A Guide to Battlecode

Ivan Geffner

1 Introduction

The purpose of this document is helping and engaging new Battlecode competitors. Here, we'll go over the Battlecode competition in general and some tips and tricks I personally found useful to perform better. We will be assuming throughout the document that Battlecode uses the Java bytecode engine, although some of the discussion might still be relevant on other engines as well.

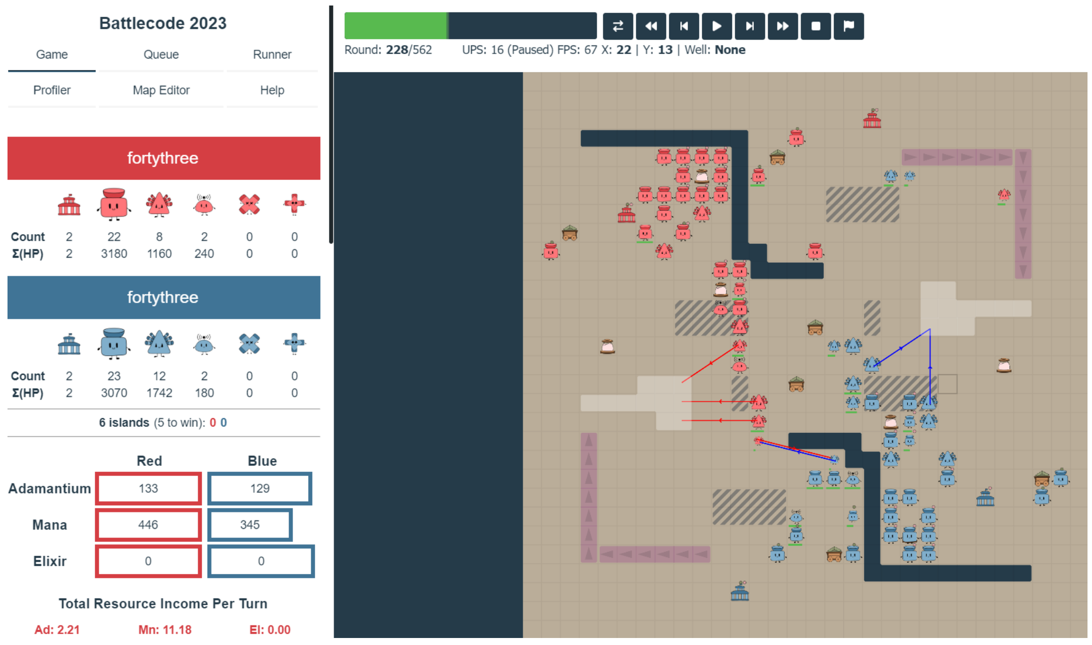
1.1 About me

I'm a Battlecode participant that has been competing since 2015 under the username **XSquare** and has spent an unhealthy amount of time on the competition. I also run AI Coliseum since 2018, which is a very similar competition based in Barcelona. On whatever free time I have left from Battlecode and AI Coliseum, I work as a Math postdoc at the Game Theory group in the Technion. You can find more about it at my site.

Figure 1: A typical day in my life.

1.2 About Battlecode

Battlecode is an AI Competition organized by the MIT in which each team must code an AI for a given real-time strategy game, which is unknown until the beginning of the competition, and try to beat the AIs submitted by other teams. Usually, these games involve controlling a bunch of different robots, managing resources, and interacting with the opponent's robots.



fortythree

Count	2	22	8	2	0	0
Σ(HP)	2	3180	1160	240	0	0

fortythree

Count	2	23	12	2	0	0
Σ(HP)	2	3070	1742	180	0	0

6 islands (5 to win): 0 0

	Red	Blue
Adamantium	133	129
Mana	446	345
Elixir	0	0

Total Resource Income Per Turn

Ad: 2.21 Mn: 11.18 El: 0.00

Round: 228/562 UPS: 16 (Paused) FPS: 67 X: 22 | Y: 13 | Well: None

Figure 2: Example of a Battlecode 2023 game

I personally think this is an amazing competition that gives a challenge both for beginners and experimented coders. Its amazing community only makes it better! :)

2 Battlecode Mechanics Overview

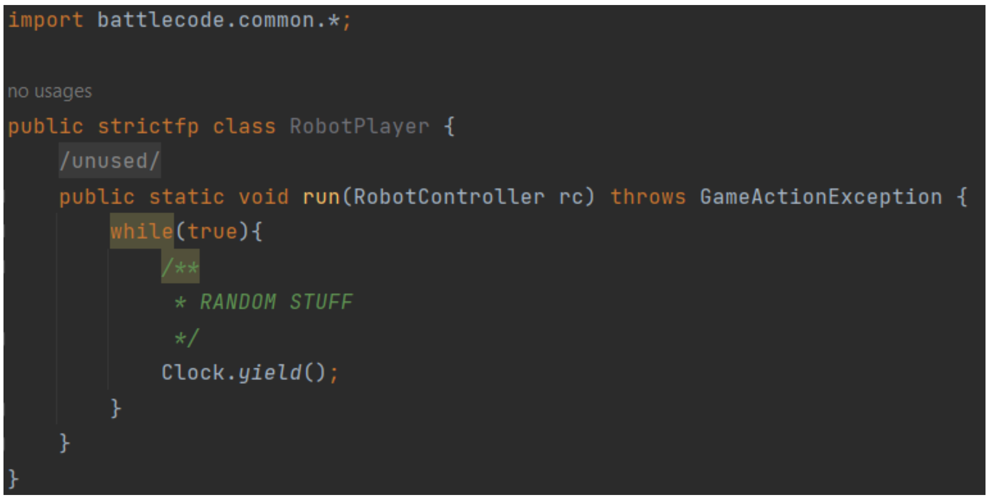
In each Battlecode edition, each team usually starts the game with several (virtual) robots placed throughout the map. The robots then take turns sequentially and, on each of these turns, each robot can perform actions such as moving, attacking, constructing other robots, etc. Usually, the amount of actions of a certain type that a robot can perform in a single turn is limited, either because they have cooldowns or because they cost in-game resources. Besides explicit actions, each robot can also use its turn to sense nearby objects, communicate with other robots, or performing internal computations (e.g., computing the optimal course of action). To prevent robots from using too much time for their computations, the engine keeps track of how many basic Java instructions (more precisely, Bytecode instructions) the robot executes during its turn. If it exceeds a certain limit, its execution is halted and resumes during its next turn (see more details in Section 6).

