Stories for Iteration 2

Who owns the backlog?

The backlog is owned by a product manager or product owner. That person is responsible for establishing what the product should do and what is considered valuable for the user. Additionally, the product manager sets the priority of features to be developed.

Gaining shared understanding

This does not mean that the software delivery team has no opportunity to collaborate on the backlog. The first point of collaboration is to gain shared understanding of the product and the users of the product. So, conversations of what each story means and its value is vital.

Influencing the backlog

Secondly, the prioritisation is also up for discussion and tweaking. Sometimes, it is not possible to implement a specific feature before another is implemented first. This may be because of oversight by the product manager, or even a technical reason. However, don’t invent technical reasons for why something can’t be done in the priority requested. Instead, challenge yourselves by answering the question "How can we implement x without y?". Your solution may be temporary until the dependent story is eventually developed. That’s acceptable too because iterative development provides opportunity for us to adjust our design as we progress.

Continuous learning

Lastly, as you work through the iteration some things will arise that may need clarification. Do not assume anything. Rather ask the product manager to confirm your understanding or explain further. Use the new understanding to guide your progress.

Contributing to the backlog

As a software developer, you are also responsible for collaborating on the content of stories and scenarios. In a professional software delivery team, there may be a person that focuses on business analysis and writes stories and scenarios for at least one iteration ahead of the current iteration. This does not mean that software developers do not get involved in developing the requirements. So, be prepared to tweak existing scenarios - but discuss it with product manager to confirm the tweak. And you may also discover a scenario that was not written or a new story that may be valuable. Again, write those up and discuss it with your team, including the product manager.

Who is part of the team?

From the above, it may create the impression that the product manager is not part of the team. It’s quite the opposite. The product manager is an intrinsic part of the team with equal responsibility for successful delivery of the product. Even better is if you can include a user as part of the team too!

Who is the product manager for Robot Worlds?

> The curriculum authors and WTC technical team are the product managers. At the start of an iteration, someone from the product management team will discuss the iteration goals with you. During the iteration, you may use Slack to ask clarifying questions.

Now that we have a better understanding of the roles and responsibilities of the product manager in software delivery, here are the stories and scenarios that has been prioritised for this iteration. Note that these are not the full list of scenarios for each story. Additional stories will be prioritised for delivery in later iterations.

Story: Move Forward

Here’s the story for moving forward.

Story: Move Forward

```
As a player
I want to command my robot to move forward a specified number of steps
so that I can explore the world and not be a sitting duck in a battle.
```

To get you started, here is the story and scenario for the forward command. Note that there are additional scenarios for forward which will be implemented later.

Scenario: Moving at the edge of the world

In this scenario, the robot is at the edge of the world and attempts to move forward.

Scenario: At the edge of the world

```
Given that I am connected to a running Robot Worlds server
And the world is of size 1x1 with no obstacles or pits
And a robot called "HAL" is already connected and launched
When I send a command for "HAL" to move forward by 5 steps
Then I should get an "OK" response with the message "At the NORTH edge"
and the position information returned should be at co-ordinates [0,0]
```

Story: Launch Robot

For this story, we want implement the following additional scenarios.

- Scenario: Can launch another robot
- Scenario: World without obstacles is full

- Scenario: Launch robots into a world with an obstacle
- Scenario: World with an obstacle is full

However, it is not possible to implement these acceptance tests in a 1x1 world. We can easily increase the size of the world by hardcoding, say, a 2x2 world.

It is possible to generate a random sized or fixed size bigger world, or place random obstacles or pits or mines, or start with random starting positions. However, this randomness makes it much harder to test. The simpler the world is and the more deterministic the test data, the easier it is to write our tests.

We are going to expand the specification for our server so that we can define command line arguments to control some parameters for our server. The test scenarios can then specify an initial state of the test world, and we can use that ensure the server is launched to match that state. This gives as a fixed set of data to test against.

Server command line arguments

If you do not provide a specific argument, then the default in the table below must be used.

Name	Argument	Value	Default	Example
Server Port	`-p`	integer 0..9999	5000	`-p 5000`
Size of world, i.e. one side	`-s`	integer 1..9999	1	`-s 100` means 100x100 world
Obstacles	`-o`	position: `x,y` OR none	none	`-o 10,5` place fixed obstacle [10,5] (i.e. top-left is [10,5] and bottom-right is [10,5])

Example configurations using the command line arguments

```
# Listen on port 5000, with a world of size 1x1 and
# with no obstacles, pits or mines
java -jar target/my-server.jar

# Start server listening at port 5000 of size 100x100 and
# with an obstacle with top-left position [10,5] and bottom-right [10,5]
java -jar target/my-server.jar -p 5000 -s 100 -o 10,5
```

Options for Command Line Arguments in Java

In the class used to run your server, you have the usual `public static void main(String… args)` method that takes an array of `String` values that contains all the arguments provided when running the program.

You can write code by hand to handle the arguments being passed to your `main` method, however there are a number of useful libraries available that makes this easier.

See this tutorial on different command line argument libraries.

