↵
green player in position (1, 3)↵
Lancaster>
```

3.8 Command `create`

Create a player in a bunker. This command is only available when there is a game in progress. players are created in an unoccupied bunker of the team (i.e., without any players inside), with a cost: `red` player (4 coins), `green` player (2 coins), and `blue` player (2 coins). This command has 2 arguments: the player's type and the bunker's name.

```
teamName> create playerType bunkerName↵
```

The command is successful if the bunker exists and belongs to the team with name **teamName** (current team), the bunker is free and there are enough coins in the bunker to create the

player. When successful, the program should write the message **"playerType player created in bunkerName"**.

In the following example, we start by querying the bunkers in the team named *Tudor*, and then creating a `green` player in the *Windsor* bunker.

```
Tudor> bunkers↵
1 bunkers:↵
Windsor with 6 coins in position (12, 20)↵
Tudor > create green Windsor↵
green player created in Windsor↵
```

Creating a player can result in some error scenarios. Assume the following game was created:

```
> game 20 20 3 4↵
4 bunkers:↵
10 15 4 Middleham↵
12 20 1 Windsor↵
1 3 10 Pontefract↵
15 16 2 Hampton Court↵
3 teams:↵
Lancaster Pontefract↵
York Middleham↵
Tudor Windsor↵
Lancaster>
```

Now, consider the following error scenarios and respective examples:

1. If the player type is invalid, the program should write the message **"Non-existent player type."**. Consider the following example where the *Lancaster* team tries to create an `yellow` player, which is a player type not supported by the game. We also illustrate that the attempt to create a player, whether successful or unsuccessful, takes one turn, changing the game control to the next team (*York*).

   ```
   Lancaster> create yellow Pontefract↵
   Non-existent player type.↵
   York>↵
   ```

2. If the name of the bunker does not exist, the program should write the message **"Non-existent bunker."**.

3. If the bunker does not belong to the current team, the program should write the message **"Bunker illegally invaded."**, shown in the following example:

   ```
   Lancaster> create green Windsor↵
   Bunker illegally invaded.↵
   York>↵
   ```

4. If the bunker belongs to the current team, but is occupied with a player from the same team, the program should write the message **"Bunker not free."**.

5. If the bunker belongs to the current team but does not have enough funds, the program should write the message **"Insufficient coins for recruitment."**.

After executing the command the current team's turn ends, and the next active team starts to play, regardless of whether or not a player has been successfully recruited. Moreover, all bunkers generate a coin since it's the end of a turn.

3.9 Command `move`

Moves a player. This command is only available when there is a game in progress. It requires three arguments: player coordinates (horizontal and vertical), and a direction the movement should follow. If the player in the given position is a `red` player then there must be two more arguments (the directions of the 2 extra moves). Remember `green` and `blue` players can only make a move on each turn. However, `red` players can perform three moves in each movement. The command's format is one of the following, with **x**, **y**, and **dir_i** the position coordinates, and the player's movement direction, respectively:

```
teamName> move x y dir_1↵

teamName> move x y dir_1 dir_2 dir_3↵
```

Each direction can take the values `north`, `south`, `east`, and `west`. The first format, with a direction, is used by the slower players (`green` and `blue` players). The second format, with 3 directions, is used by the `red` players.

If a `red` player is moving, the 3 movements are independent, i.e., even if one movement is unsuccessful (it does not change the player's position), if the `red` player is not eliminated, then they must perform the remaining movements.

Each movement may or may not change the player's position. If the position changes, at the end of each full or partial (`red` player) movement, the program must report the new player's position as follows:

```
playerType player in position (x_new, y_new)↵
```

Remember that going `north` corresponds to decreasing the position's vertical coordinate (y), going `south` increases this coordinate, going `east` increases the horizontal coordinate (x), and going `west` decreases the horizontal coordinate.

In the following example, the three active teams successively move their players. To ease your analysis, in the first turn of each team, we list the respective army, and all move commands succeed.

