BATTLESHIPS TOURNAMENT

Table of Contents

INTRODUCTION

In this flow you must write an artificial intelligence (AI) for the classic battleships game, that can beat the other players AI in a tournament.

In short, "Battleships" is a two player game. The game consists of two phases. In the first phase each player secretly positions a number of battleships on a two-dimensional board. Each player has his/her own board. In the next phase each player tries to sink the other player's ships by shooting at specific coordinates on the board.

LEARNING OBJECTIVES

- You must be able to use relevant data structures from Java class libraries
- You must understand what an algorithm is and know that some algorithms are faster than others even though they are solving the same task
- You must be able to construct and implement your own algorithm for the ship placement and for the shooting phase of the game
- You must be able to use existing search and sorting algorithms from Java class libraries, or implement you own, if relevant
- You must be able to work with two-dimensional arrays (the board)
- You must be able to work in a complex code project setting where parts of the functionality is already implemented and can be treated as a 'black-box' (game engine framework)
- You can experience how interfaces are used as design contracts

THE RULES

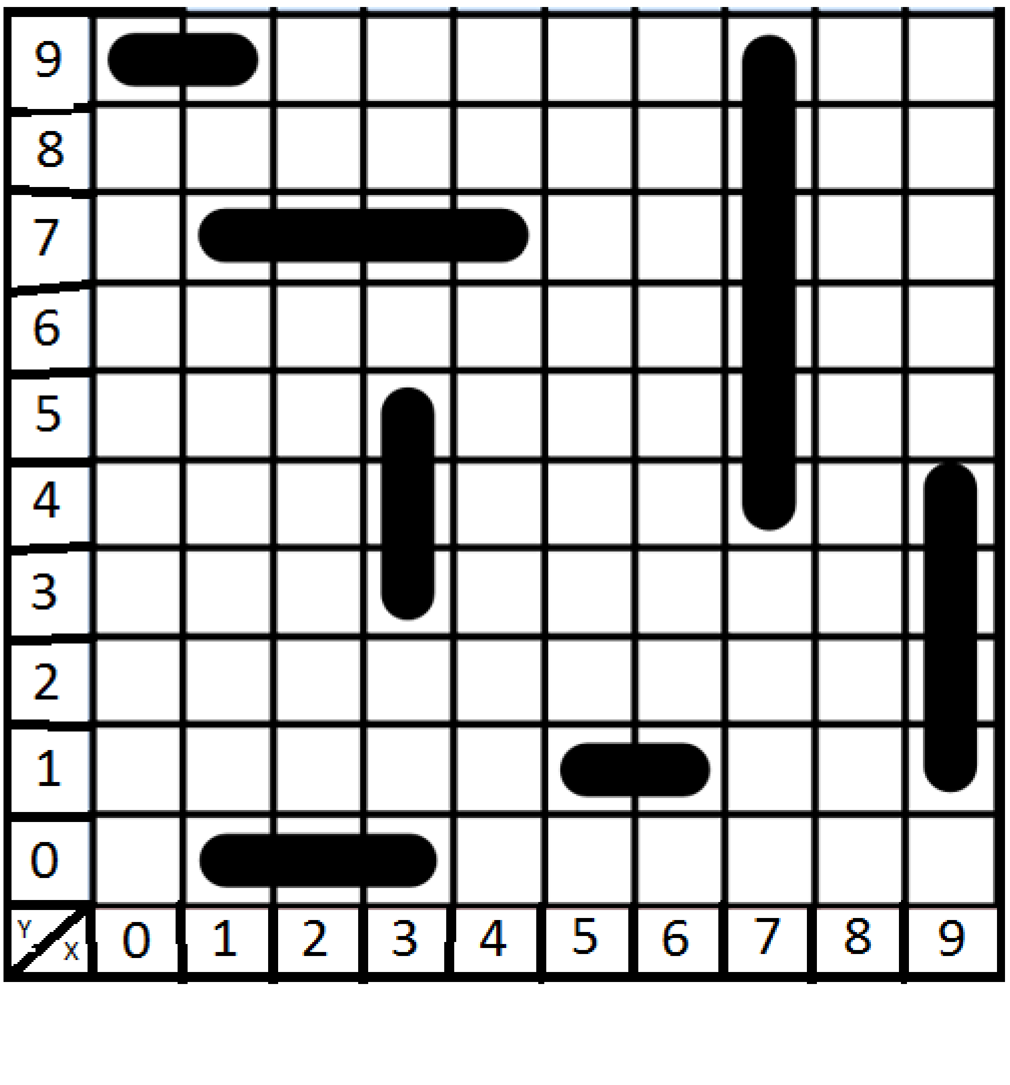
The rules of this Battleships tournament deviates in small ways from the standard battleships rules.

