Chapter 1

Example game: Battleship

The game is a version of the classic battleship puzzle game. It is played by two players who both try to sink their opponents 'ships' while avoiding their owns getting sunk. The game has two phases – the setup of the battlefield and the shooting. Bot players have 7 ships, which have distinct lengths and are one unit in width.

Ship name	**Size**	**Count**
Battleship	4	1
Cruiser	3	2
Destroyer	2	3
Submarine	1	4

Ships of a player

During the setup phase, the players place all their ships on their 10×10 grid battle battlefield vertically or horizontally. The only limitation is that ships may not border each other over edges (they can touch diagonally). After this setup is complete, the players may not reposition their ships for the remainder of the game and their layout is kept secret.

Then the shootout begins. Taking turns, the players select targets (by their coordinates, such as 'A–5') and their opponent report them the result of the attack as either 'miss', 'hit' or 'sink'. Miss means no ship has been placed at that position, hit and sink reflect that a ship in fact occupied that place, sink also means that all of the ship's part have been hit. The player

than marks the selected position on his own target field (as hit or miss), and if the result was not a miss, the other player marks the targeted part of the ship hit. The current player keeps shooting at new targets until he misses or all the enemy ships have sunk. After that – if not all ships are gone – a new turn starts and the opponent start shooting. The first player to destroy all of their adversary's vessels wins the game.

1.1 Design

Designing often involves taking a gameplay planned for real–world usage and created with physical limitations in mind and modifying it to naturally fit into a virtual environment. This, of course, is done any time a digital version of a game is being made, however in this case a notable design consideration is the fact that the players (bots) and the core game (engine) must be separated from each other.

I have decided to follow the natural separation of game phases – setup and shootout.

The setup could be done through a number of ways, for example per–ship placement requests (engine asks the bot where each ship should go), distinct placements of ships (engine asks the bot to place one of the remaining ships) or a one–time setup of all ships (engine asks for a complete setup). I have opted to use the latter, as I feel it matches the layout planning the most. Since the ships have placement constraints between them, it is probable that a bot places all of them at once to ensure the validity of the design.

The shootout part is basically the same operation done over and over on an ever–changing target. It, again, could be designed in many ways, e.g. with a single or result–dependent result callbacks or with the previous result sent at each new target request. I have elected to go with passing the results at every new request to simplify the bot interface. This means that at the first shooting the previous status is a `null` value which is not ideal due to java's lack of proper distinction between nullable and non–nullable typed, but with good bot interface documentation it is not a problem.

1.2 Game API

The api of the game is fairly simple, mostly consists of model classes that help describe the game model in a type–safe manner. For a more complex game the creation of more utilities would make sense as to make bot development easier.

1.2.1 Domain classes

These include any models that are used in the engine–bot communication or may be useful for bot development.

`Position` A simple $(x; y)$ coordinate pair, also lets one check whether it is a valid position on the game field. *(data class)*

`Direction` The orientation of a ship, may be vertical or horizontal. *(enumeration)*

`Ship` Describes the properties of a ship, including its (start) position, direction and size. *(data class)*

`ShootResult` the result of an shot, may be a miss, a hit or a sink. *(enumeration)*

`TargetMap` the view of the opponent's field. Shows all the tried positions as shoot results. When a ship is sunk, its bordering water is marked missed, and the ship itself as sunk. Each position can be queried individually and it also contains a helper method to list all non–tried targets.

1.2.2 Bot interface

The bot interface (`BotInterface`) is based on the design of the communication between the engine and the bots, therefore it has two abstract methods: `setupField` and `nextShoot`.

`setupField` takes the sizes of the ships to be placed as its parameter and must return a valid list of ships. It is called at the beginning of the game once.

`nextShoot` is called on each turn and is used to determine the target of the next shot. Its arguments are the current value of the bot's target map, the previous target of the bot, and the result of the previous shot. The latter two are `null` on the first turn of the bot. The method must return a valid, not yet tried target to shoot at.

BattleshipBot.java

```
public interface BattleshipBot extends BotInterface {

    List<Ship> setupField(final List<Integer> shipSizes);

    Position nextShoot(
        final TargetMap targetMap,
        final Position previousTarget,
        final ShootResult previousResult
    );
}
```

1.2.3 Utility

As this game is fairly simple, I have create a simple utility class for shared usage – `ShipUtil`. This helper has a simple public method, `randomSetup`, which places all the ships at random – but valid – positions at random orientations. I have found this rather practical, as the placement of the ships – if not following some elaborate scheme – is often done randomly by the players and is a non–trivial task (due to the rules of ship placement).