In most recent editions, robots run independently, and have no shared vision or shared memory. This means that each robot can only sense other robots or objects that are close enough to its location. This setting makes it difficult for robots of the same team to coordinate (for instance, if several robots of the same team must decide to either attack or retreat, each robot may reach a different conclusion depending on what they see). However, there is always some device that allows robots to communicate in some way.

Our aim is to code an AI that tells each robot what to do in each turn.

3 First Steps

After following Battlecode's installation guide, the first thing we see is a code that looks like this:



```
import battlecode.common.*;

no usages
public strictfp class RobotPlayer {
    /unused/
    public static void run(RobotController rc) throws GameActionException {
        while(true){
            /**
             * RANDOM STUFF
             */
            Clock.yield();
        }
    }
}
```

Figure 3: Battlecode's example code

More precisely, they provide us with an example code consisting of an `examplefuncs` package with a `RobotPlayer` class that looks like Figure 3. The instructions given inside the `while(true)` loop may vary from game to game. This will be the basic framework we will use to code our AI. However, before we start, it is important to know at a high level how the engine works.

3.1 The Battlecode Engine

Whenever a new game starts, the engine initializes a list with all the initial units in it. Each round, the engine iterates over the list and runs each robot's instance of instance of `RobotPlayer.run()`. Whenever a new robot r is created, r is added to the list. Each robot r's `RobotPlayer.run()` instance is run until one of the following happens:

(a) Robot r calls the `Clock.yield()` method. When this occurs, the engine halts r's instance of `RobotPlayer.run()` and proceeds to run the next robots' instances. After all other robots had their turn, u continues the execution of `RobotPlayer.run()` right after the `Clock.yield()` instruction.

(b) Robot r is killed. If this happens, its instance of `RobotPlayer.run()` is halted and is never executed again (if r is killed while `RobotPlayer.run()` is halted, it never resumes).

(c) Robot r's instance of `RobotPlayer.run()` returns. Whenever this happens, r is killed and the engine proceeds as in (b).

(d) Robot r's instance of `RobotPlayer.run()` throws an unhandled exception. In this case, r is also killed and the engine proceeds as in (b).

(e) Robot r's turn "takes too long" (i.e., it exceeds the Bytecode limit). Whenever r exceeds a certain threshold, r's turn is halted and the engine proceeds as in (a): all other robots take their turn and then r can run again starting at the point where it left off. Intuitively, it is as if r is forced to automatically call `Clock.yield()` as soon as it hits the threshold.

As we can guess from looking at the halting conditions, it is always preferable to end r's turn manually with a `Clock.yield()` call. This way, we can guarantee to perform all the necessary actions (moving, attacking, etc.) and finish the turn when there is nothing left to do. If a turn ends because of (e), it might induce unexpected behavior (for instance, a robot may end its turn without attacking). Because of this, it is usually suggested to wrap the instructions of each turn inside a `while(true)` loop, and end the loop with a `Clock.yield()` method as in Figure 3.

3.2 The `RobotController` class

Now that we know roughly what to do, how do we tell our robots where to move, who to attack, or what to do? Note that, inside the `RobotPlayer` class, the `run` method has a `RobotController` instance as an input. As the name suggests, this object will allow us to control the robot associated with this RobotPlayer's instance. In particular, if we check the Battlecode documentation, we'll see that `RobotController` has a bunch of methods like `move([...])` or `attack([...])`, as well as other methods like `getLocation()` or `senseNearbyRobots()` that allow us to get information about the current state of the game. The combination of such methods allow us to perform instructions such as "Attack the closest enemy robot", or "Move towards the closest allied robot". Of course, if we want to implement more complex strategies, these instructions will become longer and more convoluted.