Adding new functionality to an existing codebase

It is tempting to just code the command line arguments into the server and then get on with the business of implementing the other scenarios. However, this approach will result in new code without tests that verify that it works. In addition, the new code may easily introduce dependencies that impacts existing classes.

This creates a bit of a conundrum. How do we build acceptance tests when the underlying functionality does not even exist? The answer lies in unit tests.

Here's one way of going about introducing the new functionality to the existing codebase.

1. Identify the scenario that requires a specific command line parameter.
2. Write the acceptance test and run it against the server. We expect this test to fail because the server does not have the configuration we need. In fact, the server will ignore the command line arguments completely.
3. Using TDD, write unit tests to configure the server so that it has a configuration that *should* make the acceptance test pass. Note that we are not yet at the point of handling command line parameters.
4. Now that we can configure the server from code (via the unit tests), we can use TDD to implement the command line argument to specify the configuration.
5. Then the last step is to spin up the server with the command line configuration and run the acceptance test to confirm that all is well.

This is one the rare occasions where we tolerate **one** failing acceptance test along with **one** other failing unit test. Do not get yourself into a knot with multiple failing acceptance tests and multiple failing unit tests. Once the unit tests pass, the acceptance tests should also *"automatically"* pass.

Remember to run the all unit and acceptance tests to ensure that we did not break other parts of the codebase.

NOTE

The file `reference-server-0.2.3.jar` contains a reference server that supports the above command line arguments.

Implementing the `port` command line argument

To implement the `port` command line argument, go back to a scenario such as *Launch a robot* from the previous iteration.

Here's a rough outline of how to go about implementing the `port` command line argument. Note that your context may be different because your codebase may have implemented the port number in its own way. Below, we assume that the port number is a hardcoded.

1. Remove the hardcoded port number and run the acceptance test for the one scenario. The test should fail because the port number is invalid. We now have that **one** failing acceptance test.
2. Using TDD, write the code to implement the default port number of `5000` if the port is not specified.
3. Run the failing acceptance test and it should pass.
4. Run the acceptance test with a different port configuration. We expect this test to fail because the port defaults to `5000` .
5. Using TDD, write the code to implement the specified port.
6. Now run the acceptance test with the port configuration specified by the command line argument and the test should pass.

IMPORTANT

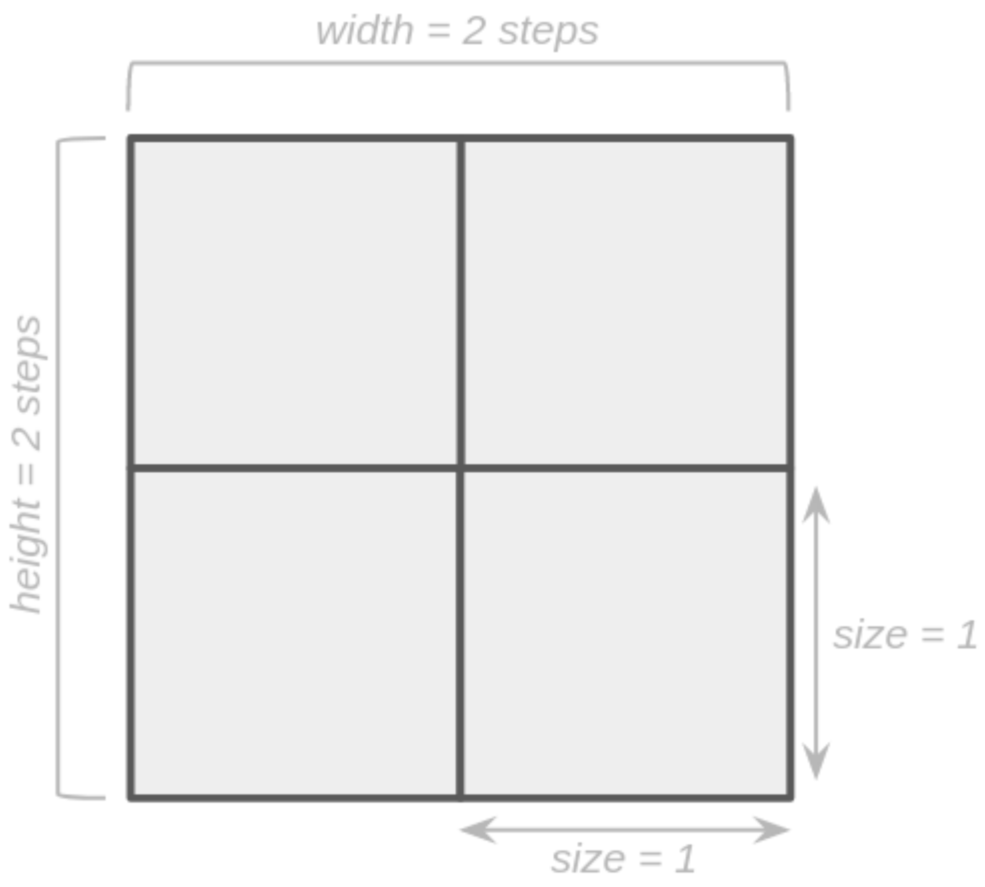
Make sure that all unit tests and acceptance tests pass before tackling the rest of the scenarios below.

