Boris bikes

The goal of this lab is to introduce you to Object-Oriented Programming (OOP) and show you the very basics of Object-Oriented Design (OOD).

Overview

You will be creating a simple system for managing Boris bikes in London. Our system will keep track of a number of docking stations and the bikes. It will allow you to rent the bikes and return them. Occasionally the bikes will break and then they will be unavailable for rental. There will also be a garage to fix the bikes and a van to move the bikes between the stations and the garage.

A system like this (in a much more complex form) is actually used by the real Boris bikes system in London. Every time you rent a bike or return it some objects get created somewhere in the system that tracks the usage of all bikes.

Prerequisites

This tutorial will assume that you are familiar with using objects in Ruby and the basics of OOP, as well as TDD.

How to use this tutorial

This tutorial is meant to show you the mental process you need to go through to build this system as well as practical steps required. However, **do not copy paste any code!** Type everything in to make sure you better understand what's going on. Even though the steps are provided here, don't follow them blindly. Use this tutorial to understand how you need to build it and then make sure you can build this system from scratch without looking at this tutorial.

The tutorial will be spoonfeeding you at the very beginning to make sure you understand the process we're going through but later we'll pick up the pace and you'll be expected to have learned the techniques we discussed in detail at the very beginning.

One of the goals of this tutorial is to show you not only the steps necessary to build the software but also to show you the correct way of doing it. After all, writing code is easy. Writing good code is where the challenge lies.

Building a domain model

The first step of the process (even before we think about a single line of code) is to create the domain model. The domain model describes key concepts in the domain and the relationships between them. In other words, it helps us understand what classes we may have and what methods they may implement.

Remember that a *Class* is a collection of data and related methods that operate on this data. In order to build a domain model we need to understand what data we'll have in the system and what operations will be performed on that data.

The first step in creating the domain model is to describe the problem in plain English.

We are building a system that manages bikes that can be rented by users from docking stations and returned there at the end of the rental. The bikes can break while being used, in which case they will not be available for rental after they are returned. There is a garage that can fix broken bikes. A van is used to move broken bikes from the stations to the garage. It can also be used to take fixed bikes back to the station(s). The van, all stations and the garage have fixed capacity, so they cannot take more bikes that they can hold.

The easy way to start thinking about what classes you may have is to take a look at all nouns in the text. They are prime candidates for classes.

We are building a system that manages **bikes** that can be rented by **users** from **docking stations** and returned there at the end of the **rental**. The **bikes** can break while being used, in which case they will not be available for **rental** after they are returned. There is a **garage** that can fix broken **bikes**. A **van** is used to move broken **bikes** from the **stations** to the **garage**. It can also be used to take fixed **bikes** back to the **station(s)**. The **van**, all **stations** and the **garage** have fixed capacity, so they cannot take more **bikes** that they can hold.

Not every noun will become a class and not every class we'll create will correspond to a noun in a plain English description. However, it's a good starting point. All data we'll have in the system will somehow be related to some class. For example, a number of bikes in a certain station is related to the station itself, not to a van or garage.

Our classes will implement a number of methods. To get an idea of what they may be, we can take a look at the verbs in our description.

We are building a system that manages bikes that can be **rented** by users from docking stations and **returned** there at the end of the rental. The bikes can **break** while being **used**, in which case they will not be available for rental after they are **returned**. There is a garage that can **fix** broken bikes. A van is used to **move** broken bikes from the stations to the garage. It can also be used to **take** fixed bikes back to the station(s). The van, all stations and the garage have fixed capacity, so they cannot **take** more bikes that they can **hold**.

Again, not all verbs will become methods and we'll have methods that don't correspond to any verbs in this description but that's ok. Right now we just need a idea of what they could be.

Setting up the project

It's time to write some code. The first step should be describing what you will build using tests. Create a new repository in your projects folder and add a `README.md` file to the repository.