The game consists of two phases:

1. Ship placement phase.
2. Shooting phase.

SHIP PLACEMENT PHASE

In this phase each AI must place his ships on the board. The board is a 2-dimensional grid, where the ships are placed. There are a fixed number of ships and each ship has a fixed length. All ships are one unit wide. A ship can be placed either horizontally or vertically (See: Illustration). In the illustration there are used numbers on the x-axis and letters on the y-axis. In our game we use numbers on both axes. If you place a ship of length *l* horizontally on position (*a*, *b*), it will cover *l* positions: (*a*, *b*), (*a*+1, *b*), ..., (*a*+*l*-1, *b*). If the same ship where placed vertically on the same position, it would cover: (*a*, *b*), (*a*, *b*+1), ... , (*a*, *b*+*l*-1). Example: A ship of length 3 placed vertically on position (2, 4) would cover the positions (2, 4), (2, 5) and (2, 6).



NB: If a ship is placed outside or partly outside the board, it will sink immediately and leave a wreck in all covered positions. If another ship is later placed on a postion with a wreck, it will also sink immediatly and leave a wreck in all covered positions. If a ship is placed on top of another ship, both ships will sink immediatly and leave wrecks in all their covered positions.

SHOOTING PHASE

In the shooting phase the goal is to sink all the enemy ships with as few shots as possible. You will have the same total number of shots as there are positions on the board. This means that unless you fire at the same position multiple times, you will always be able to sink all enemy ships. But your score will be the number of shots you have left, when all the enemy ships are sunk. So be smart and use as few shots as possible :-)

TOURNAMENT

The Battleship tournament is a tournament between Battleships AI's (Artificial Intelligence). It is played as a number of matches, where each AI will meet each of the other AI's in exactly one match. This means that if there are *n* AI's then there will be $(n * (n - 1))/2$ matches. Since battleships involves both a bit of luck and some learning about your opponents strategies, it would not make sense to just play one game in each match. Therefore each match consists of 1000 games and the AI that wins most games in a match, wins the match. In each match the winning AI gets 1 point and the loosing AI gets -1 point. If it is a draw each AI gets 0 points. This means that the total points given is always 0 and if an AI gets a positive score, it is above average and if it gets a negative score it is below average. If an AI gets exactly 0 points, it is average in the tournament.

OVERVIEW OF THE FRAMEWORK

Battleships consists of three predefined projects and your own project (illustrated by E1 below):

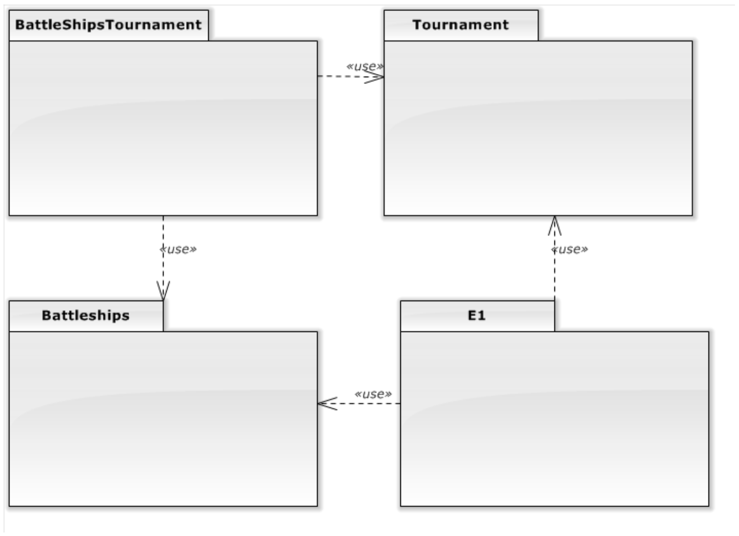


FIGURE 1 PROJECT STRUCTURE EXPRESSED AS UML PACKAGE DIAGRAM

You need to make some initial changes in the projects, for instance solving unreferenced project dependencies. See more in section Getting Started in the end of the document.