Refresher: World Coordinates

In the next few scenarios, we will write acceptance tests for a 2x2 world. As a quick refresher, this is what is meant by a 2x2 world as per the original requirements.

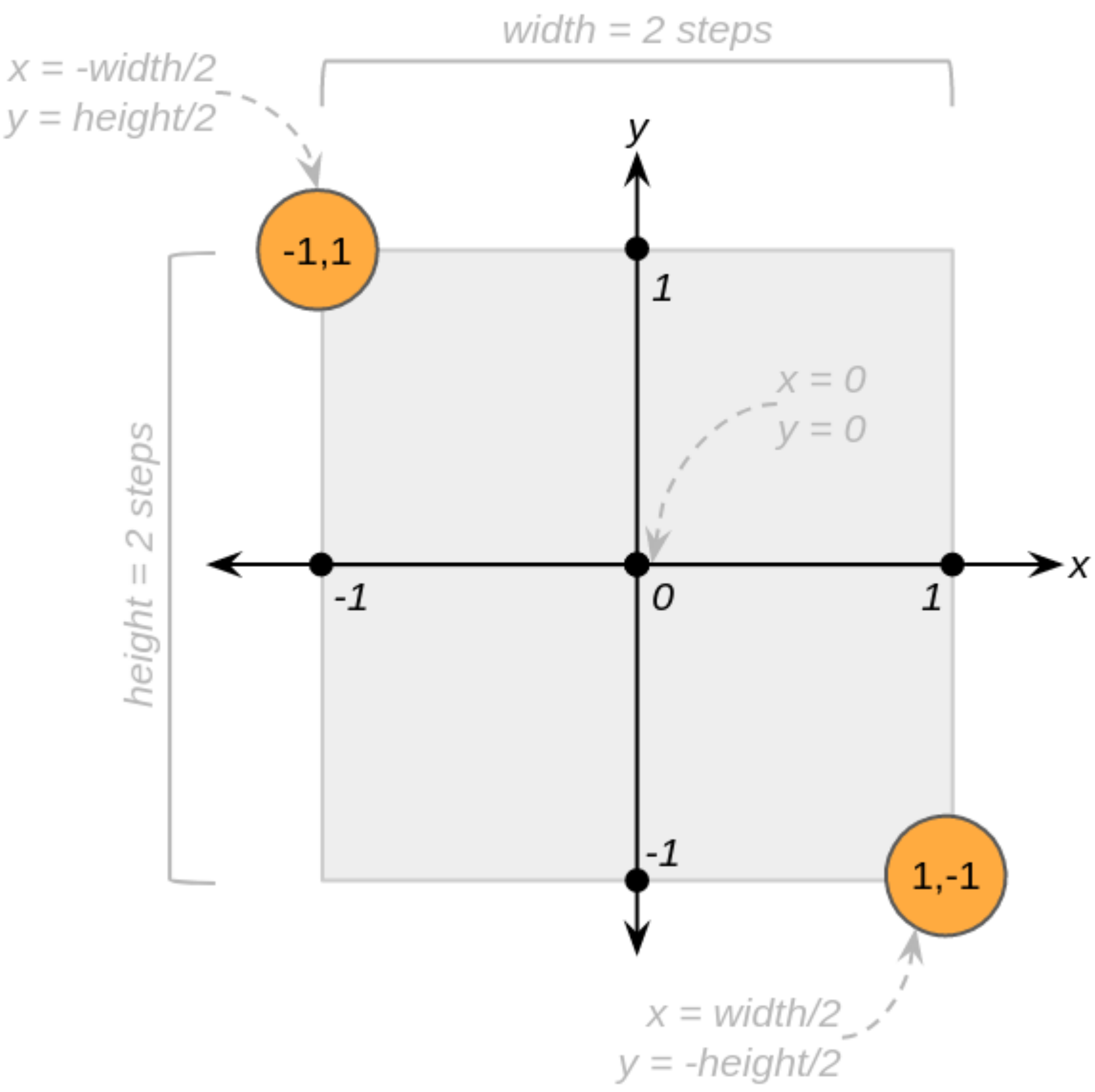
Size of the world

The size of the world is specified as a width and height measured in steps. The width is the absolute length of the top or bottom side. The height is the absolute length of the left or right side.



World coordinates

The locations which can be occupied by robots and other objects are specified using an `(x,y)` co-ordinates. The centre of the world is always at postion `(0,0)`. In a 2x2 world, each edge is 1 step away from the centre.