By now you must have rspec installed. If you don't, install it now.

```
gem install rspec
```

Once it's installed, let's create a folder for our tests. By convention it is called "spec" if we're using rspec.

```
mkdir spec
```

You can create an .rspec file in your home directory default options for rspec. For example if you want rspec's output to be colored:

```
echo "-fn --color" > ./.rspec
```

Now everytime that you call in `rspec` on the command line the options `-fn` and `--color` will be passed in automatically outputting a beautiful colored test.

The creation of the spec directory, together with a spec_helper.rb file (feel free to create it to put all the requires there) that runs before each of your specs, and a .rspec file with sensible defaults can all be automated with the command: `rspec --init`. However, it's advisable to do it a few times manually before automating the process to better understand how all the pieces fit together.

Let's also create a directory "lib" in the root folder of the project. We'll put our classes in there when we create them.

Building the bike

We cannot possibly implement all functionality at once. We need to start somewhere and then increase the number of features until we are happy. What is the absolute minimum we could implement? What is central to the entire system? A bike.

How do we model a bike? We need to distill everything that could be said about the bike to the most essential information about it that is relevant to the system we are building.

A bike has a great many properties. It has a number of gears, it has a certain weight, etc. but all that is not relevant for the task of renting them from our system. Therefore, we shouldn't be concerned with those properties. The only thing about the bike that concerns us right now is whether it's broken or not. Our bikes can be broken but we don't care about anything else at the moment. This is not the only possible starting point but it's a good start.

Let's write a test for the bike that describes how we believe the bike should behave. We're intentionally keeping everything very simple right now. It's not because this is a tutorial for beginners but because it's a good thing to do. As you design the system, you should move in small steps regardless of your experience.

Create a file `spec/bike_spec.rb`. We'll put our bike tests there. The name should end in `_spec` since this is the convention rspec uses. The first part of the filename is the name of the class. So we have just implied that the class we'll be testing will be called `Bike`. After you have created the file, open it in Sublime.

Now let's write our first test in `spec/bike_spec.rb`.

```
# we're describing the functionality of a specific class, Bike
describe Bike do
  # this is a specific feature (behaviour)
  # that we expect to be present
  it "should not be broken after we create it" do
    bike = Bike.new # initialise a new object of Bike class
    # expect an instance of this class to have
    # a method "broken?" that should return false
    expect(bike).not_to be_broken
  end
end
```

So, we begin by writing a test that describes what we want to see happening. We tell rspec that we are testing the class Bike (line 2) and we want our bike to have a single feature (line 8): it should not be broken after it's created (line 9).

The line 9 is quite complex. Let's break it down.

The expect() method is provided by rspec. It takes the object under the test as an argument and returns a special object that has different methods that make the test pass or fail, depending on their arguments. One of these methods is not*to() that takes yet another rspec method be*broken() as an argument. Then the not*to() method takes a look at its argument and realises that if we don't expect the bike to "be*broken", we must have a method broken?() in the Bike class. So it calls this method on the bike object that we passed to the expect() method and if it returns true, fails the test because we expect it "not_to" be true.

The paragraph above is still complicated. Read through it several times but if it still seems unclear, don't worry, it's a bit advanced for now. I still want to explain it so that you know that this line of code is some crazy magic. Once you get more experience with rspec, you'll get used to how it works.

Why does it have to be so complicated? Because rspec is designed to write tests that read like English:

```
# Expect bike not to be broken
expect(bike).not_to be_broken
```

In order to achieve this readability rspec goes to great lengths doing all those crazy things described above. That's why it's complicated.

Another way to rewrite this line to make it clearer is

```
# Expect the method broken? of the object 'bike' to return false
expect(bike.broken?).to be false
```

So, now that you have an idea of what this test does, let's run it. It will fail.

What you see here is called a "stack trace". We need to learn how to read it to understand where exactly the problem lies. If you can't read the stack trace, you won't be able to pinpoint and fix it.

The first line is the most important one. It explains what the problem was and where it occurred.