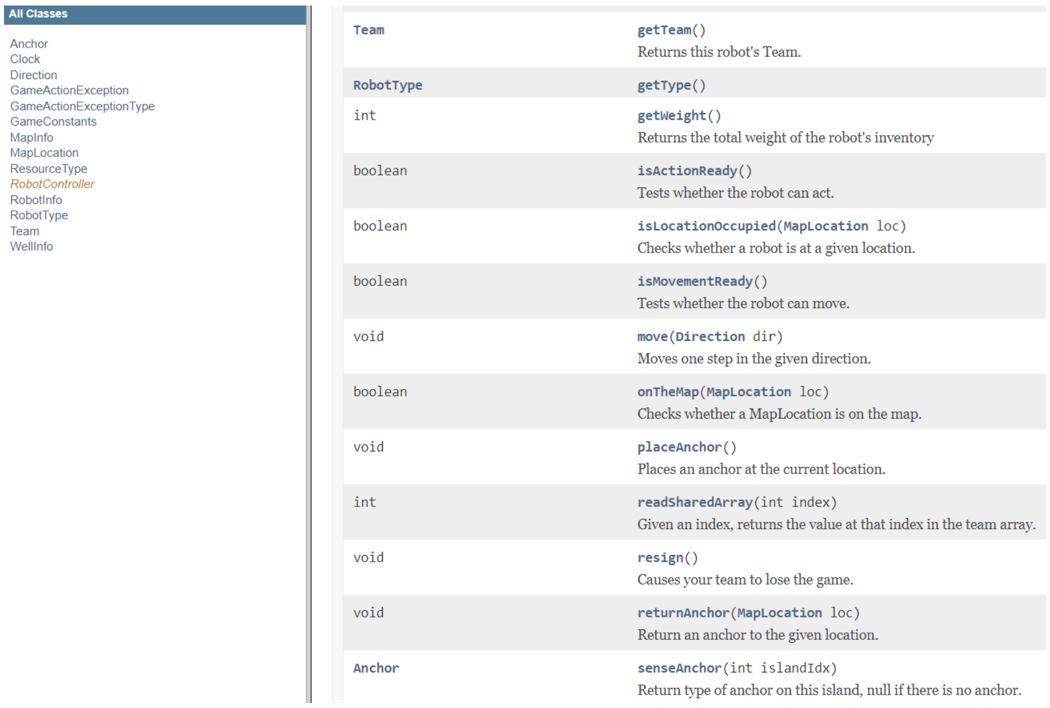
3.3 Other classes

If we check Battlecode's documentation, we'll see that there are other custom classes besides `RobotController` that will aid us in our implementation. Most of them are required as an input in some of `RobotController`'s methods, and some are obtained as outputs of such methods. Some of these classes appear in almost every single Battlecode game, such as for instance

- `Direction`: Enum with all eight main directions (`North, NorthEast, East`, etc.) in addition to `Zero`.
- `MapLocation`: Class that contains the coordinates of a given location in the map.
- `Team`: Enum with the possible factions in the game (usually there's only two - `A` and `B` - but sometimes there were neutral factions like `Zombie` or `Neutral`).
- `RobotInfo`: Class that contains the information (health, type, location, etc.) of a given robot.
- `RobotType`: Enum that contains all possible robot types.
- `Clock`: Class that contains the `yield()` method and allows us to get the amount of bytecode instructions[1] run up to that point.
- `GameConstants`: A class with all the game parameters.

Besides these, there are other classes that are more game-specific. For instance, in **Battlecode 2023** you could get resources from some *wells* that were scattered throughout the map. Therefore, in this game there was also a class `WellInfo` that contained all the necessary information about a given well (type, location, etc.).

[1]Check Section 6 for more details.

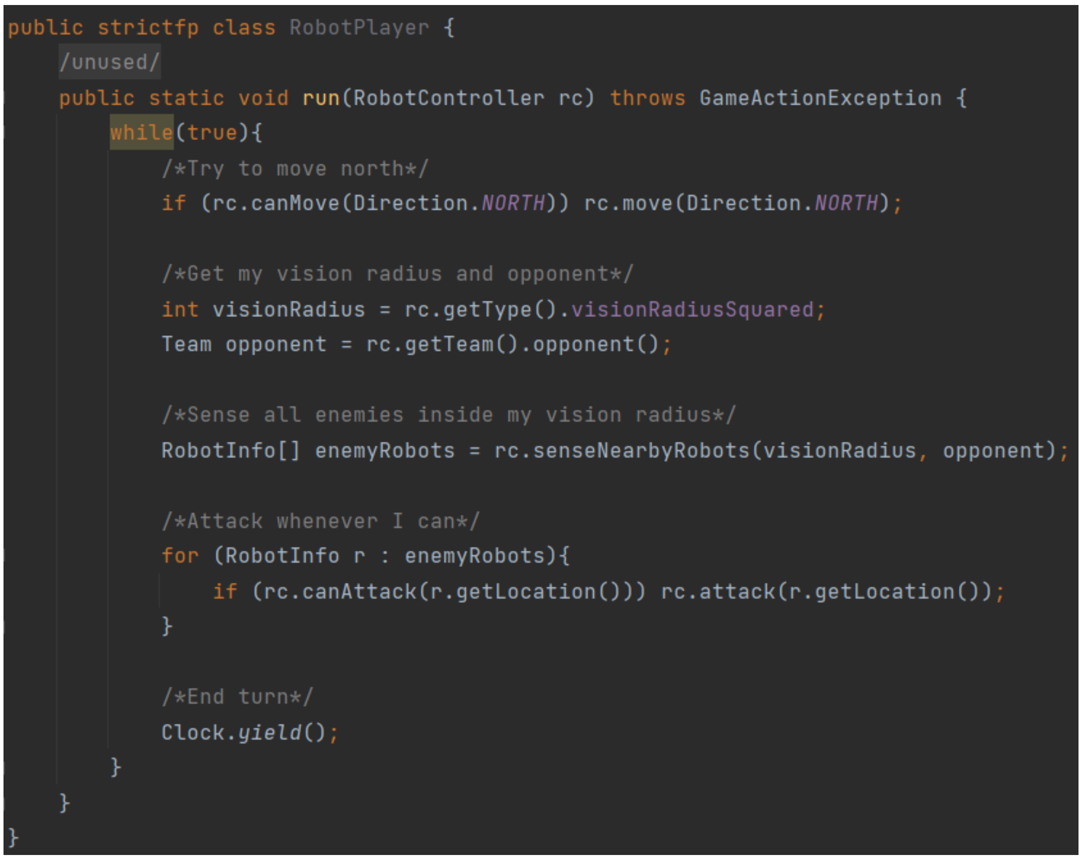

All Classes

Anchor
Clock
Direction
GameActionException
GameActionExceptionType
GameConstants
MapInfo
MapLocation
ResourceType
RobotController
RobotInfo
RobotType
Team
WellInfo

Team	`getTeam()` Returns this robot's Team.
RobotType	`getType()`
int	`getWeight()` Returns the total weight of the robot's inventory
boolean	`isActionReady()` Tests whether the robot can act.
boolean	`isLocationOccupied(MapLocation loc)` Checks whether a robot is at a given location.
boolean	`isMovementReady()` Tests whether the robot can move.
void	`move(Direction dir)` Moves one step in the given direction.
boolean	`onTheMap(MapLocation loc)` Checks whether a MapLocation is on the map.
void	`placeAnchor()` Places an anchor at the current location.
int	`readSharedArray(int index)` Given an index, returns the value at that index in the team array.
void	`resign()` Causes your team to lose the game.
void	`returnAnchor(MapLocation loc)` Return an anchor to the given location.
Anchor	`senseAnchor(int islandIdx)` Return type of anchor on this island, null if there is no anchor.