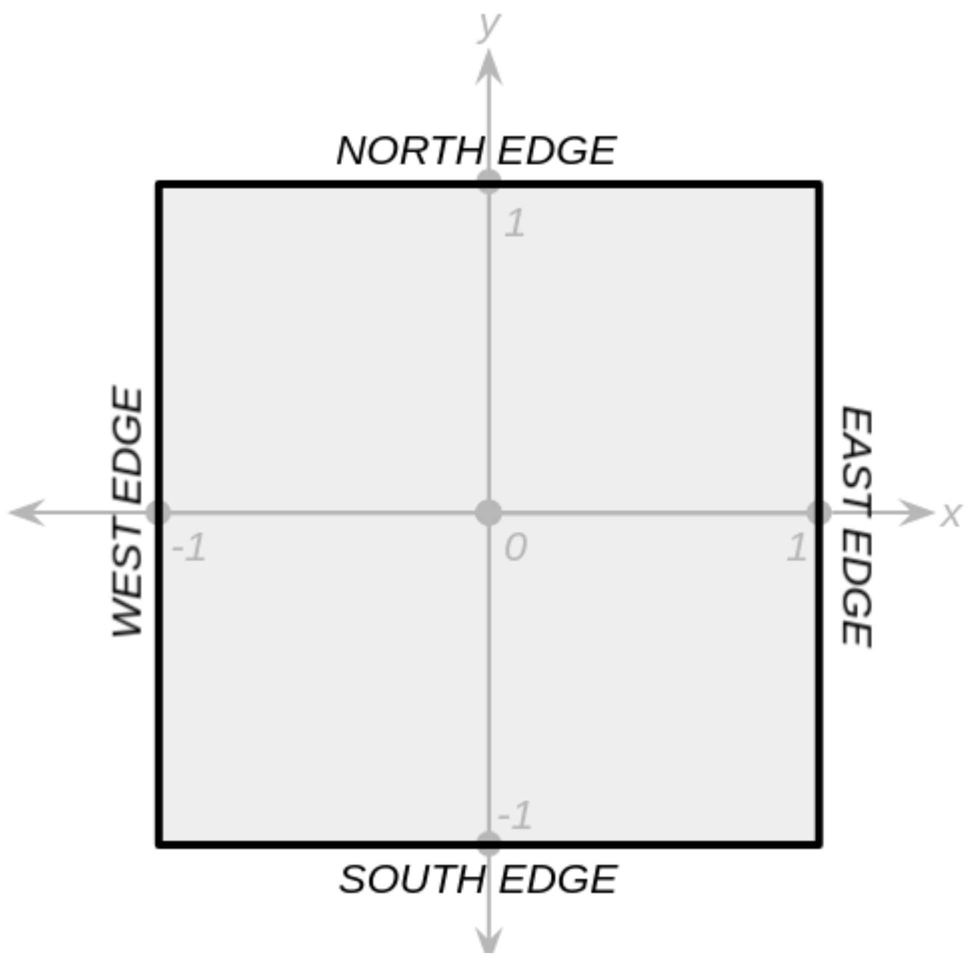


NOTE

Using these definitions of top-left and bottom-right, a 3x3 world will have the same coordinate system as a 2x2 world. In fact any `NxM` world where `N` and `M` are odd numbers, the coordinates will be same as an `(N-1)x(M-1)` world. We can tolerate with this anomaly.