In this case, the problem is in the file `boris-bikes/spec/bike_spec.rb` on line 2. The "" means that the code causing trouble is not part of any specific method. The error that occurred is of the type NameError and it's human-readable explanation is `uninitialized constant Bike`.

The `uninitialized constant Bike` means that Ruby doesn't know what `Bike` is.

Now, stop for a second and think about what could be the reason for the error. The answer may be obvious because this particular problem is so simple but it's important to take this step and ask yourself the question every time you see an error. Don't assume the first thing that comes to mind. Think about what is the most likely reason Ruby doesn't know about `Bike`.

Other lines in the output show the path in the code Ruby went through before encountering the error on line 5 of `bike_spec.rb`. The second line was executed right before the line 5 was executed.

So, getting back to our error. If you guessed that `uninitialized constant Bike` error happened because we never defined what Bike is, you're correct. It was simple in this case but it will be less trivial as we encounter more complex bugs later.

Let's define the bike. Create `lib/bike.rb` and define an empty `Bike` class.

The name of the file is `bike.rb` for a reason. If we're defining a class `Bike`, we should put it in `bike.rb`. It's not required, strictly speaking but it's a really good idea to put only one class in a `.rb` file and name it accordingly.

If you run the test right now, will it pass? Again, stop for a second and think before doing it. Right now you're predicting the result of a small experiment. If you have an expectation of whether it will pass or not, you will know whether everything is going according to your understanding of the program or not. So, will it pass or not and why? Then run it.

You'll find out it doesn't. Same error again. Ruby still has no idea about the `Bike` class. But why, when we've just defined it in `bike.rb`?

Think about it from Ruby's perspective. There are lots and lots of ruby files on your computer, defining hundreds if not thousands of different classes. Should all of them be available in every other Ruby file? Probably not. It's the programmer's responsibility to decide what should be available to Ruby code at what point. In other words, we haven't established any link between `bike_spec.rb` and `bike.rb`. That they are in the same repo or adjacent directories matters very little to Ruby. We need to explicitly link them together.

Add a "require" statement to the test.

```
# link to the Bike class
require_relative "../lib/bike"

# we're describing the functionality of a specific class, Bike
describe Bike do

  # this is a specific feature (behaviour) that we expect to be present
  it "should not be broken after we create it" do
    bike = Bike.new # initialise a new object of Bike class
    # expect an instance of this class to have a method "broken?" that should retu
rn false
    expect(bike.broken?).to be false
  end

end
```

Requiring a file is almost equivalent to just copy-pasting the contents of `bike.rb` on line 2 of the test from Ruby's perspective. However, since copy-pasting is a really bad idea, we require the file instead.

Let's break down the `require_relative` statement. We could use `require './lib/bike` which starts in the folder we run rspec and looks for a lib folder. Using `require_relative` instead tells ruby to start where the file we are writing `require_relative` is. Since its in the `boris_bikes/spec` directory we need to say, come up a level (the two dots), go in to the lib folder (`/lib/`) and this look for the file called `bike.rb`. We need this file to access the `Bike` class.

Now our test fails.

There's a difference between an error and a failure. An error happens when Ruby cannot run the test at all. For example, before we create a `Bike` class and required it, we couldn't run the test in principle – there was no class under test.

However, now the test is failing, which means that we can test the `Bike` class but it doesn't have the behaviour our test expects. Take a look at the list of failures in the output. There is only one: "Bike should not be broken after we create it". Where does the message come from? Look at the structure of the test.

```
describe Bike do
  it "should not be broken after we create it" do
    # the test goes here, omitted for brevity
  end
end
```

We describe `Bike` and we specify what it should do. Rspec simply concatenates `Bike` from the describe statement and the description of the feature from the `it` block.

So, what's the failure of the test?

```
Failure/Error: expect(bike.broken?).to be false
     NoMethodError:
       undefined method `broken?' for #<Bike:0x007f9ae30470b0>
     # ./spec/bike_spec.rb:14:in `block (2 levels) in <top (required)>'
```

First, it shows us the rspec expectation that failed. Specifically, it tells us that the method `broken?` is undefined. (`#<Bike:0x007f9ae30470b0>` refers to the instance of the `Bike` class that we have in the `bike` variable. The long number is the memory address).