Figure 4: Battlecode 2023 documentation sample

Now that we know the basics we can start coding! For example, the code in Figure 5 instructs a unit to try to move north and to attack all enemies it can every round.



```java
public strictfp class RobotPlayer {
    /unused/
    public static void run(RobotController rc) throws GameActionException {
        while(true){
            /*Try to move north*/
            if (rc.canMove(Direction.NORTH)) rc.move(Direction.NORTH);

            /*Get my vision radius and opponent*/
            int visionRadius = rc.getType().visionRadiusSquared;
            Team opponent = rc.getTeam().opponent();

            /*Sense all enemies inside my vision radius*/
            RobotInfo[] enemyRobots = rc.senseNearbyRobots(visionRadius, opponent);

            /*Attack whenever I can*/
            for (RobotInfo r : enemyRobots){
                if (rc.canAttack(r.getLocation())) rc.attack(r.getLocation());
            }

            /*End turn*/
            Clock.yield();
        }
    }
}
```

Figure 5: Sample code from Battlecode 2023

4 Code structure

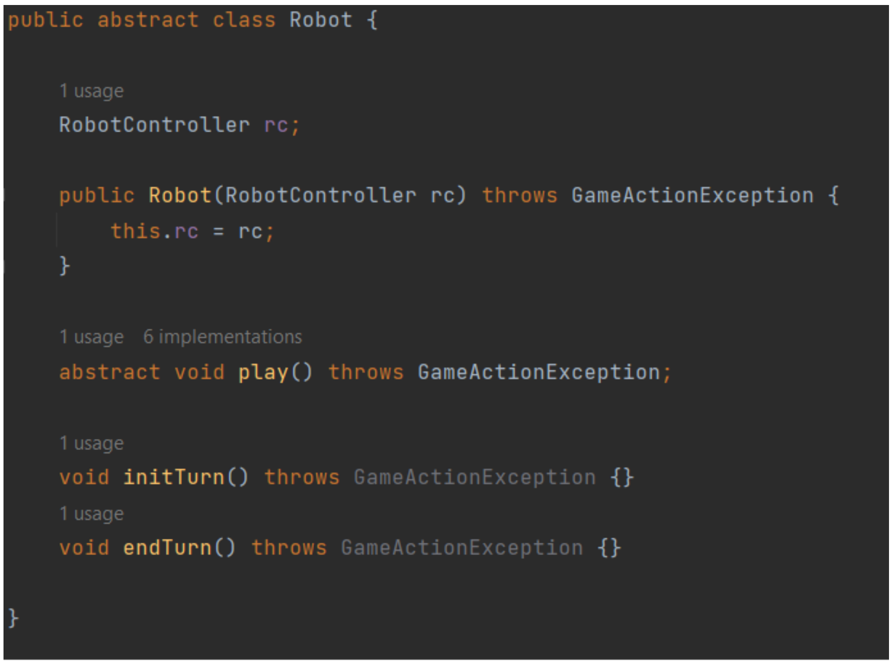
As we said just before, we can start coding... However, this doesn't mean we should! Usually, top performing bots end up consisting of several thousand lines of manually written code. This means that, instead of clicking on our keyboard right away, it is convenient to think about the high level strategy that we want to implement and structure the code in advance. There is only **one exception** to this rule! If it is your first time participating in Battlecode, I would personally suggest to code a very basic bot to get used to the documentation and how everything works. However, once you get the *feel* of Battlecode, I'd start from scratch with a well-thought structure.

Part of the code structure is game-dependent. For instance, depending on the game we might want to create custom objects to keep track of different elements of the map, or objects to handle several functionalities shared between units such as *Pathfinding, Communication,* etc. To see examples from previous editions, there are a lot of sample codes from past participants in the #open-source chat in the Discord.

That being said, there are always common elements between different Battlecode editions: the RTS component, different types of robots, the Java engine, etc. This allows us to have a generic code structure and build everything else on top of that. For instance, I usually want to code each type of unit separately. Thus, I usually create an abstract `Robot` class that is later extended by all classes associated to individual unit types. The `Robot` class has its `RobotController` as attribute, and contains three main methods:

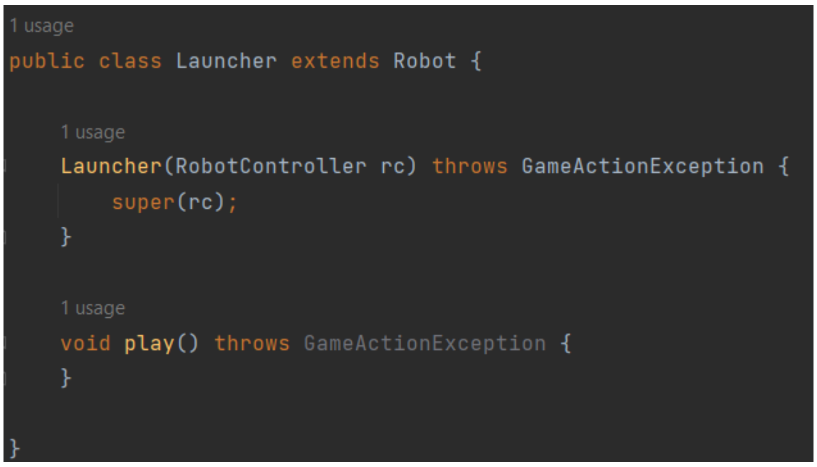
- `abstract void play()`: This method runs exactly one round of the robot and will be implemented by each unit type separately.
- `void initTurn()`: Here we include all procedures that we want all robots to run at the beginning of their turn.
- `void endTurn()`: Here we include all procedures that we want all robots to run at the end of their turns. Usually this includes expensive computations that we want them to perform during several turns with their spare bytecode.