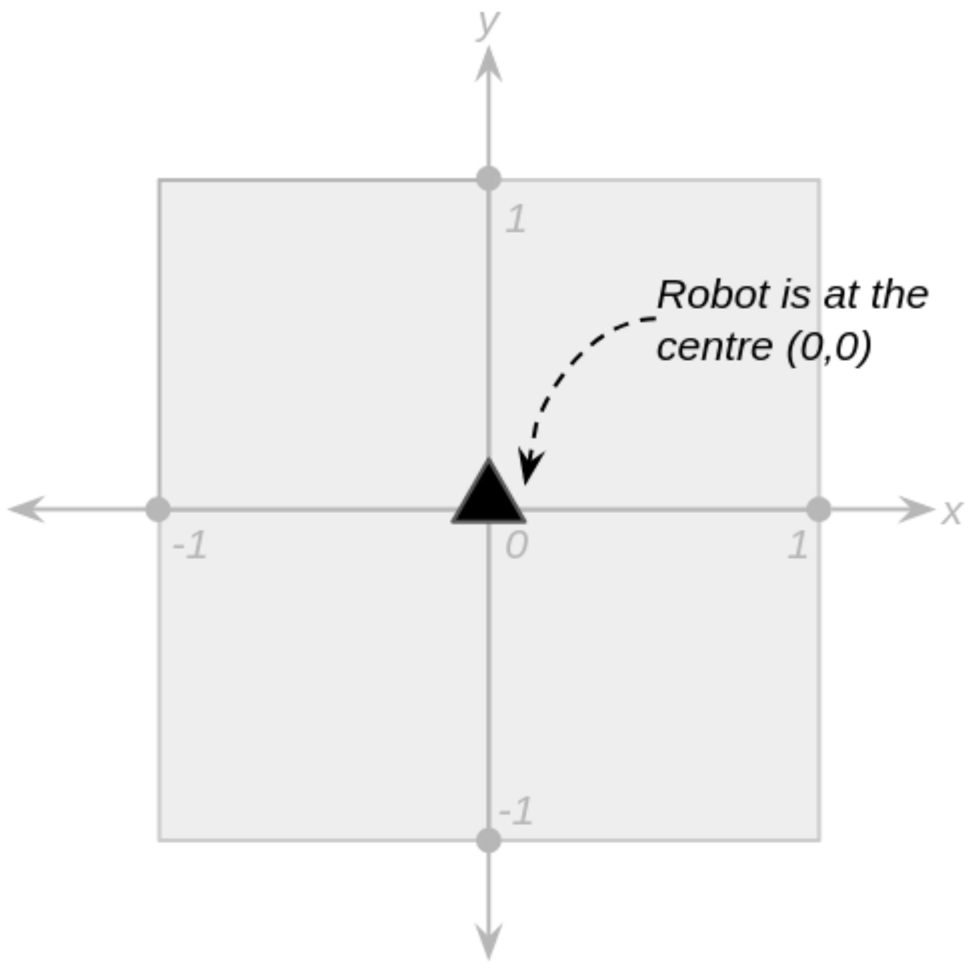
The edges of the world

Now that we now the meaning of the size of the world, and the coordinate system, let's look at the compass directions and edges. The north edge is an location that is at the top of the world; i.e. having a maximum possible `y` value. The south edge is an location that is at the bottom of the world; i.e. having a minimum possible `y` value. The east edge is an location that is at the right of the world; i.e. having a maximum possible `x` value. The west edge is an location that is at the left of the world; i.e. having a minimum possible `x` value.