1.3 Engine

The engine module contains the actual game engine class, as well as other resources that class uses, such as `ShipMap` and `ShipState` which mutably hold information of the state of a player's field.

The game engine class' structure is quite typical. For per–bot data keeping it defines a static inner class to hold the field of the bot with its last target and the result of that shot (if any). For each bot an instance of this gets created on the duel game engine's bot initialization phase, based on given bot's placement of its ships.

After that, the turn–based shooting starts. A turn here – by the duel game engine's definition – represents a set of operations that take place while one of the bots is working. This may include multiple shots, until a it finally misses or wins the game by sinking all enemy ships. After each shot attempt the engine first validates the target by checking it is a not yet tried position on the board. Based on the validation result the engine either declares the game lost due to an error, or actually processes the changes which results in a shot result. The bot's previous shot data is always updated to the current target and result, then based on the shot result some action may follow:

- If the shot was a miss, the bot's turn is over.
- If the shot was a hit, the turn continues with a next shot.
- If the shot sank a ship, the engine checks if all ships are sunk. If that is the case, the game is won by the current bot, otherwise the turn continues as with simple hits.

1.4 Random bot

I have found it often useful when implementing a game to first create a simple bot that plays the game 'dumbly', but in accordance with the rules. This helps in both testing the engine and can be used to duel against a more competent opponent later on.

It the case of battleship, I have created a bot that places its ships randomly, then at the shootout phase chooses targets also randomly – of the not yet tried tiles –, regardless of the attacks' result.

This implementation course performs very poorly ($\mathbb{E}(x) \approx 71.19$ turns to finish), but can still give a good idea of what functionality to build into the api, in this case the random ship placement was a result of this experiment.

RandomBot.java

```
@Override
public List<Ship> setupField(List<Integer> shipSizes) {
    return ShipUtil.randomSetup(shipSizes);
}

@Override
public Position nextShoot(
        final TargetMap targetMap,
        final Position previousTarget,
        final ShootResult previousResult) {

    // all valid targets
    final List<Position> targets =
            targetMap.freePositions().collect(toList());

    // choosing randomly
    return targets.get(random.nextInt(targets.size()));
}
```

1.5 A smarter bot

I have also created a better performing bot which can play the game almost as well as a human can. (`SmartBot`) Just like the previous bot, it places its ships randomly, and also shoots at random if it hasn't discovered any ships yet. However, when it hits something it starts attacking that target consciously. This starts by shooting around the hit mark to find the orientation of the ship under attack, and when that becomes known shooting at the end of a – randomly chosen – longitudinal side. It of course can take multiple turns to sink the ship, but will it will get done sooner or later.

It is not an optimal player, because it does not select new targets intelligently (filtering based on where ships can fit) when a ship is sunk, but is fairly performant due to the gameplay being inherently heavily based on sheer luck.

This smarter bot is naturally vastly superior to the previous random bot – I am yet to see it be defeated by that player.

$$X \coloneqq \text{number of turns}$$

$$S \coloneqq \text{number of shots}$$

$$P_X(n) \coloneqq \text{Solve puzzle in exactly n turns}$$

Since all 20 hits except for the very last are followed by a new shot at the same turn, the number of turns can be calculated from the number of hits as $X = S + 19$ and the probability of each turn value as $P_X(n) = P_S(n + 19)$.

$$P_L(n) \coloneqq \text{probability of the last target being after n shots, } 20 \leq n \leq 100$$

$$P_L(n) = \frac{1}{100 - (n-1)}$$

$$P_L(n) = \frac{1}{101 - n} \tag{1.1}$$

$$P_F(n) \coloneqq \text{probability of 1 target remains after n shots, } 20 \leq n \leq 100$$

$$P_F(n) = \frac{\binom{20}{19} * \binom{80}{n-20}}{\binom{100}{n-1}}$$

$$P_F(n) = \frac{20 * \binom{80}{n-20}}{\binom{100}{n-1}} \tag{1.2}$$

$$P_S(n) = P_F(n) * P_L(n)$$

$$P_S(n) = \frac{20 * \binom{80}{n-20}}{(101 - n) * \binom{100}{n-1}} \tag{1.3}$$

$$\mathbb{E}(X) = \sum_{i=1}^{100} \Big(i * P_X(i) \Big)$$

$$\mathbb{E}(X) = \sum_{i=20}^{80} \Big(i * P_S(i) \Big)$$

$$\mathbb{E}(X) \approx 71.19 \tag{1.4}$$