This way, we can instantiate the correct extension of `Robot` at the beginning of the `run()` method, and then delegate everything that is particular of that unit type to the extension. An empty example of this overall structure can be found in Figures 6 7 and 8.



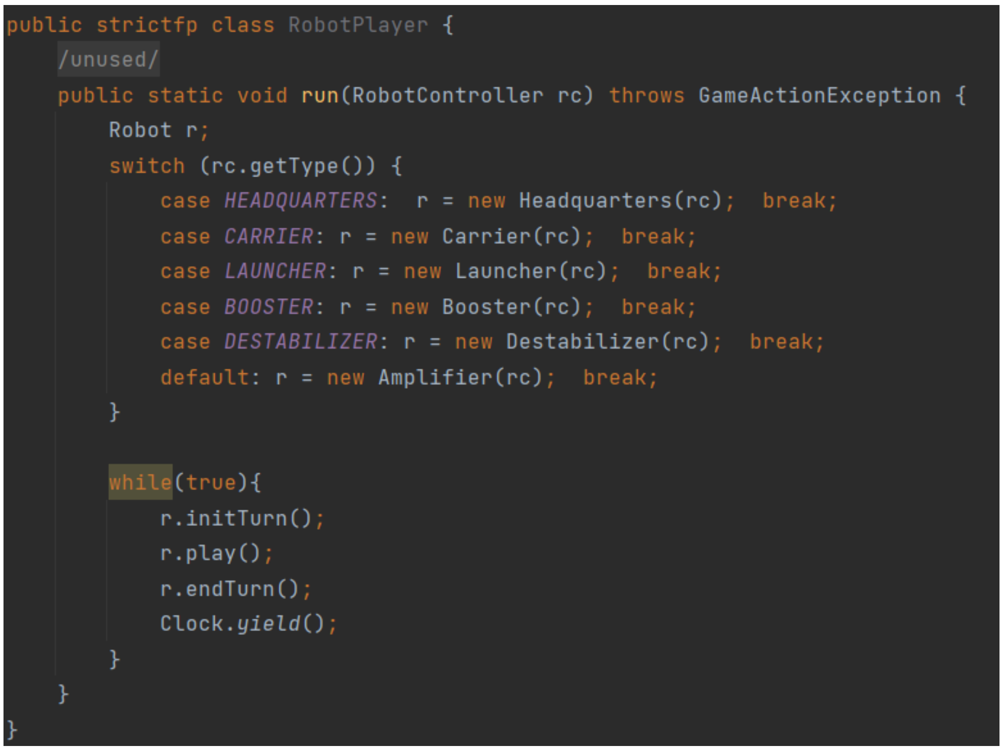
```java
public abstract class Robot {

    1 usage
    RobotController rc;

    public Robot(RobotController rc) throws GameActionException {
        this.rc = rc;
    }

    1 usage   6 implementations
    abstract void play() throws GameActionException;

    1 usage
    void initTurn() throws GameActionException {}
    1 usage
    void endTurn() throws GameActionException {}

}
```

Figure 6: Example of an empty `Robot` class



```
1 usage
public class Launcher extends Robot {

    1 usage
    Launcher(RobotController rc) throws GameActionException {
        super(rc);
    }

    1 usage
    void play() throws GameActionException {
    }

}
```

Figure 7: Example of an empty Launcher class (a Robot Type from Battlecode 2023)



```
public strictfp class RobotPlayer {
    /unused/
    public static void run(RobotController rc) throws GameActionException {
        Robot r;
        switch (rc.getType()) {
            case HEADQUARTERS:  r = new Headquarters(rc);  break;
            case CARRIER: r = new Carrier(rc);  break;
            case LAUNCHER: r = new Launcher(rc);  break;
            case BOOSTER: r = new Booster(rc);  break;
            case DESTABILIZER: r = new Destabilizer(rc);  break;
            default: r = new Amplifier(rc);  break;
        }

        while(true){
            r.initTurn();
            r.play();
            r.endTurn();
            Clock.yield();
        }
    }
}
```

Figure 8: Example of a RobotPlayer Class

I personally like to use this structure and have been using it for the past Battlecode editions. I also like to keep an instance of each object with generic utilities (pathfinding, communication, a class with all core methods, etc.), inside the `Robot` class, in such a way that they are all accessible later on. If you expect several types of robots to behave in the same way (e.g., two different type of attacking units with different stats), sometimes it is good to have them both inherit from a smaller extension. For instance, in Battlecode 2023 I had an `Attacker` class, and `Launcher, Amplifier, Booster` and `Destabilizer` inherited from it. For more concrete examples, I include here my `Robot` class from 2021, 2022 and 2023 (2023 is pretty messy).

5 Battlecode Dos and Don'ts

5.1 Do: Modular code and robust primitives

In my experience, taking your time to build robust primitives (pathfinding, communication, micro) and modular code pays off in the end. It is really convenient to have, for instance, a `Pathfinding` black box that you can use to move your unit closer to your target[2], or a `Communication` black box that you can use to query and report information about the current state of the game. Being lazy on this subject will make quite difficult to upgrade the bot during the final stages of the competition (and those days are the most important!)