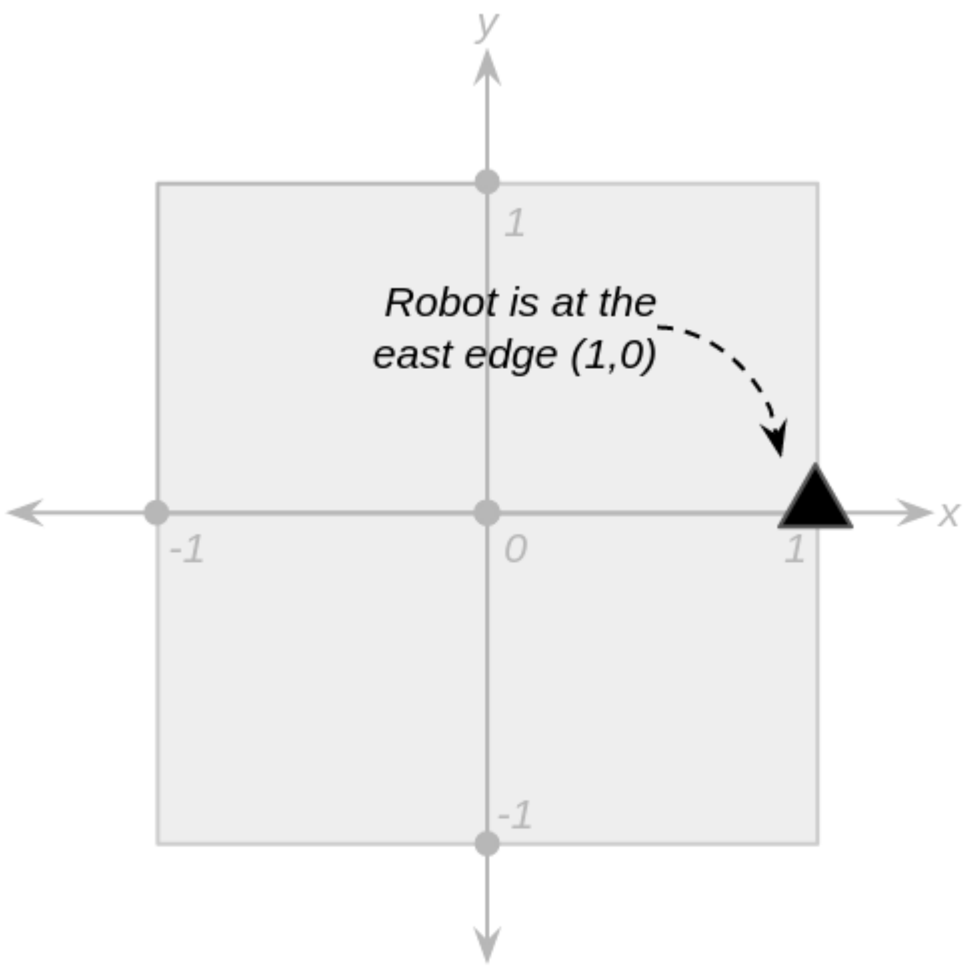


Positions in the world

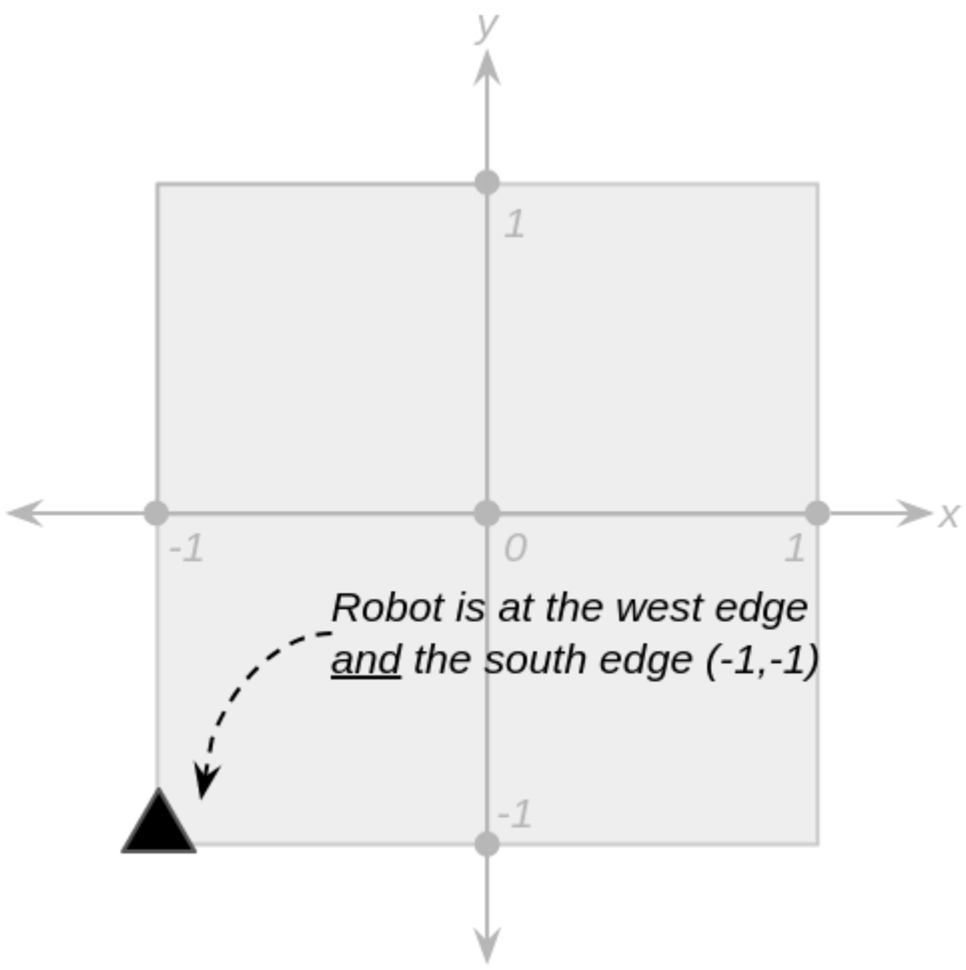
Here, the robot is at the centre `(0,0)`



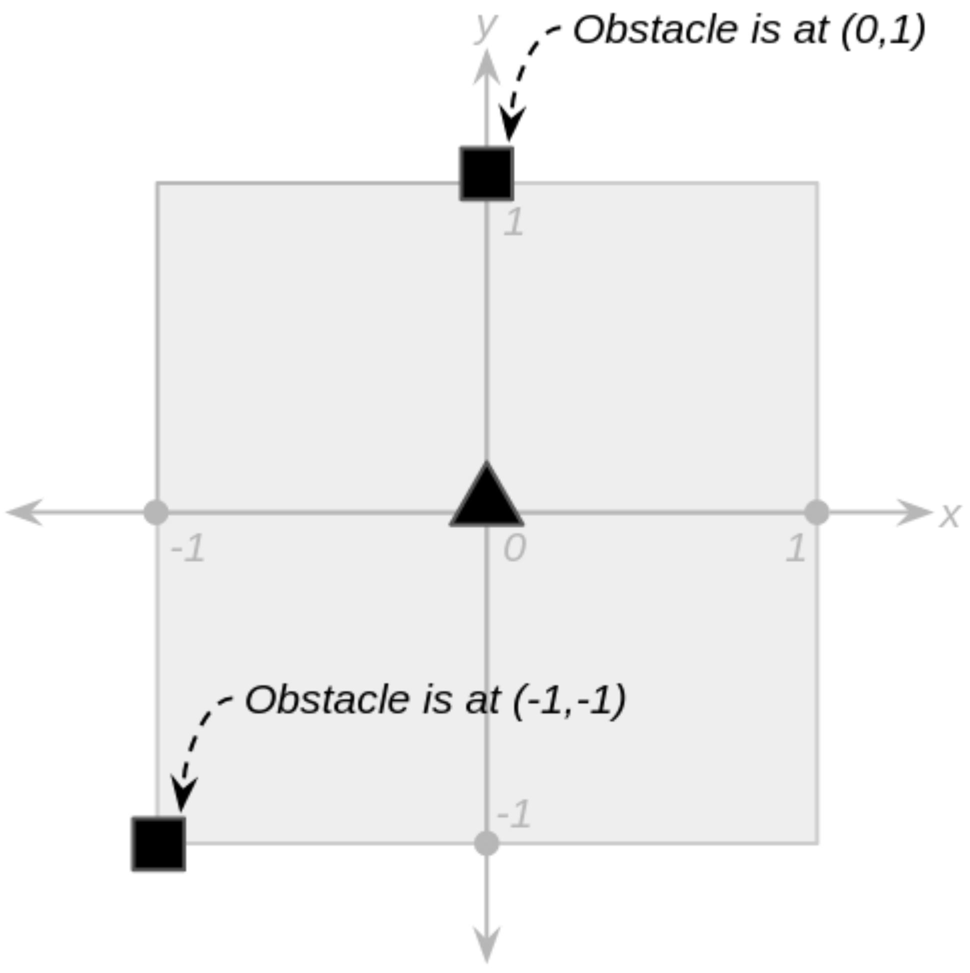
This robot is at the one of the 3 possible east edge positions `(1,0)`.