So, the test is almost telling us what to do. We don't have the method `broken?` , so let's create one. Update the `Bike` class to include this method.

```
class Bike

  def broken?
  end

end
```

Note that the method is empty. Why? Because rspec didn't complain about anything else. It failed because there was no method? Let's do the minimum required to get over that failure: create the method. Now run the test again. Will it pass or fail? Why? Think about this question every time you run a test, don't just run it blindly.

So it passed. However, you may have expected it to fail. That would have been a reasonable assumption since there's no implementation of the method. However, in Ruby a `nil` value is treated as false if a boolean value is needed. Given that a method returns `nil` if nothing else is returned, an empty method always returns `nil` . So, by writing an empty method, we satisfy the test's expectation that the bike must not be broken.

Our code is still extremely basic but we're getting somewhere. This is a good time to commit the changes.

Making the `broken?` method work properly

You may feel suspicious that our test suite pass while we don't have much code just yet. You're right, we're missing something. We're missing more tests! Let's write another test to enable the bike to be broken. After all, if the bike has both states, fixed and broken, it's fair to assume we'll need a method to break a bike as well.

```
it "should be able to break" do
  bike = Bike.new
  bike.break
  expect(bike.broken?).to be true
end
```

So in our test we are creating a new bike, telling it to break itself and finally we expect it to be broken afterwards.

Before you run the test, ask yourself again. What do you expect to see? Will it pass? Will it fail? Why?

This is the test output:

```
1) Bike should be able to break
     Failure/Error: bike.break
     NoMethodError:
       undefined method `break' for #<Bike:0x007fe7a23fd7c0>
     # ./spec/bike_spec.rb:16:in `block (2 levels) in <top (required)>'
```

If we run it, we find out that it fails on `bike.break` statement because of `undefined method break`. Fair enough, let's add it to our `Bike` class. Again, rspec complained about the lack of method, so we're doing the absolute minimum to satisfy the test.

```
class Bike

  def broken?
  end

  def break
  end

end
```

Will the test pass this time? If not, will the error be different? If yes, what will it be?

Ok, the test is telling us that we expect the bike to be broken but it isn't: the `broken?` method returns nil while we expect it to return true. What does it tell us to do? If we just make it return true, the test will pass but then the first test will fail because it only works because `broken?` is returning `nil` at the moment.

Time to write some real code. Now we have two tests that force us to write some logic. Apparently, our bike should maintain some internal state that should be changed when we break it.

Let's introduce an instance variable that holds this information. This must be an instance variable because this data is applicable only to a specific instance of the `Bike` class. One bike (instance of `Bike`) may be broken, whereas another one may not be. So we need an instance variable to save it.

```
class Bike

  # the initialize method is always called when you create a new
  # class by typing Bike.new
  def initialize
    # all instance variables begin with "@"
    # this must be an instance variable because we'll need it
    # in other methods
    @broken = false
  end

  def broken?
    # instance variables are accessible in all methods
    @broken
  end

  def break
    # and any instance method can update them
    @broken = true
  end

end
```

Now our tests pass.

Now that our tests pass it's a perfect time to commit our changes.

Fixing the bike

If a bike can be broken, it can be fixed too, right? Sounds like a test.

```
it "should be able to get fixed" do
  bike = Bike.new
  bike.break
  bike.fix
  expect(bike.broken?).not_to be true
end
```

Why don't we assert that it's broken after we break but before we fix it? Also, why don't we assert that it's not broken before we break it? Because these cases are covered by other tests. We don't want to test the same thing again, it would be a waste of time. Other tests make sure that a new bike is not broken and, after we break it, it is broken.

