Walkthrough - the Single Responsibility Principle

Back to the Challenge

We've done RED, we've done GREEN - we haven't done any refactoring recently, so now would be a good time to look for opportunities.

This is a great opportunity to introduce the Single Responsibility Principle (SRP). Everything in code should have a *single recognizable responsibility*. Our `DockingStation` class is responsible for docking and releasing bikes. While you might argue that that's *two* responsibilities(!), they are inextricably dependent on one-another, so it's OK. But we do separate those responsibilities into individual methods. Similarly, each method should have a *single responsibility*. In our code, the `dock` method is responsible for handling the docking of a bike:

```
fail 'Docking station full' if @bikes.count >= 20
@bikes << bike
```

and defining the rule for capacity:

```
@bikes.count >= 20
```

This is a bad thing for 2 reasons. Firstly, because it breaks the single responsibility principle, and secondly because it necessitates an additional cognitive step when reading the code. As a reader, I am forced to infer that `@bikes.count >= 20` corresonds to the docking station being full.

Let's fix this and improve the readability of our code by introducing private helper methods `empty?` and `full?`. In general we don't write unit tests for private methods.

> Why do you think that is?

```
class DockingStation
  def initialize
    @bikes = []
  end

  def release_bike
    fail 'No bikes available' if empty?
    @bikes.pop
  end

  def dock(bike)
    fail 'Docking station full' if full?
    @bikes << bike
  end
```

```
    private

    def full?
      @bikes.count >= 20
    end

    def empty?
      @bikes.empty?
    end
  end
```

Having made the above refactoring, you will of course want to immediately run `rspec` again to ensure that we haven't accidentally introduced any errors. And of course you'll want to manually test that everything still works in `irb`.

> Unless, of course, you've already created a `feature_spec.rb` file to automate your feature tests. Check Solution 12 for more on that.

Forward to the Challenge Map

Walkthrough - Removing Magic Numbers

Back to the Challenge

We really should also deal with the 'magic number' `20`. Magic numbers are a common source of bugs in computer programs. They occur wherever a numeric literal (e.g `20`) is used in code and is related to a domain concept; in this case, the default capacity of a docking station. In a large and complex program, if we were to see the literal `20` all over the place, it would not be obvious, without reading the context in which it is used, whether it is a reference to capacity or some other domain concept that happens to also be 20. What if the default capacity changes?

To deal with this, we *encapsulate* the literal in a *constant* then use this constant everywhere else:

```
class DockingStation
  DEFAULT_CAPACITY = 20

  # other code omitted for brevity

  def full?
    bikes.count >= DEFAULT_CAPACITY
  end
end
```

This is a good start, however is there anywhere else where we use the magic number `20`? How about in our tests? Here's a handy blog post on the subject of testing with magic numbers.

Remember this line in our tests?

```
# in docking_station_spec.rb
20.times { subject.dock Bike.new }
```

Let's refactor it like so:

```
DockingStation::DEFAULT_CAPACITY.times do
  subject.dock Bike.new
end
```

You can see we use `::` as a **Namespace operator**: we use it to access the `DEFAULT_CAPACITY` constant defined within the `DockingStation` class.

> What happens if we define `DEFAULT_CAPACITY` outside of the `DockingStation` class? Any reason that might be a bad idea?

Forward to the Challenge Map

Walkthrough - Initialization defaults

Back to the Challenge

We have a User Story relating to capacity:

```
As a system maintainer,
So that busy areas can be served more effectively,
I want to be able to specify a larger capacity when necessary.
```

It feels like we need a `capacity` attribute for our docking station. But we can't introduce one without a unit test:

```
describe DockingStation do
  # other tests omitted for brevity

  it 'has a default capacity' do
    expect(subject.capacity).to eq DockingStation::DEFAULT_CAPACITY
  end
end
```

See if you can make this test pass. Hint: use `attr_reader` to create the `capacity` method and use `initialize` to set its initial value. Don't forget to change the `full?` method to use our new `capacity`.