```
Lancaster> players↵
4 players:↵
red player in position (1, 3)↵
blue player in position (2, 2)↵
red player in position (2, 3)↵
green player in position (1, 1)↵
Lancaster> move 1 3 south east east↵
red player in position (1, 4)↵
red player in position (2, 4)↵
red player in position (3, 4)↵
```

```
York> players↵
1 players:↵
blue player in position (10, 15)↵
York> move 10 15 north↵
blue player in position (10, 14)↵
Tudor > players↵
1 players:↵
green player in position (12, 20)↵
Tudor > move 12 20 west↵
green player in position (11, 20)↵
Lancaster> move 2 2 east↵
blue player in position (3, 2)↵
York>
```

The playing ground is a dangerous place and several things can go wrong when a player tries to move. Below are some examples of an interactive session in which one or more errors occur when giving orders to a player. Note that in all described cases (1 to 6) the player does not move.

1. The given position (x, y) is not an existing position on the game map, i.e., it is outside map limits. The program should write the message **"Invalid position."**.

2. The given direction (**dir_i**) is not a valid direction (*south*, *north*, *east*, or *west*). The program should write the message **"Invalid direction."**.

3. There is no player in the given position (x, y). The program should write the message **"No player in that position."**.

   ```
   Lancaster> players↵
   4 players:↵
   red player in position (1, 3)↵
   blue player in position (2, 2)↵
   red player in position (2, 3)↵
   green player in position (1, 1)↵
   Lancaster> move 3 3 north↵
   No player in that position.↵
   York>
   ```

4. The player in the given position (x, y) is not a red player and three movements have been specified. The program must write the message **"Invalid move."**.

5. The player in the position (x, y) is moving to a position already occupied by an active player of the same team (eliminated players do not take up space on the map, completely disappearing from the game when eliminated). The player is unable to move, and the message **"Position occupied."** should be written.

```
Lancaster> players↵
4 players:↵
red player in position (1, 3)↵
blue player in position (12, 9)↵
red player in position (12, 7)↵
green player in position (12, 8)↵
Lancaster> move 12 8 north↵
Position occupied.↵
York>
```

6. The player in position (x, y) is moving off the map. In this case, the message **"Trying to move off the map."** should be written.

```
Lancaster> players↵
4 players:↵
red player in position (1, 3)↵
blue player in position (4, 1)↵
red player in position (2, 3)↵
green player in position (1, 1)↵
Lancaster> move 4 1 north↵
Trying to move off the map.↵
York>
```

In following example on a 20×20 map, a red player is in position (19, 12) and receives the order to move in the `east east south` directions. In this case, they can still perform the first movement, then fail the next one, finally managing to make the third movement:

```
Lancaster> players↵
1 players:↵
red player in position (19, 12)↵
Lancaster> move 19 12 east east south↵
red player in position (20, 12)↵
Trying to move off the map.↵
red player in position (20, 13)↵
York >
```

Besides these error scenarios, there are many other possible scenarios. For instance, fights between players, bunkers being seized, etc. Next, we describe scenarios where the moving player seizes bunkers, fights other players (illustrating both winning those flights and being eliminated):

1. Player in position (x, y) who tries to move to another position already occupied by an `abandoned` bunker, occupies that position (moves) and seized the bunker for their team. In this case, the message **"Bunker seized."** should be written, followed by the player's new position.

2. When moving, the player in the position (x, y) may attempt to take another position occupied by an opponent. In this case, a fight takes place. If the attacking player wins the fight, the opponent is eliminated and the message to write is **"Won the fight."**. Since the player moved, the player's new position should be displayed.

3. However, if the player in the position (x, y) is eliminated during the fight, the message **"Player eliminated."** should be written. If it is a `red` player, it may happen that the player is eliminated in one of the partial movements. If this is the case, the next movements are ignored.

4. Following a fight, the player in the position (x, y) can win over the opponent who was in the bunker, thus acquiring the bunker for their team. In this case, they first eliminated the opponent and then seized the bunker. Therefore, you must first write the message **"Won the fight and bunker seized."**, and finally the player's new position.

5. Following a fight, the game may end. The game can end following a player's movement (full, or partial) because:

   - When the player moves, they seized the last bunker of the last opposing team that no longer had players. In this case, the team of the player who carries out the attack wins the game.
   - When the player moves, they fight and wins over the last opposing player from the last active opposing team with no bunkers, thus winning the game.
   - When the player moves, fights and is eliminated, leaving their team without players and without bunkers, it results in the team becoming inactive. If there is only one active team left, the game ends, and the winner is the opposing team.

   In these scenarios, the message **"Winner is teamName."** should be written, where `teamName` is the name of the winning team.

Bellow, we present several examples of fights, in which the game does not end, and the active teams are *Lancaster*, *York*, and *Tudor*.

1. The attacking player wins a fight. In the following example, the `red` player of *Lancaster* in position (10, 10) tries to move three times `north`, finding, successively, a `red` player of *York*, a `blue` player of *Tudor*, and a third empty position. The red player wins over both the opposing `red` and `blue` players along the way. However, on the next turn, they are eliminated by a *York* `green` player.

```
Lancaster> players↵
4 players:↵
red player in position (10, 10)↵
blue player in position (12, 1)↵
red player in position (12, 17)↵
green player in position (12, 3)↵
Lancaster> move 10 10 north north north↵
Won the fight.↵
red player in position (10, 9)↵
Won the fight.↵
red player in position (10, 8)↵
red player in position (10, 7)↵
York> move 9 7 east↵
Won the fight.↵
green player in position (10, 7)↵
Tudor>
```

2. The attacking `red` player is eliminated. Let's see what happens when a `red` player is eliminated on the first, second, and third movements. In this example the three `red` players are eliminated by opposing `green` players. Naturally, eliminated `red` players stop moving, taking orders, and disappear from the team they were part of.

```
Lancaster> move 4 1 south south south↵
Player eliminated.↵
York> move 4 10 north north north↵
red player in position (4, 9)↵
Player eliminated.↵
Tudor> move 15 10 east east east↵
red player in position (16, 10)↵
red player in position (17, 10)↵
Player eliminated.↵
York>
Lancaster> move 13 20 west↵
Won the fight and bunker seized.↵
blue player in position (12, 20)↵
York>
```

Bellow are some examples of combat scenarios involving the end of the game. It should be noted that after the game ends, the program prompt is the > symbol.

1. In the following example, there are only two active teams: *Lancaster* and *York*. The `red` team from *Lancaster*, in position (10, 10), tries to move `north` three times, finding on the first movement a `red` player from *York*. The `red` player from *Lancaster* wins over the `red` player from *York*, who was the last active member of that team, which also no longer has bunkers. Without an members or bunkers, *York* is eliminated. So the game is won by the team *Lancaster*. In this scenario, the `red` player last two moves are ignored.

```
Lancaster> move 10 10 north north north↵
Won the fight.↵
red player in position (10, 9)↵
Winner is Lancaster.↵
>
```

2. In the following example, the last player of *Lancaster*, the `red` player in position (10, 10), tries to move `north` three times, successively encountering a `red` and a `green` player of *York*. The *Lancaster* `red` player wins over the *York's* `red` player, but is then is eliminated by the `green` player. Because they are the last player in their team, and the team no longer has bunkers, the team loses the game, so the team of *York* wins the game. In this scenario, the `red` team's last movement is ignored.

```
Lancaster> move 10 10 north north north↵
Won the fight.↵
red player in position (10, 9)↵
Player eliminated.↵
Winner is York.↵
>
```

3. In this example, the `blue` player in position (13, 16) moves `east`, finding a free bunker belonging to the only active opposing team. Furthermore, the opposing team has no players left, therefore the team of *Lancaster* is the winner of the game.