5.2 Don't: Be afraid of tackling Battlecode alone or with a small team

Every year, there are quite a few solo teams, and a lot of them perform extremely well. I personally find that the **ideal team composition** is one person coding and $N - 1$ people watching replays, giving insight, testing, and keeping track of the TO-DOs . In fact, with this setting, only one person must know how to code, although it helps if the other team members are familiar with how coding works. It is important to make clear that not everyone shares my opinion on this. However, I have participated in several editions with teams where everyone codes, solo, and with teammates that didn't code, and - as a person who likes to code - I was the most comfortable in the latter setting (and this doesn't have anything to do with how *good* the other coders are. They were all brilliant and some of them were IOI and ICPC finalists). In case there is more than one coder on the team, following the instructions in the previous section becomes more important.

5.3 Do: Copy, a lot

All top teams in Battlecode copy from one another and, if they tell you they're not, they're lying! As the competition advances, the meta changes, and the strategies that prove to be better start spreading among all the top ranked bots. Don't be afraid to test them out and find ways to improve on them!

[2]Note that, because of bytecode constraints, pathfinding is usually not trivial.

5.4 Don't: Overfit for current maps

Sometimes, we spend way more time than we should making sure a bot wins on a very specific map. This is especially true at the beginning of the competition, in which your rating depends on your results against other bots in the same three very generic maps. Overfitting for these maps may prove fatal since **Teh Devs** release a complete new set of maps for each tournament. It may sometimes be useful to build your own maps.

5.5 Do: Test against old versions

The best way to know if your bot is improving is testing against old versions of your bot on the largest possible map set. Do not forget to test on both sides (red and blue) since your bot may have a larger winrate in one side[3]!

5.6 Don't: Get discouraged when losing

If your bot consistently loses against higher-ranked bots, it means that they are doing something better. It is your job to find what are they doing and how! Analyzing games against them is the perfect opportunity to find new features to improve your bot.

5.7 Do: Take your time to analyze replays and think about your strategy

Sometimes it feels like, if we are not coding, then we are pretty much wasting our time. However, I personally have found that some of the days in which I make the most improvement are those in which I take some time off, watch replays, and spend some time thinking out of the box about the stuff that can be improved.

5.8 Don't: Be hasty to discard upgrades

It happens more times than we'd like that we implement a brand new improvement or feature to our bot and... it proceeds to lose against the old version. If the winrate is close to 50% it might be just bad luck. I usually proceed as follows: if I feel that the upgrade should be good and winrate is close to 50%, I keep it. It is also really important to check for any possible bugs in the implementation, since if we miss one we might discard a pretty good improvement and never go back to it again!

[3]This may happen because of several reasons: rng seeds, fixed loop over directions, turn order, etc.

5.9 Do: The simple the better

In most cases we tend to think that a very complicated mechanism is much better than a straightforward simple approach. However, in Battlecode, it is usually the other way around. Simple stratagies usually perform better, especially when having to deal with coordination: if some global functionality can be approximated by independent behavior from each robot, this is usually the way to go.

5.10 Don't: Add too many upgrades at once

Sometimes we add a bunch of upgrades and the bot clearly starts performing worse. What happened? Which upgrades were good and which were bad? Is there a bug somewhere? Who knows, now we have to go back, revert parts of the code, and test subsets of upgrades. It would have been much much better if we had tested each upgrade individually!

Adding a lot of upgrades is often a consequence of being greedy hoping for a big increase in winrate against old bots. Often, when we only modify a part of our code, it is not reflected that much on the overall winrate. This happens because a lot of the game outcomes might not depend specifically on that particular upgrade. However, for this purpose, we can design maps in which that upgrade is especially important, and test the upgrade in those maps. For instance, did we upgrade the micro? Let's make small maps, or maps with no obstacles where all the resources are behind the base. Did we upgrade the resource gathering? Let's make maps with more resources than average. Pathfinding? Mazes and complicated maps, etc.

5.11 Do: Use the indicators and Game Constants

Battlecode provides a `GameConstants` class with all the game constants (unit stats, game parameters, etc.). We usually want to use these values instead of having magic numbers all around our code. From time to time, devs decide to change the specs to balance the game, and if this happens while having magic constants, we have to replace them all!

Additionally, Battlecode also provides some debug tools that prove to be quite useful. With the indicator commands inside `RobotController`, we can have our robots display points, lines and strings each turn, which helps a lot during the debugging process.

5.12 Don't: focus too much on bytecode optimization at the beginning

Since the number of bytecode instructions each turn is limited, it is natural to try to bytecode-optimize every single piece of our code. However, as shown in Section 6, this often takes time, and makes our code larger and harder to read. Moreover, it usually also makes our code harder to edit in the future. I'd suggest, during the first days, to avoid explicitly optimizing for bytecode and to follow good coding practices unless you believe that the potential bytecode saved by optimizing is large (e.g., when optimizing a piece of code inside a large loop). I'd only suggest to focus on Bytecode optimization once all the basics have been covered and we're in the later stages of the competition. For more insight about this, check Section 6.

5.13 Do: Write long-lasting code

Battlecode veterans like me put a lot of value on writing code that we can re-use on future competitions. However, it turns out that, in my experience, if a code is recyclable, it usually performs the best too! This can be explained by the fact that, if a code can be re-used, it probably encapsulates its functionality quite well.

5.14 Don't: Blindly use external code/algorithms

Sometimes, it is very tempting to skip a challenge by using a well-known solution or a mechanism implemented by a past Battlecode participant (e.g., those found at the #open-source chat in the Discord channel). Doing this will save some time, however we lose the potential of improvement/optimization, and it often brings trouble in the future.