Actually, since docking station is already taking care of this, we can use the `capacity` that's already defined:

```
describe DockingStation do
  describe 'dock' do
    it 'raises an error when full' do
      subject.capacity.times { subject.dock Bike.new }
      expect { subject.dock Bike.new }.to raise_error 'Docking station full'
    end
  end
end
```

We might also use a *private* `attr_reader` to have all our references to the `@bikes` instance variable go through a single interface:

```
class DockingStation
  DEFAULT_CAPACITY = 20

  attr_reader :capacity

  def initialize
```

```
    @bikes = []
    @capacity = DEFAULT_CAPACITY
  end

  def dock(bike)
    fail 'Docking station full' if full?
    bikes << bike
  end

  def release_bike
    fail 'No bikes available' if empty?
    bikes.pop
  end

  private

  attr_reader :bikes

  def full?
    bikes.count >= capacity
  end

  def empty?
    bikes.empty?
  end
end
```

We prefer to reference our instance variable within our class via getter methods in order to DRY out the '@' symbols, and so also that if we need to make some consistent initialization or change to our instance variable, then we can do it in one place, rather than having to update a series of scattered references to instance variables throughout the class.

Going back to our User Story:

```
As a system maintainer,
So that busy areas can be served more effectively,
I want to be able to specify a larger capacity when necessary.
```

The key question is: `how will a new capacity be passed to the docking station?` Secondarily, `can docking stations change their capacity over time?` These are questions for the client. We can email them, but in the meantime what should we assume?

1. That docking station capacities can change over time, or
2. that they are fixed once?

Depending on what we assume, we might choose a different route to pass the new capacity to a docking station. If fixed once then we might well pass through an initialize method like so:

```
$ irb
2.1.5 :001 > require './lib/docking_station'
 => true
2.1.5 :002 > docking_station = DockingStation.new 50
  => #<DockingStation:0x007fa37eba3be0 @bikes=[], @capacity=50>
```

which would rely on an initialize method in our class

```
class DockingStation
  def initialize(capacity)
    @capacity = capacity
  end
end
```

however if docking station capacities can vary over their lifetime then perhaps we could create a writeable attribute like so:

```
class DockingStation
  attr_accessor :capacity
end
```

so that we can update the docking station capacity at will:

```
$ irb
2.1.5 :001 > require './lib/docking_station'
 => true
2.1.5 :002 > docking_station = DockingStation.new
=> #<DockingStation:0x007fa37eb90338 @bikes=[], @capacity=20>
2.1.5 :003 > docking_station.capacity = 25
=> #<DockingStation:0x007fa37eb90338 @bikes=[], @capacity=25>
2.1.5 :004 > docking_station.capacity = 55
=> #<DockingStation:0x007fa37eb90338 @bikes=[], @capacity=55>
```

In the absence of information from the client, let's assume capacities are only declared at initialization. So, using the above feature test, we need something like this at the Unit Test level:

```
# in docking_station_spec.rb
describe 'initialization' do
  it 'has a variable capacity' do
    docking_station = DockingStation.new(50)
    50.times { docking_station.dock Bike.new }
    expect{ docking_station.dock Bike.new }.to raise_error 'Docking station full'
  end
end
```

We can DRY this up later. Let's make this pass:

```
# in docking_station.rb
class DockingStation
  def initialize(capacity)
    @capacity = capacity
    @bikes = []
  end
end
```

OK! Now for default capacity at instantiation. No feature test needed for this one, as we already have one. So, the Unit Test is pretty much the same as the one we just wrote, but with our default capacity instead. Let's write a slightly DRYer version of the above:

```
# in docking_station_spec.rb
describe 'initialization' do
  subject { DockingStation.new }
  let(:bike) { Bike.new }
  it 'defaults capacity' do
    described_class::DEFAULT_CAPACITY.times do
      subject.dock(bike)
    end
    expect{ subject.dock(bike) }.to raise_error 'Docking station full'
  end
end
```

Let's implement the failing functionality with a **default argument value**:

```
# in docking_station.rb
class DockingStation
  # See how we use a default value
  # in the parameter
  def initialize(capacity=DEFAULT_CAPACITY)
    @capacity = capacity
    @bikes = []
  end
end
```

OK, we're good! All our feature tests pass, as do our Unit Tests. Let's proceed.