```
Lancaster> move 13 16 east↵
Bunker seized.↵
blue player in position (14, 16)↵
Winner is Lancaster.↵
>
```

If the **move** command does not result in the end of the game, the current team's turn ends and the next active team in the game starts to play, regardless of whether or not the player's movement was successfully carried out. Furthermore, all bunkers generate a coin, as it is the end of a turn.

3.10 Command `attack`

Attacking a team. This command is only available when there is a game in progress. It receives no further arguments. When attacking, the players of the current team do not move but attack the opposing players in different ways, depending on their type (`blue` players attacks players in the same horizontal coordinate, `green` players on the diagonals, and `red` players in rectangular areas, as previously explained).

The attack is carried out by all the team's players, by their recruitment order. In these attacks they can seized bunkers, winning or losing fights, turn the current team inactive[1], and even end the game. If the current team continues active, the program writes the map of the team that just carried out the attack (command `map`). If the current team becomes inactive during the attack, the program writes the message **"All players eliminated."**. If the game ends, the message **"Winner is teamName."** should be written, where `teamName` is the name of the winning team.

To illustrate a team's attack, let us assume we have two teams *Lancaster* and *York*, each of them with a bunker, and several players. To make it easier to visualize the attack, we present two code excerpts, showing each teams' players and bunkers:

```
York> players↵
5 players:↵
blue player in position (1, 7)↵
red player in position (5, 6)↵
green player in position (1, 10)↵
blue in position (3, 10)↵
green player in position (10, 1)↵
York> bunkers↵
2 bunkers:↵
bunker White with 12 coins in position (1, 10)↵
bunker Magenta with 10 coins in position (10, 1)↵
York> map↵
10 11↵
```

[1] This only happens if the current team has no bunkers, and all its players are eliminated in this attack.

```
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . . . . . O↵
 2 . . . . . . . . . .↵
 3 . . . . . . . . . .↵
 4 . . . . . . . . . .↵
 5 . . . . . . . . . .↵
 6 . . . . P . . . . .↵
 7 P . . . . . . . . .↵
 8 . . . . . . . . . .↵
 9 . . . . . . . . . .↵
10 O . P . . . . . . .↵
11 . . . . . . . . . .↵
Lancaster> players↵
3 players:↵
red player in position (8, 10)↵
blue player in position (10, 10)↵
green player in position (8, 3)↵
Lancaster> bunkers↵
2 bunkers:↵
Pontefract with 12 coins in position (10, 5)↵
Windsor with 3 coins in position (1, 11)↵
Lancaster> map↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . . . . . B↵
 2 . . . . . . . . . .↵
 3 . . . . . . . P . .↵
 4 . . . . . . . . . .↵
 5 . . . . . . . . . B↵
 6 . . . . . . . . . .↵
 7 . . . . . . . . . .↵
 8 . . . . . . . . . .↵
 9 . . . . . . . . . .↵
10 . . . . . . . P . P↵
11 B . . . . . . . . .↵
```

Example of a *Lancaster* attack: The players involved, be the order in which they attack, are:

1. The `red` player in position (8, 10), when attacking in their rectangular area, does not eliminate any player on the opposing *York* team:

```
Lancaster> players↵
3 players:↵
red player in position (8, 10)↵
blue player in position (10, 10)↵
green player in position (8, 3)↵
Lancaster> bunkers↵
2 bunkers:↵
Pontefract with 12 coins in position (10, 5)↵
Windsor with 3 coins in position (1, 11)↵
```

2. The `blue` player in position (10, 10), when attacking horizontally, fights the *York* `blue` player at (3, 10), winning the fight. Then fights a *York* `green` player at (1, 10) losing the fight and being eliminated:

```
Lancaster> map↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . . . . . .↵
 2 . . . . . . . . . .↵
 3 . . . . . . . P . .↵
 4 . . . . . . . . . .↵
 5 . . . . . . . . . B↵
 6 . . . . . . . . . .↵
 7 . . . . . . . . . .↵
 8 . . . . . . . . . .↵
 9 . . . . . . . . . .↵
10 . . . . . . . P . P↵
11 B . . . . . . . . .↵
Lancaster> attack↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . . . . . B↵
 2 . . . . . . . . . .↵
 3 . . . . . . . . . .↵
 4 . . . . . . . . . .↵
 5 . . . . . . . . . B↵
 6 . . . . . . . . . .↵
 7 . . . . . . . . . .↵
 8 . . . . . . . . . .↵
 9 . . . . . . . . . .↵
10 . . . . . . . P . .↵
11 B . . . . . . . . .↵
```

3. The `green` player at position (8, 3), when attacking on the diagonals, fights a *York* `green` player in (10, 1), winning the fight and seizing the bunker. Then fights the *York* `red` player at (5, 6) losing the fight and being eliminated:

```
York> map↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . . . . . .↵
 2 . . . . . . . . . .↵
 3 . . . . . . . . . .↵
 4 . . . . . . . . . .↵
 5 . . . . . . . . . .↵
 6 . . . . P . . . . .↵
 7 P . . . . . . . . .↵
 8 . . . . . . . . . .↵
 9 . . . . . . . . . .↵
10 O . . . . . . . . .↵
11 . . . . . . . . . .↵
Lancaster>
```

If the `attack` command does not result in the end of the game, the turn of the current team ends, and the next active team starts playing. Furthermore, all bunkers generate a coin, as it is the end of a turn.

4 Complete Example

In this section we present a complete example of a game. Note that this example is not intended to be exhaustive, being merely illustrative of a typical gaming session.

```
> help↵
game - Create a new game↵
help - Show available commands↵
quit - End program execution↵
> game 20 20 3 5↵
5 bunkers:↵
10 15 4 Middleham↵
12 20 -1 Windsor↵
Bunker not created.↵
1 3 10 Pontefract↵
7 -2 8 Thornbury↵
Bunker not created.↵
15 16 2 Middleham↵
Bunker not created.↵
3 teams:↵
Lancaster Thornbury↵
Team not created.↵
York Middleham↵
Tudor Windsor↵
Team not created.↵
FATAL ERROR: Insufficient number of teams.↵
> game 10 11 2 3↵
3 bunkers:↵
3 2 10 Windsor↵
6 5 15 Magenta↵
6 2 7 My↵
2 teams:↵
TeamBlack Windsor↵
TeamMagenta Magenta↵
```

```
TeamBlack> status↵
10 11↵
3 bunkers:↵
Windsor (TeamBlack)↵
Magenta (TeamMagenta)↵
My (without owner)↵
2 teams:↵
TeamBlack; TeamMagenta↵
TeamBlack> help↵
game - Create a new game↵
move - Move a player↵
recruit - Create a player in a bunker↵
attack - Attack with all players of the current team↵
status - Show the current state of the game↵
map - Show the map of the current team↵
bunkers - List the bunkers of the current team, by the order they were seized↵
players - List the active players of the current team, by the order they were created↵
help - Show available commands↵
quit - End program execution↵
TeamBlack> create red Windsor↵
red player created in Windsor↵
TeamRed> create red Magenta↵
red player created in Magenta↵
TeamBlack> move 3 2 south east east↵
red player in position (3, 3)↵
red player in position (4, 3)↵
red player in position (5, 3)↵
TeamMagenta> bunkers↵
1 bunkers:↵
Magenta with 14 coins in position (6, 5)↵
TeamRed> move 6 5 north east north↵
red player in position (6, 4)↵
red player in position (7, 4)↵
red player in position (7, 3)↵
TeamBlack> players↵
1 players:↵
red player in position (5, 3)↵
TeamBlack> create green Windsor↵
green player created in Windsor↵
TeamRed> players↵
1 players:↵
red player in position (7, 3)↵
TeamRed> move 7 3 west west west↵
red player in position (6, 3)↵
Won the fight.↵
red player in position (5, 3)↵
red player in position (4, 3)↵
TeamBlack> move 3 2 east↵
green player in position (4, 2)↵
TeamMagenta> move 4 3 west north north↵
red player in position (3, 3)↵
Bunker seized.↵
red player in position (3, 2)↵
red player in position (3, 1)↵
```