If we run the test, we'll find that the method `fix` is undefined.

```
1) Bike should be able to get fixed
     Failure/Error: bike.fix
     NoMethodError:
       undefined method `fix' for #<Bike:0x007fe9ed903010 @broken=true>
     # ./spec/bike_spec.rb:23:in `block (2 levels) in <top (required)>'
```

Easy enough to fix it (HAHA). Add the method to the class.

```
def fix
end
```

Again, don't write the implementation even though it may be obvious to you. Run the test again. What message do you expect to see this time?

```
1) Bike should be able to get fixed
     Failure/Error: expect(bike.broken?).not_to be true
       expected broken? to return false, got true
     # ./spec/bike_spec.rb:24:in `block (2 levels) in <top (required)>'
```

So, the test tells us that the bike should not be broken after we fix it but it actually is. Now it's a good time to implement the `fix` method.

```
def fix
  @broken = false
end
```

Do you expect the tests to pass? Check it to make sure they do.

Now it's a good time to check in the code.

Refactoring

Let's pause for a while before continuing implementing more features. Is our code really good? Is there any chance to refactor it?

Refactoring is the process of changing the structure of the code in order to make it more readable, maintanable or extendable without adding any new functionality.

Software developers, even experienced ones, rarely write perfect code on their first attempt. First you write the code that works, then you refactor it to make it good. Let's refactor the code we have so far.

Before starting refactoring, always make sure the tests pass. If your tests fail or if you have no tests, you have no idea whether your refactored code is doing the same thing as before. Just hoping that it does is not enough: some changes in functionality are subtle and non-obvious even to experienced developers and may produce ripple effects. So, make sure the tests pass before you begin.

Take a look at the `Bike` class. What is the most obvious thing that is wrong here? Think for yourself before reading the answer below.

```
class Bike

  def initialize
    @broken = false
  end

  def broken?
    @broken
  end

  def break
    @broken = true
  end

  def fix
    @broken = false
  end

end
```

One of the problems is the code repetition. Lines 4 and 16 are the same. Code repetition violates the DRY principle: Don't Repeat Yourself. Repeating anything in the code or system design is a likely source of nasty bugs.

Instead of setting the instance variable in the initialiser, let's call the `fix` method instead.

```
def initialize
  fix
end
```

Now run your tests to see if they pass. They should. This is how you know that the refactoring was successful: it didn't change the functionality while reducing the repetion in the code.

Now take a look at the tests. What could be improved here?

```
require './lib/bike'

describe Bike do

  it "should not be broken after we create it" do
    bike = Bike.new
    expect(bike.broken?).to be false
  end

  it "should be able to break" do
    bike = Bike.new
    bike.break
    expect(bike.broken).to be true
  end

  it "should be able to get fixed" do
    bike = Bike.new
    bike.break
    bike.fix
    expect(bike.broken?).to be false
  end

end
```

It should be obvious by now that lines 6, 11 and 17 are crying to be refactored. We're doing the same operation at the start of every test. Fortunately, rspec provides a way to define variables that are used by all tests. Try this:

```
require './lib/bike'

describe Bike do

  let(:bike) { Bike.new }

  it "should not be broken after we create it" do
    expect(bike.broken?).to be false
  end

  it "should be able to break" do
    bike.break
    expect(bike.broken).to be true
  end

  it "should be able to get fixed" do
    bike.break
    bike.fix
    expect(bike.broken?).to be false
  end

end
```

Line 5 calls an rspec helper `let()` that defines a local variable `bike` using the block provided `({ Bike.new })` before every test. This makes our tests more DRY. Run them to see if they still work. They should.

Even though there are other repeated lines (8 and 19), it doesn't make sense to extract them into their own methods for readability's sake. However, in some cases it may be better to extract them: always use your best judgement. The code should look and feel elegant to you.

Now that we've finished with refactoring, it's a good time to push to Github.

Docking station

If the bike is the only thing we've got, our system isn't going to be very functional. At the very least we'll need docking stations. Let's think about the functionality we need for the docking station.