Forward to the Challenge Map

Walkthrough - dealing with broken bikes

Back to the Challenge

So, we have these User Stories to implement:

```
As a member of the public,
So that I reduce the chance of getting a broken bike in future,
I'd like to report a bike as broken when I return it.

As a maintainer of the system,
So that I can manage broken bikes and not disappoint users,
I'd like docking stations not to release broken bikes.

As a maintainer of the system,
So that I can manage broken bikes and not disappoint users,
I'd like docking stations to accept returning bikes (broken or not).
```

Let's start with the first one. Imagine how our manual feature test would look:

```
$ irb
2.0.0-p195 :001 > station = DockingStation.new
 => #<DockingStation:0x007fae7b3b8950>
2.0.0-p195 :002 > bike = Bike.new
 => #<Bike:0x007fae7b3c0dd0>
2.0.0-p195 :003 > bike.report_broken
 => true
2.0.0-p195 :004 > bike.broken?
 => true
2.0.0-p195 :005 > station.dock bike
 => #<Bike:2fa237227f @broken=true>
```

We should be getting used to these sorts of error messages by now:

```
2.0.0-p195 :003 > bike.report_broken
NoMethodError: undefined method `report_broken' for #<Bike:0x007fad7c030ee0>
  from (irb):8
  from /Users/tansaku/.rvm/rubies/ruby-2.2.2/bin/irb:11:in `<main>'
```

Let's implement the Unit Test for the `report_broken` method we're missing, in the `bike_spec.rb` file.

```
# in bike_spec.rb
it 'can be reported broken' do
```

```
  subject.report_broken
  # let's use one of RSpec's predicate matchers
  expect(subject).to be_broken
end
```

We now have a failing unit test to match our manual feature test error:

```
Failures:

  1) Bike can be reported broken
     Failure/Error: subject.report_broken
     NoMethodError:
       undefined method `report_broken' for #<Bike:0x007fe0b9315a50>
     # ./spec/bike_spec.rb:7:in `block (2 levels) in <top (required)>'
```

So, our implementation:

```
# in bike.rb
class Bike
  def report_broken
    @broken = true
  end

  def broken?
    @broken
  end
end
```

All our specs are passing.

So, on to the next feature:

```
As a maintainer of the system,
So that I can manage broken bikes and not disappoint users,
I'd like docking stations not to release broken bikes.
```

Let's draft a feature test using `irb`:

```
$ irb
2.0.0-p195 :001 > station = DockingStation.new
 => #<DockingStation:0x007fae7b3b8950>
2.0.0-p195 :002 > bike = Bike.new
 => #<Bike:0x007fae7b3c0dd0>
2.0.0-p195 :003 > bike.report_broken
 => true
```

```
2.0.0-p195 :004 > station.dock bike
 => #<Bike:2fa237227f @broken=true>
2.0.0-p195 :005 > station.release_bike
 RuntimeError: No bikes available
  .... stack trace omitted ....
```

But at the moment our docking station will happily release a broken bike. We have to put a stop to that.

At this point, we're going to pull back on the solution support: it's up to you to implement the last two features, using your newfound testing skills.

Don't forget you can rely on others in your cohort and Slack Overflow for support. When we start hitting new topics again, the solutions will kick back into action.