In the following, each project is described briefly to give you an overview of the design of the code basis that you find in Battleship.zip on fronter.

THE BATTLESHIPS PROJECT

This project defines the game of Battleships and the rules. This project is responsible for running the games. It also defines the interface *BattleshipsPlayer*, that your AI must implement.

BATTLESHIPSPLAYER INTERFACE

```
public interface BattleshipsPlayer
{
    public void startMatch(int rounds, Fleet ships, int sizeX, int sizeY);
    public void startRound(int round);
    public void placeShips(Fleet fleet, Board board);
    public void incoming(Position pos);
    public Position getFireCoordinates(Fleet enemyShips);
    public void hitFeedBack(boolean hit, Fleet enemyShips);
    public void endRound(int round, int points, int enemyPoints);
    public void endMatch(int won, int lost, int draw);
}
```

- **`public void startMatch(int rounds, Fleet ships, int sizeX, int sizeY)`**
  This method is the first method that is called by the framework. It gives the AI information about the match that is about to start. The parameter *rounds* is the number of rounds in the match (typically 1000), the parameter *ships* is the fleet (number of ships and the sizes of the ships) used in this match. The parameters *sizeX* and *sizeY* defines the size of the board used in this match. Your AI will typically use this information to initialize your own datastructures and classes for information gathering and processing during the match (See below for explanation of the Fleet interface).

- **`public void startRound(int round)`**
  This method is called every time a new round starts in the match. The parameter round is the number of the round that are starting, the first round is number 1. Your AI will typically use this method to initialize temporary datastructures that are used to gather information in a single round.

- **`public void placeShips(Fleet fleet, Board board)`**
  This method is called when your AI should place the ships for this round. The parameter *fleet* represents the ships that should be placed and the parameter *board* represents the board to place the ships on. You **must** use the ships in the fleet parameter to place on the board. If you make your own ship class or use the ships in the fleet from startMatch, the board will silently not accept them (See below for explanation of Fleet and Board interfaces).

- **`public void incoming(Position pos)`**
  This method is called every time the enemy fires a shot. Your AI will typically use this to learn the shooting patterns of the enemy (See below for explanation of the Position class).

- **`public Position getFireCoordinates(Fleet enemyShips)`**
  This method is called every time your AI is supposed to fire a shot at the enemy ships. You must return an instance of the Position class that represents the position your AI wants to fire at. The parameter enemyShips contains the enemy ships that have not yet been sunk before the current shot is fired.

- **`public void hitFeedBack(boolean hit, Fleet enemyShips)`**
  This method is always called immediatly after *getFireCoordinates* and provides feedback on the fired shot. The parameter *hit* is *true* if a ship was hit, *false* otherwise. The parameter *enemyShips* contains the enemy ships that are not yet sunk **after** the shot.

- **`public void endRound(int round, int points, int enemyPoints)`**
  This method is called when a round has ended. The parameter *round* is the number of the round that has just ended. The parameter *points* is your AI's points (shots left) in the round and the parameter *enemyPoints* is the opponent AI's points (shots left) in the round.

- **`public void endMatch(int won, int lost, int draw)`**
  This method is called when the match has ended. The parameter *won* is the number of rounds that your AI won in the match, the parameter *lost* is the number of rounds that your AI lost in the match and the parameter *draw* is the number of rounds that ended in a draw in the match.

FLEET INTERFACE

```
public interface Fleet extends Iterable<Ship>

{

    public int getNumberOfShips();

    public Ship getShip(int index);

}
```

The *Fleet* interface extends *Iterable<Ship>*, so it can be used in for-each loops.

- **`public int getNumberOfShips()`**
  This method returns the number of ships in the fleet.

- **`public Ship getShip(int index)`**
  This method returns the ship on the given index. Indexes are zero based. An index less than zero or grater than or equal to the number of ships in the fleet will throw an exception.