A docking station must be able to accept a bike and to release it. It should also have some capacity limit, because we shouldn't be able to put an infinite number of bikes into it. If a bike is broken, the docking station should not release it (or, rather, it should only release it to the van but we haven't got a van yet). Finally, a station must know how many bikes it has.

Let's start with the most basic functionality: accepting the bikes. As usual, we need to start with the test describing what's going on. Put your test in `spec/docking_station_spec.rb`.

```
describe DockingStation do

  it "should accept a bike" do
    bike = Bike.new
    station = DockingStation.new
    # we expect the station to have 0 bikes
    expect(station.bike_count).to eq 0
    # let's dock a bike into the station
    station.dock(bike)
    # now we expect the station to have 1 bike
    expect(station.bike_count).to eq 1
  end

end
```

Why did we choose to name the method that puts a bike into the station `dock`. Naming things is one of two hardest problems in Computer Science. We could have chosen a different name but this one seems good enough.

Now let's run it. We'll get an error straight away.

```
docking_station_spec.rb:1:in `<top (required)>': uninitialized constant DockingSta
tion (NameError)
```

Unless the reason for this error is immediately obvious, stop and think for a second about why it happened. Why would uninitialised constant (whatever this means) be a problem? Why would Ruby expect `DockingStation` to be initialised? Which line in the file raises the error? Why would line one be a problem?

It looks like we are referencing the class `DockingStation` on line one but since we never defined it, Ruby has no idea what `DockingStation` is. This is obviously a problem because in our test we're instantiating the class and calling its methods. So Ruby definitely needs it. However, it doesn't exist yet. Let's create it.

Create an empty class in `lib/docking_station.rb`

```
class DockingStation
end
```

If you run the test now, you'll get the same an error again. Again, unless it's immediately obvious, think about what's going on. We created the class but Ruby can't link the test to the class definition because we never *required* the file. Remember, we had exactly the same problem with the `Bike` class? Fix the problem by requiring the class file just like we've done it in the bike test (I'm omitting the exact line of code as an tiny exercise for you).

If you've done it correctly, you should now see a test failure.

```
  1) DockingStation should accept a bike
     Failure/Error: expect(station.bike_count).to eq 0
     NoMethodError:
       undefined method `bike_count' for #<DockingStation:0x007fcab45c3678>
     # ./spec/docking_station_spec.rb:9:in `block (2 levels) in <top (required)>'
```

Why do we get `undefined method 'bike_count'`? Because we haven't written it. Create an empty `bike_count` method in the `DockingStation` class. Don't put any implementation because the test isn't asking us to do it yet. Let the test drive your code.

If your method is empty, the next error you should see is

```
1) DockingStation should accept a bike
     Failure/Error: expect(station.bike_count).to eq 0

       expected: 0
            got: nil

       (compared using ==)
     # ./spec/docking_station_spec.rb:9:in `block (2 levels) in <top (required)>'
```

Can you explain why this is happening? An empty method always return `nil`, whereas we expect the `bike_count` to be `0` in our tests. Let's update the method to return `0`.

This may sound unnecessary. Why would you make the method return `0` if it's an obviously incorrect implementation that will be changed before the next commit? The reason we do this is that this approach forces us to write the absolute minimum necessary to make the test pass. It also ensures that we don't write the code that's not covered by the tests. Let the tests drive your code.

In fact, please forgive me for digressing, I have yet to meet a developer who would use TDD really well, writing good tests and letting them to drive the code, who would be a bad coder. Quite the opposite, I've seen many programmers who ignored tests because they were "slowing them down" only to produce a piece of unmaintainable code that would be a pain to work with.

You can argue that it's possible to take the tests too far, testing absolutely every possible scenario and not writing a single line without a test telling you to do it. Use your best judgement. If it were possible to tell exactly when and how to write tests, we'd have computers writing them for us. TDD isn't a silver bullet but it's a very powerful weapon in your arsenal. Let the tests drive your code. Trust the tests.