For instance, suppose that we blindly implement Bugnav for pathfinding on maps with passable/impassable tiles. Now, assume that we want one of our robots (R) to go to the closest target (T) of a given type. Then, we could get the following infinite loop:

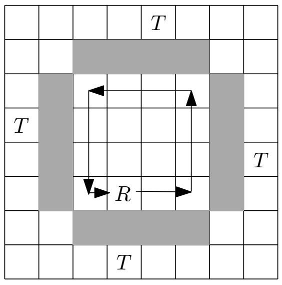


Figure 9: Example of an infinite loop with Bugnav

What happens is that the robot switches target as soon as it gets close to another wall, looping forever this way. This could be avoiding by having the robot R switch from target T to T' if and only if the distance $d(R, T')$ from R to T' is less than the historical minimum of $d(R, T)$ since d had T as target (as opposed to switch whenever $d(R, T') < d(R, T)$). It can be shown that if we switch this way, then R eventually is guaranteed to eventually reach a target. However, to prove this, it requires some knowledge about how Bugnav works.

5.15 Do: Join the community at the Discord channel!

One of the best things about Battlecode is its amazing community. Make sure to join the Discord channel. There you'll be able to discuss about strategies, implementation, etc. and help or get help from **Teh Devs** or other participants. Don't be shy!

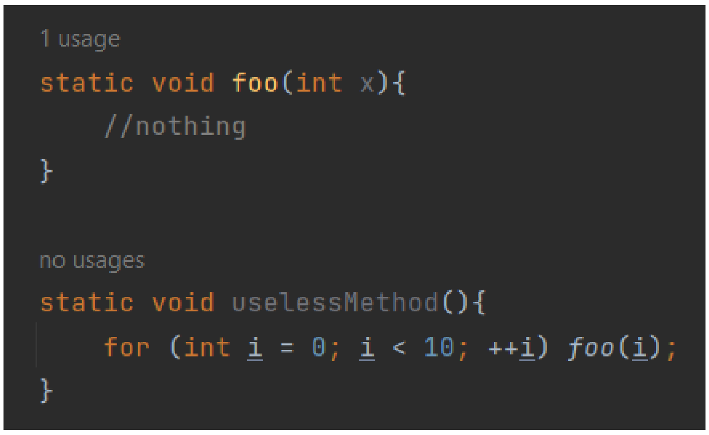
6 Bytecode

As stated in previous sections, Battlecode caps each of our robots turns by the number of bytecode instructions executed. It usually limits the number of such instructions at around 10.000 per robot. For people not used to Java, the number of bytecode instructions may sound mysterious. However, fortunately Battlecode always provide a couple of methods inside the `Clock` class that allows us to check, at any point in the code, how much bytecode we have used up to that point. During my first years I found these methods quite useful to get a feel of how much bytecode a piece of code would use.

At a high level, the more runtime complexity our program has, the more bytecode it consumes. However, even though these values are generally correlated, there are ways in which we can improve a lot the bytecode usage that do not improve the program's runtime. This makes the higher stages of bytecode optimization quite unintuitive. Going bytecode instructions in detail goes beyond the scope of this guide. If you want more information about the subject, you can read this guide from Cory Li, a former Battlecode competitor. I'll focus this section on giving some simple tips for bytecode optimization that are good to have in mind and cover most cases:

- Accessing **static class variables** cost 1 bytecode less than accessing non-static ones.
- Use **switch statements** instead of if-statements when possible. Each if-statement costs at least two bytecode: the statement itself plus the comparison. As opposed to that, all cases of a switch statement cost a total of one bytecode.

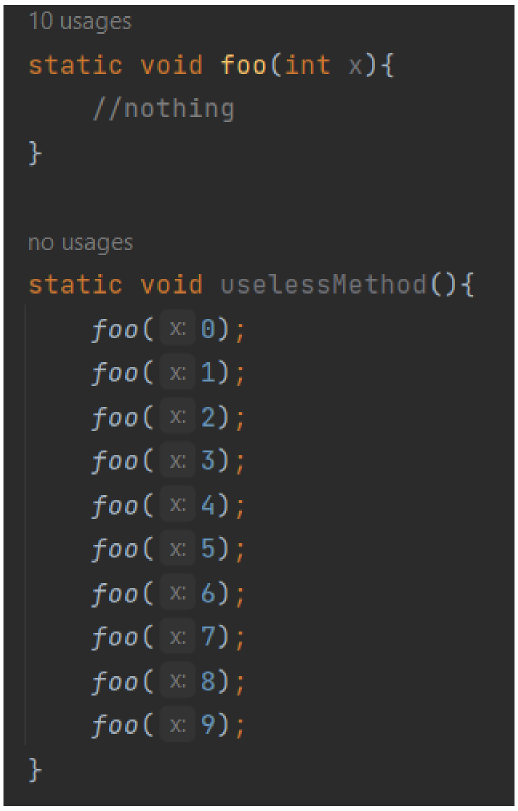
- Loops that have finite length can be improved a lot by *unrolling them.* For instance, a call of `uselessMethod()` costs 76 bytecode with the following implementation:



```
1 usage
static void foo(int x){
    //nothing
}

no usages
static void uselessMethod(){
    for (int i = 0; i < 10; ++i) foo(i);
}
```

Figure 10: Example of an uninteresting method

However, with this implementation, it costs only 21:



```
10 usages
static void foo(int x){
    //nothing
}

no usages
static void uselessMethod(){
    foo( x: 0);
    foo( x: 1);
    foo( x: 2);
    foo( x: 3);
    foo( x: 4);
    foo( x: 5);
    foo( x: 6);
    foo( x: 7);
    foo( x: 8);
    foo( x: 9);
}
```

Figure 11: Example of an uninteresting method, but unrolled

Unrolling methods can be particularly useful when dealing with loops involving all directions or all locations in visible range.