```
TeamBlack> bunkers↵
Without bunkers.↵
TeamBlack> players↵
1 players:↵
green player in position (4, 2)↵
TeamBlack> move 4 2 east↵
green team in position (5, 2)↵
TeamMagenta> bunkers↵
2 bunkers:↵
Magenta with 20 coins in position (6, 5)↵
Windsor with 13 coins in position (3, 2)↵
TeamRed> players↵
1 players:↵
red player in position (3, 1)↵
TeamRed> create blue Windsor↵
blue player created in Windsor↵
TeamBlack> players↵
1 players:↵
green player in position (5, 2)↵
TeamBlack> move 5 2 east↵
Bunker seized.↵
green player in position (6, 2)↵
TeamMagenta> players↵
2 players:↵
red player in position (3, 1)↵
blue player in position (3, 2)↵
TeamMagenta> move 3 1 east east east↵
red player in position (4, 1)↵
red player in position (5, 1)↵
red player in position (6, 1)↵
TeamBlack> players↵
1 players:↵
green player in position (6, 2)↵
TeamBlack> move 6 2 north↵
Won the fight.↵
green player in position (6, 1)↵
TeamMagenta> players↵
1 players:↵
blue player in position (3, 2)↵
TeamMagenta> bunkers↵
2 bunkers:↵
Magenta with 24 coins in position (6, 5)↵
Windsor with 15 coins in position (3, 2)↵
TeamMagenta> move 3 2 north↵
blue player in position (3, 1)↵
TeamBlack> players↵
1 players:↵
green player in position (6, 1)↵
TeamBlack> move 6 1 east↵
green in position (7, 1)↵
TeamMagenta> move 3 1 east↵
blue player player in position (4, 1)↵
TeamBlack> move 7 1 west↵
green player in position (6, 1)↵
```

```
TeamMagenta> move 4 1 east↵
blue player in position (5, 1)↵
TeamBlack> move 6 1 west↵
Player eliminated.↵
TeamMagenta> players↵
1 players:↵
blue player in position (5, 1)↵
TeamMagenta> bunkers↵
2 bunkers:↵
Magenta with 30 coins in position (6, 5)↵
Windsor with 21 coins in position (3, 2)↵
TeamMagenta> move 5 1 west↵
blue player in position (4, 1)↵
TeamBlack> players↵
Without players.↵
TeamBlack> bunkers↵
1 bunkers:↵
My with 27 coins in position (6, 2)↵
TeamBlack> create yellow My↵
Non-existent player type.↵
TeamMagenta> players↵
1 players:↵
blue player in position (4, 1)↵
TeamMagenta> move 4 1 south↵
blue player in position (4, 2)↵
TeamBlack> create blue My↵
blue player created in My↵
TeamMagenta> move 4 2 west↵
blue player in position (3, 2)↵
TeamBlack> bunkers↵
1 bunkers:↵
My with 29 coins in position (6, 2)↵
TeamBlack> players↵
1 players:↵
blue player in position (6, 2)↵
TeamBlack> move 6 2 north↵
blue player in position (6, 1)↵
TeamMagenta> players↵
1 players:↵
blue player in position (3, 2)↵
TeamMagenta> create red Magenta↵
red player created in Magenta↵
TeamBlack> players↵
1 players:↵
blue player in position (6, 1)↵
TeamBlack> move 6 1 south↵
blue player in position (6, 2)↵
TeamMagenta> map↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . .↵
 2 . . P . . . . .↵
 3 . . . . . .↵
 4 . . . . . .↵
 5 . . . . . O . . . .↵
```