So, our `DockingStation` isn't terribly useful right now.

```
class DockingStation

  def bike_count
    0
  end

end
```

However, it gets us to the next error.

```
  1) DockingStation should accept a bike
     Failure/Error: station.dock(bike)
     NoMethodError:
       undefined method `dock' for #<DockingStation:0x007fd369c95bc8>
     # ./spec/docking_station_spec.rb:11:in `block (2 levels) in <top (required)>'
```

By now you know that you need to create the method `dock` and rerun the test. Don't forget that the method `dock` takes an argument, a bike. After you do it, you'll get to the error that finally forces you to write some real code:

```
  1) DockingStation should accept a bike
     Failure/Error: expect(station.bike_count).to eq(1)

       expected: 1
            got: 0

       (compared using ==)
     # ./spec/docking_station_spec.rb:13:in `block (2 levels) in <top (required)>'
```

Now we can't just make the method return 1 to get rid of this error because that would make the test break on a line above. So we need to write some real code.

Our *docking station* needs to know how many *bikes* it's got. In principle, we could get away with creating a counter that just increments every time a bike is docked without actually storing a reference to the bike. The test would still pass. However, there's a difference between making a test return a constant instead of a real implementation and writing an obviously incorrect implementation that we'll need to change later. In this particular case, I choose to actually store the bike inside a station instead of just pretending to do it.

We need to retain a reference to the bike inside the station. The best place for this would be an instance variable. Let's create an array of bikes and return its size and the number of bikes in the station.

```
class DockingStation

  def initialize
    @bikes = []
  end

  def bike_count
    @bikes.count
  end

  def dock(bike)
  end

end
```

Let's now rerun the test to makes sure the error is still the same. Since the array is empty, the `bike_count` method should return `0`. The reason we're running the test now to see the same error is to make sure that we didn't introduce any other problems along the way.

However, to make this error go away, we need to write a real implementation of the `dock` method.

```
def dock(bike)
  @bikes << bike
end
```

The `<<` is called a shovel operator. It puts an element in the array. Now our `bike_count` method will return the correct value after we dock a bike. All tests should pass.

Now that our tests pass it's a good time to check the code in.

If our docking station can accept bikes, it will need to release them at some point. Let's write a test to release a bike.

```
it "should release a bike" do
  station.dock(bike)
  station.release(bike)
  expect(station.bike_count).to eq 0
end
```

To get rid of code repetition, I also put a couple `let()` statements at the very beginning of the "describe" block, just like we've done earlier in the `Bike` test.

```
let(:bike) { Bike.new }
let(:station) { DockingStation.new }
```

Run the test, create an empty `release` method, run the test again. Now the reason for the failure is that the method doesn't work. Let's implement it.

```
def release(bike)
  @bikes.delete(bike)
end
```

Now all tests should pass. Great news: we can now dock bikes and release them! Let's check in the code.

Let's now make our docking stations more realistic. Right now you can dock any number of bikes in there and it'll be ok as long as you have available memory on your machine (that'll be many, many millions of bikes). Let's introduce some limit on the capacity, set when the station is being initialised. As always, first comes the test.

```
it "should know when it's full" do
  expect(station.full?).not_to be true
  20.times { station.dock(Bike.new) }
  expect(station.full?).to be true
end
```

Why 20? Let's pass the capacity as a a parameter to the initialiser.

`let(:station) { DockingStation.new(:capacity => 20) }`

So we're initialising the station as the station that has the capacity of 20 and we're filling it with 20 bikes. We expect it to be full after that. Run the test (it will complain about the wrong number of arguments for the initialiser). Let's fix the problem the test has uncovered.

```
def initialize(options = {})
  @capacity = options.fetch(:capacity, DEFAULT_CAPACITY)
  @bikes = []
end
```

Let's stop and discuss this code. The first question is why pass the capacity as a key-value pair of an options hash. We could have passed it as the first argument:

```
def initialize(capacity=DEFAULT_CAPACITY)
```

However, then we'd have to use it like this:

```
DockingStation.new(20)
```

The problem with this code is that it doesn't communicate what 20 is. Is it the capacity? Do we want to create 20 docking stations in one go? Is it the number of bikes a station should be initialised with? Our code must communicate the intent well.

