CHAPTER 1

A little Saturn PathFinder

We launched a pathfinder robot on Saturn and the communication with the robot is difficult. To interact with this robot we send it *orders* in *scripts* from Earth and the robot executes them. Energy and communication are limited so we use a compact representation of scripts.

Your mission is to define different orders and functionalities such as replay, way back home, and path optimizations.

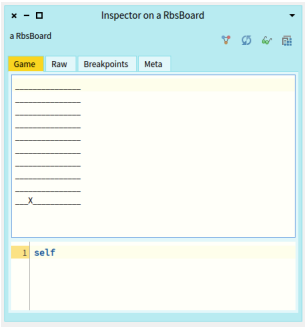

Figure 1-1 A 2D space and a robot in ascii.

Doing this project you will learn the Command design pattern and delegation to objects that encapsulate their behavior.

1.1 A robot in its space

A robot lives in a 2D space. It starts in a location. The following code snippet is producing Figure 1-1.

```
| rb b |
rb := RbsRobot new.
b := RbsBoard new.
rb setBoard: b.
rb x: 4 y: 1.
rb inspect
```

A board is composed of cells. Pay attention that a cell in the board only contains one element: a ground tile or a robot. So when moving a robot to a cell will 'erase' the background. So when moving a robot should put back the previous tile.

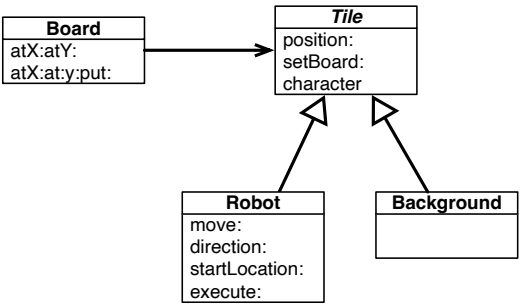

Figure 1-2 A minimal design.

1.2 Scripts

A robot receives a script as strings containing *orders*. The following test illustrates this.

- First a robot is created.
- Second a board is created. The robot is placed in the space.
- Third the robot can execute a script.

```
testExecute

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb execute:
```

```
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

The script contains different orders: such as `mov 3`, `dir #north`.

1.3 Getting the code

To help you develop this project we provide some core behavior. The robot code is available at: `https://github.com/pharo-mooc/AdvancedDesignMoocProjectCo`

- To start, load the baseline name `RobotsProject`, it contains the board logic, and board tests in addition to basic behavior for tiles composing the space. Note that this
- Once you define the class `RbsRobot` (for example in a package named `Robots`) as a subclass of `RbsAbstractRobot`, load the package `Robots-Tests`. It contains the tests for the behavior you will have to define.

1.4 Basic robot behavior

Define methods `direction:` and `direction` to define the direction of the robot and initialize it for example to point to the east.

```
testRobotDefaultDirection

  | rb |
  rb := RbsRobot new.
  self assert: rb direction equals: #east
```

1.5 Robot move

The first step is to implement orders such as `mov`, `dir`. Each order can be implemented by defining a method such as `move: aDistance` and `direction:`. Propose an implementation for these methods. Here is a possible test for the `move:`.

```
testRobotMove

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  "should make sure that previous tile is put back"
```

```
  self assert: (rb board atX: 4 atY: 1) equals: rb.
  rb move: 10.
  self assert: (rb board atX: 14 atY: 1) equals: rb.
  self deny: (rb board atX: 4 atY: 1) equals: rb
```

Pay attention that `move:` should put back the ground after moving.

To help you we propose to use the following method `computeNewPosition:`, but there is a bug (it does not return a point). Write a couple of tests and fix the method.

```
computeNewPosition: anInteger
  "Returns a point representing the location of the next move."
  ^ direction = #east
    ifTrue: [ self x + anInteger ]
    ifFalse: [ direction = #west
      ifTrue: [ self x - anInteger  ]
      ifFalse: [ direction = #north
          ifTrue: [ self y + anInteger ]
          ifFalse: [ self y - anInteger ].
        ]
      ]
```

The method `move:` now handles the fact that we put back the background tile when moving the robot. But we were tired and there was a bug in that method, fix it!