- Comparing with 0 is one bytecode less than comparing with arbitrary integers. In particular, the loop

```
for (int i = a.length; i-- > 0;) {}
```

costs one bytecode less per iteration than

```
for (int i = 0; i < a.length; ++i) {}
```

- Avoid using standard library objects such as `ArrayList`, `Queue<>`, `HashSet<>`, etc. Usually their operations are quite bytecode-expensive. Stick with simple arrays and, if you really need some of these functionalities, it is usually better to build your own object.

- Use the methods provided by Battlecode. Battlecode always has a set of methods that, instead of being instrumented as usual, they set them to have a fixed bytecode cost (you can easily find the document with all these fixed-cost methods in Battlecode's github repository). Some of these methods are much cheaper than what they'd cost with instrumentation.

- In particular, use **Strings**. Maybe this will get patched in the future since now I think it is kind of cheating, but often Battlecode allows for complex String operations and sets their cost to just one bytecode.

6.1 Codegen

As you can guess from the previous suggestions, most bytecode optimization techniques (especially loop unrolling), make our code longer and more complicated to read. Often, it makes the code so long that it is better to write a script to generate it. I've known of several teams that resorted to this technique at some point. I'll focus here in two examples I've used in the past that use most of the tips in the previous section.

If-sort

In Battlecode 2017, we needed to sort an array of integers for our micro. The built-in `sort` method would use 6000 bytecode to sort 25 elements. We tested our own quicksort implementation and that number went down to 3000. However, this number was still quite large assuming that we had only 10.000 bytecode for the whole turn. Moreover, in some turns we would have more than 50 elements in the array. In the end, we resorted to a bytecode-efficient implementation that would only work for fixed size arrays of size N: It would perform approximately $\log N$ if comparisons, and then sort the whole array depending on the outcome of these. We coded a script that would produce the code for this implementation for all $N \leq 6$ (for larger N the method was too large for Java to compile), and we also implemented a hardcoded *merge* method in the same way, that would allow us to merge two sorted arrays of length 6 each. With this, we could sort arrays of length up to 12 using minimal bytecode. To sort arrays of any size, we would perform quicksort until one of the sub-arrays had length less than 12, and then use our if-sorts in this case. Our full implementation can be found here.

Unrolled "BFS"

In Battlecode 2021 (and 2022), robots could go through any tile of the map. However, each of these tiles had its own *passability constant*, which was a number between 0.1 and 1. The cooldown added to a robot when moving from that tile was divided by the passability constant of that tile. Therefore, in order to arrive to our destination quicker, it was better to prioritize going to tiles with high passability. A standard greedy approach was choosing at every turn to move to the adjacent tile with the highest passability rate, with the constraint that it has to bring you closer to your target. However, approaches like these would not consider all the information inside vision range. For instance, in the following diagram, a robot R with 1 movement cooldown would take more than 30 turns to reach its target, when it can do it in 7:

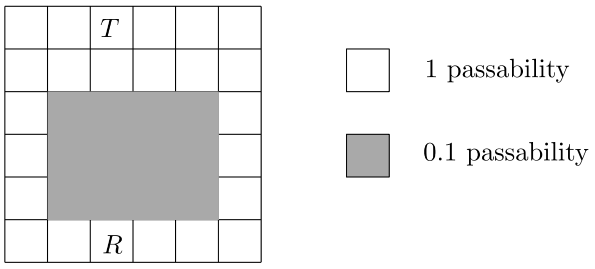
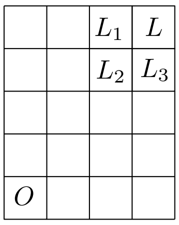


Figure 12: The robot would move through the low passability tiles.

We thought it was useful to take into account all tiles inside vision range to avoid cases like the one in the previous example. Since performing a standard BFS/Dijkstra is unrealistic because of Bytecode constraints, we came with the following relaxation. Given our original location O, for each tile L, we would compute the minimum number of turns $t(L)$ to get to L assuming we are always moving further away from O (note that it does not include all possible paths). Then, if $S(L)$ denotes the set of adjacent tiles of L' that are strictly closer to O, we have the following recurrence:

$$t(L) = p(L) + \min_{L' \in S(L)} t(L')$$

This recurrence can also be represented by this diagram:



$$t(L) = p(L) + \min(t(L_1), t(L_2), t(L_3))$$

Figure 13: Diagram of this DP algorithm

This recurrence is well defined since we can compute $t(L)$ in order, starting with the ones closest to O, and going from closest to farthest. This way, when it is time to compute $t(L)$, t will be already computed for all $L' \in S(L)$. If we were to implement this algorithm naturally, it would be quite bytecode-expensive. However, note that the order in which we should traverse the locations and the comparisons we should make (in particular, given L, which tiles are in $S(L)$) are all predetermined except for the offset of the origin. Therefore, we can unroll every single loop and get something like this, which was our implementation in 2021. Of course, this code was also generated by a script. Note that, after performing all comparisons, we check with a big switch statement if the target is in or vision range. If it is, we return the tile we should move into to get the minimal distance, and if it is not, we chose it according to some heuristic. In particular, we tried to go to the tile L in the boundary of vision range that maximized $(d(O,T) - d(L,T))/t(L)$, where $d(A,B)$ is the euclidean distance between A and B.

7 Conclusion

Battlecode is a fun and challenging coding/AI competition which I totally recommend to try! I hope this document provides a bit of insight about its fundamentals and helps new participants for their first run. For further discussion you can always find me on Discord (**XSquare**).