SHIP INTERFACE

```
public interface Ship
{
    public int size();
}
```

The Ship interface represents a single ship in a fleet.

- **`public int size()`**
  This method returns the size of the ship.

BOARD INTERFACE

```
public interface Board
{
    public int sizeX();
    public int sizeY();
    public void placeShip(Position pos, Ship ship, boolean vertical);
}
```

The Board interface represents the board that the ships are placed upon.

- **`public int sizeX()`**
  Returns the width of the board.

- **`public int sizeY()`**
  Returns the height of the board.

- **`public void placeShip(Position pos, Ship ship, boolean vertical)`**
  This method is used to place a ship on the board. The parameter *pos* is the position for the first

unit of the ship. The parameter *vertical* determines the direction of the ship, if *true* the ship will be vertical, meaning the following units of the ship will be placed with increasing y-coordinates, if false, the ship will be horizontal, meaning the following units will be placed with increasing x-coordinates. Be aware that if ships are placed outside the board, partly outside the board or on top of each other, they will immediately sink and create a wreck in their place. Following ships that are placed upon a wreck will also sink and create a wreck themselfes.

POSITION CLASS

```
public class Position implements Comparable<Position>
{
    public final int x;
    public final int y;

    public Position(int x, int y)
    public int hashCode()
    public boolean equals(Object obj)
    public int compareTo(Position o)
    public String toString()
}
```

The class is used to represent positions on the board. This is an immutable class, meaning that when you have created an instance of this class, it can not be altered. The class implements *Comparable*, that sorts positions in y first, then x. It also has reasonable overwrites of *hashCode*, *equals* and *toString*.

THE TOURNAMENT PROJECT

The tournament project is a general tournament executer that can manage tournaments for two-player AI games. It is an all against all match based tournament, where each AI play each other AI in excatly one match. The matches are run in parallel to utilize modern multicore processors. Since matches are run in parallel, each participant may be involved in more than one match at any given time. To make it simple for the AI to keep track of the individual matches, each match is played by a seperate instance of the AI. To facilitate this and give the tournament project the ability to create new instances of the AI's we use the factory pattern and each AI must implement the *PlayerFactory* interface.

PLAYERFACTORY INTERFACE

```
public interface PlayerFactory<PlayerType> extends PlayerInfo
{
    public PlayerType getNewInstance();
}
```

- **`public PlayerType getNewInstance()`**
  This method should return a new instance of the AI. The framework calls this method once for each match that the AI participates in.

PLAYERINFO INTERFACE

This interface provides information about the player (AI). The *PlayerFactory* interface extends the *PlayerInfo* interface, so when you implement the *PlayerFactory* interface, you also need to implement the methods from *PlayerInfo*.

```
public interface PlayerInfo
{
    public String getID();
    public String getName();
    public String[] getAuthors();
}
```

- **`public String getID()`**
  This method should return the ID of the AI. The ID is a short unique identifier for the AI. In the battleships tournament, the ID should be the group ID of the group that created the AI. The group ID consist of an uppercase G, Y or R followed by a number. Example IDs: G42, Y23, R12. The letter refers to the league of the group, there are 3 leagues: Green, Yellow and Red.

- **`public String getName()`**
  This method should return the name of the AI. This should not be too long.

- **`public String[] getAuthors()`**
  This method should return the names of the authors of the AI.