```
 6 . . . . . .↵
 7 . . . . . .↵
 8 . . . . . .↵
 9 . . . . . .↵
10 . . . . . .↵
11 . . . . . .↵
TeamMagenta> attack↵
10 11↵
**1 2 3 4 5 6 7 8 9 10↵
 1 . . . . . .↵
 2 . . P . . O . . . .↵
 3 . . . . . .↵
 4 . . . . . .↵
 5 . . . . . O . . . .↵
 6 . . . . . .↵
 7 . . . . . .↵
 8 . . . . . .↵
 9 . . . . . .↵
10 . . . . . .↵
11 . . . . . .↵
The winner is TeamMagenta.↵
> quit↵
Bye.↵
```

5 Project Development Methodology

This section provides some recommendations on a suitable work methodology for a software system development project. It is very important to be methodical in any activity of any duration, particularly in a project of this type. It entails a sequence of activities that must be carried out in a disciplined manner (i.e., a development process), to obtain a quality final "product", reducing the occurrence of defects (bugs) and facilitating their detection when they occur, easing the readability of the program, and... obtaining a good grade.

The program must make the most out of the elements taught in the Object-Oriented Programming course. It should be as extensible as possible to make it easier to add new features (e.g., new player actions), or new types of players.

Start by defining your program's main user interface, clearly identifying which methods it should support, and the respective inputs, outputs and preconditions for each functionality. These interface features are identified considering account the requested commands. Next, it is necessary to identify and specify the entities (interfaces and classes) needed for its implementation. If you identify collections, you should be careful to reuse code rather than repeating it over and over again. Furthermore, you must document your solution using a class diagram, in addition to documenting your code appropriately, with Javadoc.

It's a good idea to build a skeleton of your Main class to handle data input and output. At an early stage, your program won't actually do much, but it is ready for the incremental development of the application. Remember the stable version rule: don't try to do everything at once. Build the program incrementally and test small increments as you implement new features. If necessary, create small test programs to test your classes and interfaces.

To test its functionalities, see the examples illustrated in the statement and the test files. It should start with a very basic program with the commands `help` and `quit`, which are good enough to check if your command interpreter is working well, when there is no game running.

Then, implement the `game` command, and test this feature individually and together with the other features already implemented. Continue with the listing features (`status`, `bunkers`, and `players` commands). Some of these listings are empty (`players`). The next step is to start creating players (command `create`), add them to the game, and view the map of the current team (command `map`). Then implement the command `move`, also in an incremental way, first for simple movements, then for movements with seizing bunkers, fights, and with the end of the game. Finally, implement the `attack` command incrementally from the simplest to the most complex, and so on... Step by step, you will add features to your program incrementally and always with a version with the expected behaviour.

6 Project submission

The program must be submitted to the Mooshak contest POO2324-TP1, by 5:00 pm, April 30, 2024 (Lisbon time).

Each group of 2 students must register in the contest. See the registration rules for Mooshak on the course page. Only programs submitted to Mooshak by users following these rules will be accepted to evaluation. For the project to be evaluated, students must have complied with the DI code of ethics (see the course page), and have, at the end of the deadline, a submission to the Mooshak contest. You can submit your code more than once. Only the last submission will be evaluated.

The project evaluation has 3 components:

1. Correctness of the produced results evaluation: up to 12 points.
   (a) Contest submission having the maximum score (100 points), where the concepts of *interface* and *inheritance* defined in classes are used, will have 12 points in this component.
   (b) A submission with errors in some of the features will have a classification corresponding to correctly implemented features.
   (c) If the submitted program does not use the concepts of *interface*, *inheritance*, and *abstract* classes, the score is reduced to half of that obtained in (a) or (b).
2. Code quality evaluation: up to 8 values.
3. Written discussion (see rules on the course page or Clip).