```
move: anInteger

  | newPosition |
  newPosition := self computeNewPosition: anInteger.
  self previousTile position: newPosition.
  previousTile := self board atPosition: newPosition.
  self position: newPosition.
```

The following test should pass:

```
testRobotMovePreservesGround

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  self assert: rb previousTile class equals: RbsGround.
  self assert: rb previousTile x equals: 4.
  rb move: 10.
  self assert: (rb board atX: 4 atY: 1) class equals: RbsGround.
  self assert: (rb board atX: 14 atY: 1) equals: rb.
  self assert: rb previousTile position equals: 14@1
```

1.6 Sending order to robots

Now we are ready to implement the method `execute:` that will execute the orders. The following helper method splits the script into line based orders.

```
RbsRobot >> identifyOrdersOf: aString

  | orders |
  orders := aString splitOn: Character cr.
  orders := orders collect: [ :each | each splitOn: Character space
    ].
  ^ orders
```

In addition you can use the following expression `Object readFrom: aString` to get the Pharo object represented by the string.

```
Object readFrom: '1'
> 1

Object readFrom: 'true'
> true
```

You should make the following test passes:

```
testExecute

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb execute:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

1.7 Adding new orders

We propose now to introduce new orders.

Base

It was strange to not have the base position as part of the script so we propose to introduce a new order `base` taking two numbers as x and y.

```
base 10 20
```

Here is a test that should pass.

```
testStartPositionAsOrder

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb execute:
'base 4 1
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

Dropping an item

Introduce the possibility for the robot to drop an item on the map. Introduce the class `RbsItem` with, for example, the character `o` as textual representation and handle the order in the `execute:` method.

```
dropL
```

1.8 Introducing commands

Imagine that you originally defined the `execute:` method as follows:

```
execute: aString

  (self identifyOrdersOf: aString)
    do: [ :each |
    each first = #mov
      ifTrue: [ self move: (Object readFrom: each second) ]
      ifFalse: [
        each first = #dir ifTrue: [
          self direction: (Object readFrom: each second) ] ] ]
```

You certainly saw that adding a new order is tedious and make the conditional statements more and more complex. This can get even more complex if we want to implement a replay of the orders. We propose to use Commands. Commands are objects representing actions.

Load the package named `Robots-BasicCommands-Tests`. It contains some tests to help you creating commands.

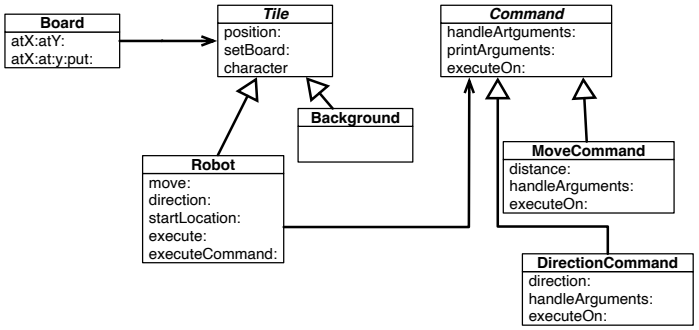

Figure 1-3 A design with Command.

Command

Each command can have its own state and in addition its should know how to execute itself and convert the order arguments into a Pharo object. Here is an example for the `RbsMoveCommand`. What you see is that it has its own state, an `executeOn:` method and a way to handle the arguments of the script.

```
RbsCommand << #RbsMoveCommand
  slots: { #distance };
  package: 'Robots'
```

```
RbsMoveCommand >> executeOn: aRobot
  aRobot move: distance
```

```
RbsMoveCommand >> handleArguments: aCol
  distance := Object readFrom: aCol first
```

Registering commands

We need a way to associate orders to commands. We do it by defining the method `commandName` on the class side of the command class.

