# Reviewer guidelines for the Robot coding challenge

The Toy Robot is based on a blog post by Jon Eaves. Read it here https://joneaves.wordpress.com/2014/07/21/toy-robot-coding-test/

It's a common test used by a few different Melbourne based companies, and for got reason. It's simple and can be completed in a reasonable time frame, but is also fairly flexible and open ended in the way a developer can approach it. Because of this, you can gain great insight into how the developer thinks about code. Do they consider the problem domain in their code? Do they correctly use design patterns? Are they a functional or object-oriented programmer? Do they over-engineer or practice YAGNI?

The Toy Robot also provides a great starting point for an in-interview pairing exercise. Candidates can work on code they are familiar with, which helps with creating a flowing pairing session in a limited time window.

## Aspects of the problem

There are 3 main areas to deal with in this problem that apply no matter what programming paradigm you use:

1. Handling the user input and output

2. Command parsing, validation and routing

3. Domain structure and logic

### Handling I/O

The instructions call for a choice between using file based or CLI based IO. Which ever option is chosen, look for appropriate use of I/O libraries, separation of concerns and decoupling. Consider whether the I/O could be replaced with a different mechanism (e.g. switching from CLR to file to web UI) without needing extensive refactoring throughout the application.

### Command parsing, validation and orchestration

Extra points should be awarded for people that use regular expressions to validate and parse string based user input. If using an OO language, this part of the exercise lends itself nicely to a command pattern. If they choose this option, then commands should be thin with no domain logic in them. They should defer to other domain objects e.g. Robot. A nice touch with a command pattern is to make use of reflection and a HashMap to route commands. This removes the need for ever growing if/switch statements. There are other acceptable approaches, however the main thing to look for is readability and extensibility. If adding a command requires modifying code in multiple places, it might be more complicated than it needs to be.

### Domain structure and logic

There are 2 obvious domain concepts in this problem. The Robot and the Table. You should see both of these represented in some structure of encapsulated domain objects. We are looking for high cohesion which means there should not be unrelated logic living in the same place. For example, if you see string parsing in a Robot class, with would be a bleeding of the IO concerns into the Robot domain. The robot should work with normalised data and not be couple to the structure of user commands. Another common mistake when using the command pattern is to put robot logic in the commands. This is the opposite of encapsulation and creates tight coupling across the the rest of the app.

## Key practices and principles to look for

There are so many ways to approach this problem, but the principles we value in software engineering are consistent no matter how you approach it.

\*\*YAGNI and over engineering\*\* - the solution should not make provision for future functionality at the cost of complexity. Code that is easy to reason about will inevitably be easy to extend. For example, you'll often see OO solutions with up to 10 different classes covering every possible abstraction. The core of this solution should require less than 200 lines of code with varying amounts of boilerplate depending on the language.

\*\*TDD\*\* - although you can't know if they did their tests first, you can pick up on evidence they considered testablity when writing the implementation. Generally you'll get 2 types of tests, unit tests and end-to-end tests. Neither is better than the other. However they should be written correctly. Unit tests should follow a BDD style, mock their dependencies and assert one thing. End-to-end tests should be complete (test the full E2E), comprehensive (but not exhaustive) and fully automated. Sometimes you'll see E2E tests that don't actually test the full E2E. Likewise with unit tests, you'll see double testing, broad tests and mis-use of mocks. It's also common for people to provide integration tests that skip some of the IO and command orchestration logic. Lookout for which parts of the application are highly testable and which parts are impossible to test. A common interview question is to ask how a particular area could be refactored to improve testability.

\*\*Imperative vs declarative code\*\* - this is pretty easy to spot. If you're seeing long nested `if` statements or methods that do lots of things, this is likely to be excessive imperetive code. Generally, we like to contain imperative logic into composable chunks that can be assembled in a declarative fashion. This creates code that is easy to reason about. The Robot challenge lends itself well to this style.

\*\*State management\*\* - particularly for functional solutions, we should see use of immutability and pure functions with state objects being passed around. For OO, we're more concerned with correct encapsulation of state in domain objects.

\*\*SOLID, OO principles and design patterns\*\* - Single responsibility, encapsulation, polymorphism, coupling, inversion of control, etc. - generally speaking, a solution that follows these principles will be simple, extensible, easy to reason about and highly testable.

\*\*Appropriate abstractions\*\* - look for evidence of a domain structure that mirrors the problem description. Look for language consistent with the instructions e.g. do they have a Robot class, a Table class, the commands are named consistently with the requirements. If they introduce new terms, do they make sense and add to the domain. A solution with appropriate abstractions will feel very easy to navigate and locate the various parts of the program. A lack of abstractions or over abstractions will feel like wading through mud.

\*\*A note on frameworks\*\* - in the instructions, we specifically discourage the use of frameworks because we want to see the individuals interpretation of how to solve the somewhat simple problems presented by this exercise using basic design patterns and coding techniques. If you see heavy framework use, consider what is not demonstrated by hiding behind the framework. This could result in interesting questions and discussion in an interview.

## Extending the solution in a pairing exercise

The following are some extensions for an in-interview pairing exercise. It's best to choose one that's appropriate for the candidates solution. Don't choose one that will require a complete rewrite, but maybe discuss the limitations of their design.

### Make the table size configurable

The user enters a number as the first command to configure the table size. If they've written their code well, this will have minimal impact and can be completed quickly.

### Add 2 (or n) robots on the table

Allow the user to specify n number of robots that can be placed with the place command. This introduces a robot ID that needs to be passed to each command. The new format of the commands looks like this:

```

Enter number of Robots

> 2

> PLACE 1,2,NORTH

places robot with ID 1

> PLACE 1,3,NORTH

places robot with ID 2

> MOVE 1

will throw a collision error

> RIGHT 1

> LEFT 1

> REPORT 1

```

### Add more directions

Add the diagonal directions: NORTHEAST, NORTHWEST, SOUTHEAST, SOUTHWEST. Allow the robot to be placed facing these directions and when move is called, the robot moves to the next diagonal location.

### Add obstacles

Allow the user to enter an obstacle with a command `OBSTACLE X, Y, SIZE`. The size specifies how many blocks to take up in a square shape with X,Y denoting the bottom left corner.

### Support a different shaped table

Support a table in the shape of a cross or any arbitrary shape.

### Macros

Create new commands that execute a combination of movement e.g. double move, perform a 1 square loop, move in a diagonal line.

### Add a 3rd dimension

Allow a Z dimension in the placed command and track the robot in a 3D cube.