And this robot is at the west and south edge `(-1,-1)`.

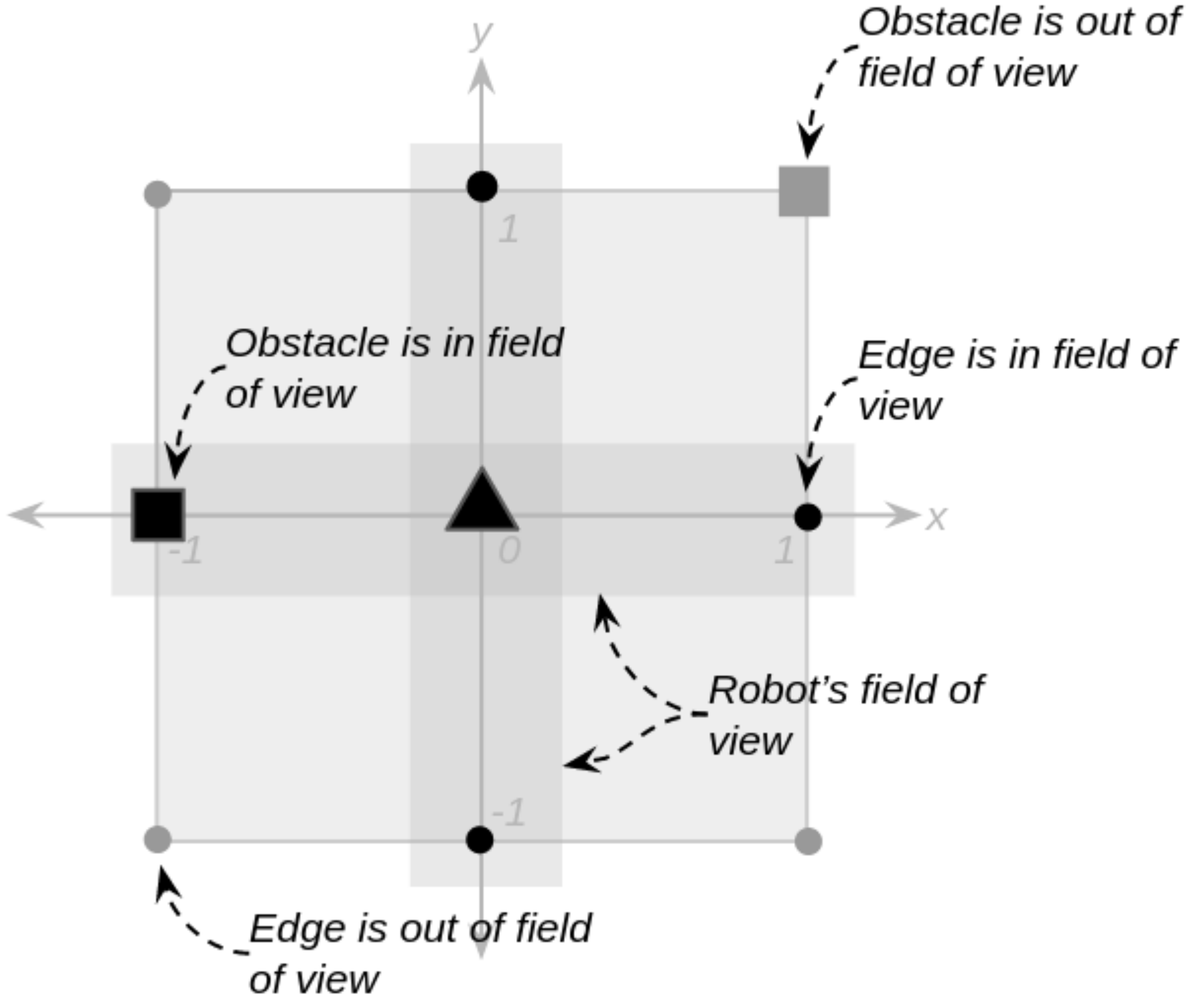


Here is a world with one robot and two obstacles.