```
RbsMoveCommand class >> commandName
  ^ 'mov'
```

The robot class should have a way to map 'mov' to the class of the command Something like:

```
initializeCommandMapping

  cmdMap := Dictionary new.
  RbsCommand allSubclassesDo: [ :each |
    cmdMap at: each commandName put: each
    ]
```

that the `executeCommandBased: aString` should use when creating and executing commands.

```
executeCommandBased: aString

  (self identifyOrdersOf: aString) do: [ :each |
    ((self commandClassFor: each first) new
      handleArguments: each allButFirst; yourself) executeOn: self ]
```

Implement all the commands so that the following test should pass.

```
testExecuteCommandBased

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb executeCommandBased:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4
```

1.9 Challenge: Replay

We would like to monitor what the robot is doing to be able to replay it. Load the package `Robots-Replay-Tests`. Here is a typical script and we can replay it with another starting position.

```
testReplay

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
  rb x: 5 y: 1.
  rb replay.
  self assert: rb position equals: 10@4
```

Let us imagine that the method execute `commandBased:` was implemented as

```
RbsRobot >> executeCommandBased: aString

  (self identifyOrdersOf: aString) do: [ :each |
    ((self commandClassFor: each first) new
      handleArguments: each allButFirst; yourself) executeOn: self ]
```

You should introduce a way to keep the created commands so that they can be replayed. For example consider adding an instance variable `path` initialized as an OrderedCollection and add commands when you create them in the previous method.

Introduce new commands to control replay

Note that in the test above we used `rb x:5 y: 1.` instead of `rb executeCommandBased: 'base 5 1'`. This is due to the fact that we cannot control when the recording is starting and that we cannot reset it or stop it either. We propose you to introduce the following commands: `startM`, `stopM`, `restM`, and `replay`.

Add a new instance variable monitoring to the robot class and two methods to control it as well as an initialization.

```
startMonitoring
  monitoring := true
```

```
stopMonitoring
  monitoring := false
```

The following test shows that we are registering `stopM` as a command. We will fix that below.

```
testMonitoringIsOnPerDefault

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
mov 3'.
  self assert: rb path size equals: 3
```

The following test verifiess that once the monitoring is stopped and the path reset, the path is empty

```
testReset

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
resM
mov 3'.
  self assert: rb path size equals: 0
```

1.10 Non recording commands

The following test may loop so pay attention because replay will replay the sequence that will replay itself endlessly.

```
testReplayAsCommand

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'resM
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
rb executeCommandBased: 'base 5 1
replay'.
  self assert: rb position equals: 10@4
```

We could rely on the script programmer to always stop the monitoring before placing a replay order. But to have better security and avoid endless loop because `replay` would be replaying itself, it is important that `replay` is not added to the path of commands. The following test loops because the replay order is causing itself to be kicked.

Propose one solution where `replay` is not added to the path. Such a solution can be defined without any conditional by giving each command the responsibility to add itself to the path.

Instead of doing a conditional before adding the command the path, we can just ask the command to add itself to the path of the robot. This way the replay command can ignore it. So introducing a hook in place of calling directly the path addition (`path addLast: cmd.`) is a nice solution because each command can define its own behavior.

```
executeCommandBased: aString
  ...
  path addLast: cmd.
  ...
```

becomes

```
executeCommandBased: aString
  ...
  cmd addToPathOf: self
  ...
```

This forces us to introduce a method named for example `addToPath:` in the robot class to expose path addition. Once the corresponding logic is added and used the following test will pass.

```
testAddToPathCommandsDoesNotContainReplay

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
mov 3
replay'.
  self assert: rb path size equals: 3
```

Once the command `stop`, `start`, `reset` and `replay` are not recorded anymore the tests should be changed. For example `testMonitoringIsOnPerDefault` checks that the path is now only containing two commands.