The second question is what this line does:

```
@capacity = options.fetch(:capacity, DEFAULT_CAPACITY)
```

This is a common pattern for passing optional arguments into Ruby. The `Hash.fetch` method retrieves the value for the given key (`options[:capacity]` in this case) and if the key is not found, it returns the second argument (`DEFAULT_CAPACITY`). So if the capacity is passed, it's used, otherwise the default one will be assigned.

Finally, you'll need to define the default capacity at the top of the `DockingStation` class.

```
class DockingStation
  DEFAULT_CAPACITY = 10
```

There's a different way of achieving the same effect: named arguments in Ruby 2.0. However, since this is a relatively new feature, you're likely to see the pattern we're using in the real world.

Let's run the test again.

Undefined method `full?` . That's because we don't have it. Create it but leave it empty. Next error:

```
  1) DockingStation should know when it's full
     Failure/Error: expect(station).to be_full
       expected full? to return true, got nil
```

Now let's implement it. What does it mean for a station to be `full?` ? To have the number of bikes to be equal to the capacity.

```
def full?
  bike_count == @capacity
end
```

Why are we using `bike_count` as opposed to `@bikes.count` ? Doing so would lead to repetition. If we have a method for giving us the bike count, we must use it.

Do you think the tests would pass now? Make a prediction, then run them. If they do, it's a good time to commit the code.

However, what happens if we try to dock the bike into a station that's full?

```
it "should not accept a bike if it's full" do
  20.times { station.dock(Bike.new) }
  expect { station.dock(bike) }.to raise_error RuntimeError
end
```

From now on I'll start skipping the test failures because you've seen quite of few of them now.

This test expects that docking a bike into a station should not raise an error. It will fail because this functionality isn't implemented yet. Let's update the `dock` method.

```
def dock(bike)
  # if the capacity is reached, raise an exception
  raise "Station is full" if full?
  @bikes << bike
end
```

Now our tests should pass but we have a long line of code in our tests that is repeated twice.

```
20.times { station.dock(Bike.new) }
```

Let's refactor the code by extracting the method to a helper method (put it inside the `describe DockingStation` block).

```
def fill_station(station)
  20.times { station.dock(Bike.new) }
end
```

Now our test look better.

```
it "should not accept a bike if it's full" do
  fill_station station
  expect { station.dock(bike) }.to raise_error RuntimeError
end
```

If everything passes, it's a good time to check everything in.

When you need to get a bike from a station, you need to know what bikes are available. Some bikes can be broken and they shouldn't be available for rental. Let's create a method that will return the list of bikes that are available.

```
it "should provide the list of available bikes" do
  working_bike, broken_bike = Bike.new, Bike.new
  broken_bike.break
  station.dock(working_bike)
  station.dock(broken_bike)
  expect(station.available_bikes).to eq([working_bike])
end
```

The implementation is fairly simple.

```
def available_bikes
  @bikes.reject {|bike| bike.broken? }
end
```

Now that the tests pass, it's time to check the code in.

Our station is mostly done but there are a few more things, listed in the Exercises section below for you to finish.

Van

Before writing more code, let's discuss what the other classes are and why we need them, starting with the van.

The van is moving broken bikes from the stations to the garage. Once they are fixed, the van moves them back to the stations. So the van must be able to accept the bikes at the source and release them at the destination. Obviously, the van must have some limit on the capacity, just like the station.

However, the van isn't going to be very different from the station. It may have some additional methods to interact with the station and the garage but overall it's fairly similar to the station.

Garage

The situation with the garage is very similar. What's the difference between a garage and a docking station? Only a van is supposed to take bikes out of a garage. Also, the bikes get fixed once they get to the garage. Otherwise, the garage is not much different from the station.