Field of view

A robot can only see obstacles directly in front, behind, left and right of it - all along straight lines. This means that it cannot see objects and edges that are diagonal to it in any direction.


y
Obstacle is out of field of view
1
Obstacle is in field of view
Edge is in field of view
x
-1
0
1
Robot's field of view
-1
Edge is out of field of view

Scenario: Can launch another robot

Here's the scenario for successfully launching a second robot into the world. Note that the location of the new robot is not specified

Can launch another robot

```
Given a world of size 2x2
and robot "HAL" has already been launched into the world
When I launch robot "R2D2" into the world
Then the launch should be successful
and a randomly allocated position of R2D2 should be returned.
```

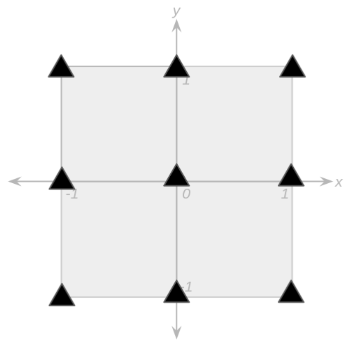
Scenario: World without obstacles is full

In this scenario, the world is full and the another robot cannot be launched. This seems to be a duplication of the acceptance test from the previous iteration. However, in that iteration the world was 1x1 and, maybe, we did not need to write code to find a random unoccupied location. So, this scenario is a sanity check for the additional code that we ended up writing for the previous scenario.

World without obstacles is full

```
Given a world of size 2x2
```

```
and I have successfully launched 9 robots into the world
When I launch one more robot
Then I should get an error response back with the message
"No more space in this world"
```

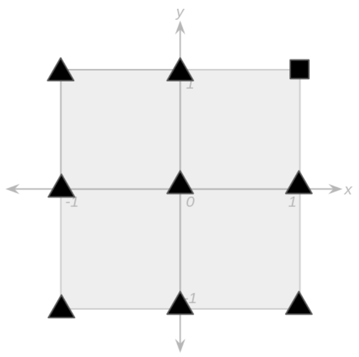


Scenario: Launch robots into a world with an obstacle

In this scenario, you will have to implement the command line configuration for setting up an obstacle in the world. Remember the approach of using unit tests in conjunction with the acceptance test.

Launch robots into a world with an obstacle

```
Given a world of size 2x2
and the world has an obstacle at coordinate [1,1]
When I launch 8 robots into the world
Then each robot cannot be in position [1,1].
```

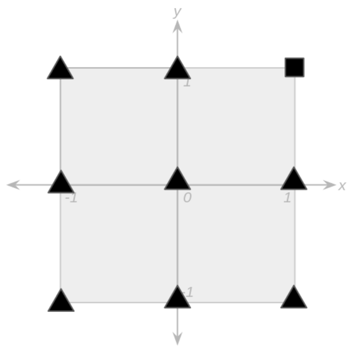


Scenario: World with an obstacle is full

Notice how we are testing just one slight variation with this scenario. It is tempting to combine this with the other scenarios. Though, keeping it separate allows us to evolve our codebase in small frequently taken steps.

World with an obstacle is full

```
Given a world of size 2x2
and the world has an obstacle at coordinate [1,1]
and I have successfully launched 8 robots into the world
When I launch one more robot
Then I should get an error response back with the message
"No more space in this world"
```



Story: Look

Now that we can configure a world with objects, we can cater for the scenario wherein a robot can see an obstacle and other robots.

Scenario: See an obstacle

Notice that we are not yet dealing with the case of visibility. That will be dealt with in later iterations

See an obstacle

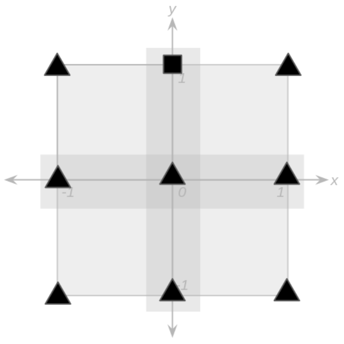
```
Given a world of size 2x2
and the world has an obstacle at coordinate [0,1]
and I have successfully launched a robot into the world
When I ask the robot to look
Then I should get an response back with an object of type OBSTACLE at a distance
```

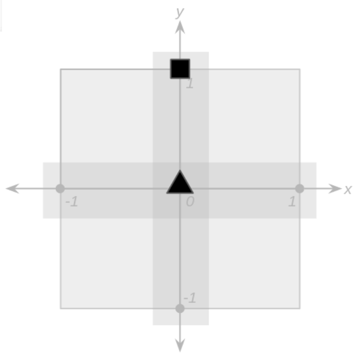
```
of 1 step.
```

Scenario: See robots and obstacles

See a robot

```
Given a world of size 2x2
and the world has an obstacle at
coordinate [0,1]
and I have successfully launched 8
robots into the world
When I ask the first robot to look
Then I should get an response back
with
one object being an OBSTACLE that
is one step away
and three objects should be ROBOTs
that is one step away
```





What you need to do for this goal

The following scenarios must be developed for this iteration. These stories must be completed using acceptance tests and unit tests as needed.

- Story: Move Foward
  1. Scenario: Moving at the edge of the world
- Story: Launch Robot
  1. Scenario: Can launch another robot
  2. Scenario: World without obstacles is full
  3. Scenario: Launch robots into a world with an obstacle
  4. Scenario: World with an obstacle is full
- Story: Look
  1. Scenario: See an obstacle
  2. Scenario: See robots and obstacles