```
testMonitoringIsOnPerDefault

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 5 1
dir #east
stopM
mov 3'.
  self assert: rb path size equals: 2
```

Now we are ready to use `replay` as an order. The following test verifies it.

```
testReplayAsCommand

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb executeCommandBased:
'base 4 1'.
  rb executeCommandBased:
'resM
dir #east
mov 2
mov 3
dir #north
mov 3'.
  self assert: rb position equals: 9@4.
rb executeCommandBased: 'stopM
base 5 1
replay'.
  self assert: rb position equals: 10@4
```

1.11 Challenge: Automatic way back home

It can be tedious to bring back the robot to its location be inverting one by one the orders that compose a script. We propose to enhance our robots with a `wayBack` order. Load the package `Robots-WayBack-Tests`. A way back action with take a list of commands and produce a new list of commands with the opposite actions. Figure 1-4 illustrates the behavior:

When we have a simple path

```
dir #east
mov 5
dir #north
mov 3
dir #east
mov 4
wback
```

the robot should perform the following path back. We stressed that the directions should be inversed.

```
dir #east => west
mov 4
dir #north
mov 3
dir #east => west
mov 5
```

What we see is that we should not only

- remove the way back order
- reverse the list
- but also convert direction in the opposite ones.

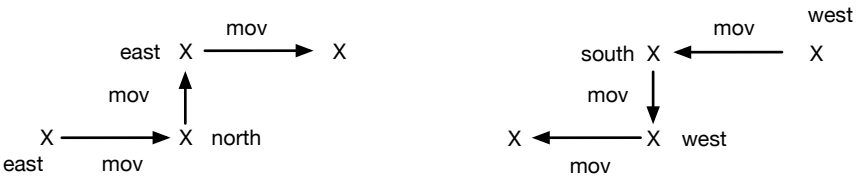

Figure 1-4 A simple path and a way back home.

Notice that multiple mov orders can be before a change direction as in the equivalent path:

```
dir #east => west
mov 2
mov 2
dir #north
mov 1
mov 1
mov 1
dir #east => west
mov 5
```

A a first step we propose to introduce a simple message on the direction command class and the root of command.

```
RbsCommand >> asWayBack

  ^ self
```

Imagine the implementation for the direction commands.

```
testDirectionWyaBAck
  | opposite |
  opposite := (RbsDirectionCommand new direction: #east) asWayBack.
  self assert: opposite direction equals: #west.
  opposite := (RbsDirectionCommand new direction: #west) asWayBack.
  self assert: opposite direction equals: #east.
```

To help you in this challenge we propose you to use the following method `ifCutOn: isSplitterBlock doWithCutAndUncuts: aTwoArgBlock finally: aBlock`. If it is not available in Pharo, just define it on `SequenceableCollection`. The following tests should illustrate clearly what the method does.

```
testCut

  | res |
  res := OrderedCollection new.
  #(2 2 #east 1 1 1 #north 5 #east 666)
    ifCutOn: [ :s | s isSymbol ]
    doWithCutAndUncuts: [ :cut :before | res addLast: cut; addAll:
    before ]
    finally: [:u | res addLast: u].
  self assert: res equals: #(#east 2 2 #north 1 1 1 #east 5 666)
    asOrderedCollection
```

```
SequenceableCollection >> ifCutOn: isSplitterBlock
    doWithCutAndUncuts: aTwoArgBlock finally: aBlock
  "Applies aTwoArgBlock (with current splitter objects and previous
    unsplit objects) to the receiver. When uncuts are left executes
    aBlock with them.
  An optimised version could work with indexes to avoid creating
    intermediate collections."