BATTLESHIPSTOURNAMENT PROJECT

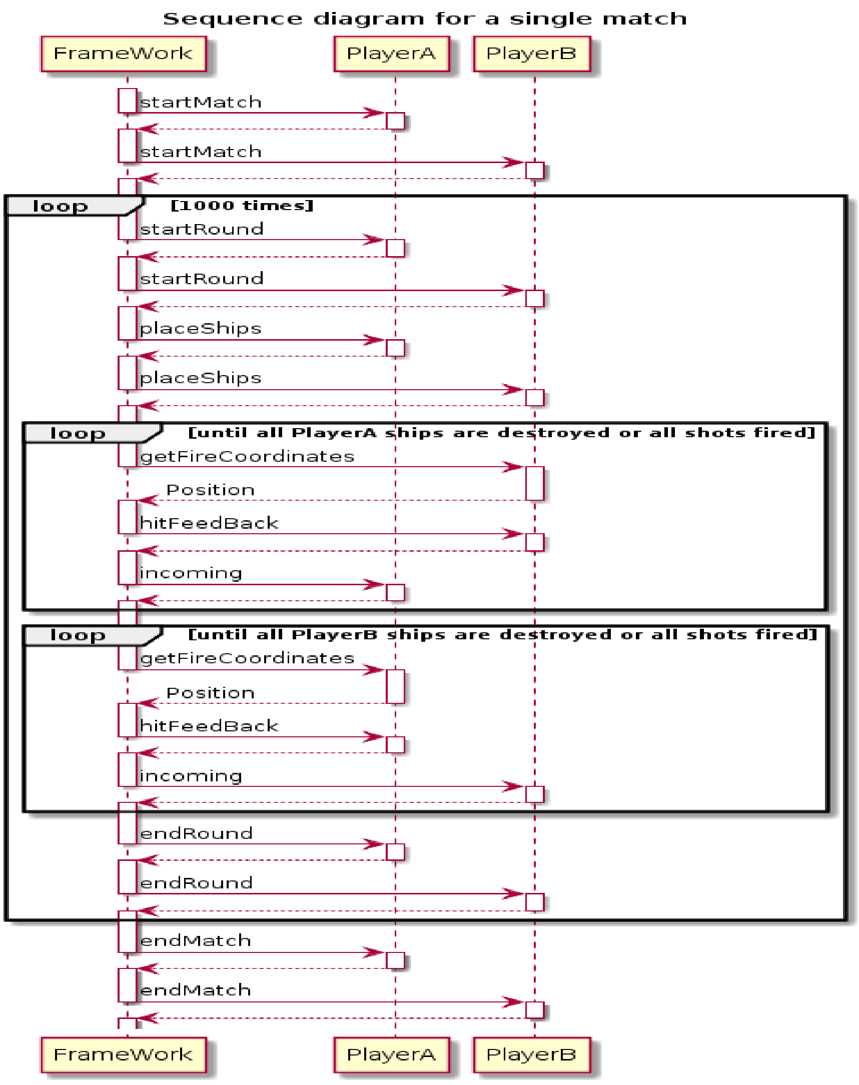
This is the project that ties the ends together. The responsibility of this project is to load the AI's from jar-files and run the tournament. This project is dependent on both the Tournament project and the Battleships project. We load the individual AI's from jar-files instead of having this project be dependent on each of the individual AI projects. This saves a lot of time when setting up the tournament. Since the AI's are automatically loaded from the jar-files, they need to obey some naming rules to be able to enter the tournament. These naming rules are described in the Handin Section of this document.

EXAMPLE AI'S

There are two example AI projects, they are named E1 and E2. You are given the source code to these two. These two examples are very bad players, that play mostly random. You are also given a number of AI's that you can test your own AI against. They are named X1, X2, ... etc. You will only get the jar-files for these, not the source code.

THE FLOW OF A MATCH

Below is the sequence diagram for a single match between player A and player B:



HAND-IN

NOTE: It is very important that you follow these rules when handing in your Battleships AI

You must make a group hand-in on moodle where every member of your team is added to the hand-in.

You must hand in a single jar-file and nothing else.

The following naming conventions **must** be obeyed:

1. The jar file **must** be named after your group ID with an **uppercase** letter. Examples: **G3.jar**, **Y12.jar** and **R42.jar**.

2. The main package where your class that implements *PlayerFactory* is placed **must** be named after your group ID with a **lowercase** letter. Examples: **g3**, **y12**, **r42**. You **may** have subpackages inside the main package with helper classes, but to avoid name clashes with other groups, you should **not** put **any** code outside the main package.

3. Your class implementing *PlayerFactory* **must** be named after your group ID with an **uppercase** letter. Examples: **G3**, **Y12**, **R42**. The fully classified name for your factory class should look like these examples: **g3.G3**, **y12.Y12**, **r42.R42**.

Make sure to **test** your jar-file before you hand it in. Use the Battleships Tournament project to verify that your jar-file can load and your AI can win against at least E1 and E2 before you hand in.