  | uncuts cut current |
  uncuts := OrderedCollection new.
  1 to: self size do: [ :i |
    current := self at: i.
    cut := isSplitterBlock value: current.
    cut
      ifFalse: [ uncuts addLast: current ]
      ifTrue: [
          aTwoArgBlock value: current value: uncuts.
          uncuts := OrderedCollection new ]].
  uncuts isEmpty
    ifFalse: [ aBlock value: uncuts ]
```

Extensions

- We could introduce a turn back message that given a command return its opposite based on its previous state. Given a path sequence east mov 5 north mov 3 east mov 7 it would generate the sequence west mov 7 north mov 3 south mov 5...

1.12 Challenge: Path optimizations

This extension is about supporting path optimizations. Load the package 'Robots-Optimize-Tests. Let us imagine that the treatment of a command is costly on Saturn. Then it can be better to optimize the received script before executing it. Optimization can be

`quite simple, indeed n mov‘` commands can be merged as a single move command with the sum of the commands.

The following orders

```
move 10
move 20
move 5
```

can be replaced by a single one:

```
move 35
```

Several following direction commands can be merged as the last command.

The following sequence

```
dir #east
dir #south
dir #north
```

is optimized as

```
dir #north
```

We suggest the following design. Introduce a message `aCommand mergeWith: anotherCommand` that returns a list containing the situation after trying to merge:

- When two commands can merge, returns a list containing the command resulting from the merge.
- When two commands do not merge, returns a list containing the two original commands.

You can use double dispatch to determine how commands of different classes are merged. As a default you can decide that different commands do not merge.

```
RbsRobotTest >> testMergeMoveCommandsProducesTheSum

  | cmdList |
  cmdList := (RbsMoveCommand new distance: 10; yourself)
    mergeWith: (RbsMoveCommand new distance: 10; yourself).
  self assert: cmdList size equals: 1.
  self assert: cmdList first distance equals: 20.
```

```
RbsRobotTest >> testMergeUNmergeableCommandsBecauseDifferent

  | cmdList |
  cmdList := (RbsMoveCommand new distance: 10; yourself)
    mergeWith: (RbsDirectionCommand new direction: #east; yourself).
  self assert: cmdList size equals: 2.
  self assert: cmdList first distance equals: 10.
```

Once the merge semantics is in place you can use this logic to optimize full paths as illustrated by the following test. Pay attention because this is a bit tricky in particular since

```
mov 1
mov 2
mov 3
dir #east
```

leads to

```
mov 3
mov 3
dir #east
```

and then finally to

```
mov 6
dir #east
```

The following test should pass.

```
testOptimizeMergeThreeMovesAndOthers

  | rb b |
  rb := RbsRobot new.
  b := RbsBoard new.
  rb setBoard: b.
  rb x: 4 y: 1.
  rb optimizePath:
'mov 2
mov 3
mov 4
dir #east'.
  self
    assert: (rb path collect: [ :each | each printString ])
    equals: #( 'mov 9' 'dir #east')
```

Extensions

You can also add the fact that a mov 5 followed by a mov -5 does not produce any command. Returning an empty list should be managed.

1.13 Extensions

Here is a list of extensions:

- The robot should be able to pick an item.
- It can have a certain capacity and cannot carry too many items.

- Passing a symbol to the direction is bad because the script developer may mistype it and exposing the internal logic is a bad idea. Propose a solution.
- The definition of the new location of a robot is based on a boring conditional. Can you imagine a better way?

```
computeNewPosition: anInteger
  "Returns a point representing the location of the next move."
  ^ direction = #east
    ifTrue: [ self x + anInteger ]
    ifFalse: [ direction = #west
      ifTrue: [ self x - anInteger  ]
      ifFalse: [ direction = #north
          ifTrue: [ self y + anInteger ]
          ifFalse: [ self y - anInteger ].
        ]
      ]
```

To give you a hint, we could have a little hierarchy with direction and each direction would decide the new location when told to compute it.

```
East computeFor: 4@1 distance: 10
> 14@1
```

1.14 Conclusion

This micro project shows you that representing actions as objects lets us manipulate programs at the right level of abstractions. Functionality as undo, replay, or path optimizations are easier to develop using commands. In addition refraining from using conditions is interesting because it forces us to delegate responsibilities to the objects and this makes your design more modular.