Back to the Challenge Map

Walkthrough - isolating tests with doubles

Back to the Challenge

Let's look at the Unit Tests for `DockingStation`:

```
require 'docking_station'

describe DockingStation do
  it { is_expected.to respond_to :release_bike }

  it 'releases working bikes' do
    subject.dock Bike.new
    bike = subject.release_bike
    expect(bike).to be_working
  end

  it 'does not release broken bikes' do
    bike = Bike.new
    bike.report_broken
    subject.dock bike
    expect {subject.release_bike}.to raise_error 'No bikes available'
  end

  it { is_expected.to respond_to(:dock).with(1).argument }

  describe 'release_bike' do
    it 'raises an error when there are no bikes available' do
      expect { subject.release_bike }.to raise_error 'No bikes available'
    end
  end

  describe 'dock' do
    it 'raises an error when full' do
      subject.capacity.times { subject.dock Bike.new }
      expect { subject.dock Bike.new }.to raise_error 'Docking station full'
    end
  end
end
```

The purpose of a unit test is to exhaustively test a single component. In this case `DockingStation`. But can you spot a potential problem?

Our test is not *isolated*. It is dependent on another component of the system - `Bike` and therefore could be unexpectedly affected by changes to `Bike`. We cannot say our unit test is truly a unit test if its outcome is influenced by changes in other components.

So how do we overcome this? This is where we introduce *doubles*. A *double* is a temporary, disposable object that we can use as a stand-in for some other real object - like a `Bike` for example. The difference is, we can

precisely define the behaviour of a double on a test-by-test basis to remove any uncertainty that the real object might introduce.

Let's zoom in on our `dock` test:

```
describe 'dock' do
  it 'raises an error when full' do
    subject.capacity.times { subject.dock Bike.new }
    expect { subject.dock Bike.new }.to raise_error 'Docking station full'
  end
end
```

We are actually creating 21 instances of `Bike` in this test. However, the test is not interested in the bikes, just that the docking station is full. It could be full of pogo sticks, or strings, for all we care. So rather than introduce the dependency on `Bike`, let's try using a String:

```
describe 'dock' do
  it 'raises an error when full' do
    subject.capacity.times { subject.dock "I'm a string, not a bike" }
    expect { subject.dock :bike }.to raise_error 'Docking station full'
  end
end
```

Instead of docking an instance of `Bike`, we are simply docking the string `"I'm a string, not a bike` instead. It doesn't matter what we put in this String: we're just proving that this test will pass even if we don't give a `Bike` instance to the `DockingStation`. You can dock `20` of *anything*: once it's taken 20 objects, it will raise the error.

Passing `String` instances instead of `Bike` instances might suffice in the short term, but it's not a very robust solution. If the docking station needs to interact with methods of a `Bike` object - like `working?`, then a `String` is no good: try calling `"Hello World".working?` in `irb`. See? `NoMethodError`. Strings aren't going to cut it when we need to query other classes within our Unit Tests.

Fortunately, RSpec provides a sophisticated framework for creating test doubles using the method `double(:class)`:

```
describe 'dock' do
  it 'raises an error when full' do
    subject.capacity.times { subject.dock double :bike }
    expect { subject.dock double(:bike) }.to raise_error 'Docking station full'
  end
end
```

As we'll see in the next challenge, we can add behaviour to these `double`s when we create them.

> A little secret: it doesn't matter what symbol you use when creating a double using `double :symbol`. However, we tend to use the name of the doubled class in the symbol. Why might it be advantageous for future developers to see the name of the class we're doubling?

Back to the Challenge Map

Walkthrough - Mocking Behaviour on Doubles

Back to the Challenge

Our tests are failing, because our `double`s cannot respond to the methods we need them to.

Take a look at the following test. In order to test that a broken bike is not released, we have to create a bike and then report it as broken:

```
it 'does not release broken bikes' do
  bike = Bike.new
  bike.report_broken
  subject.dock bike
  expect {subject.release_bike}.to raise_error 'No bikes available'
end
```

This is annoying as we're not interested in testing the bike itself, only that the docking station contains a broken bike and doesn't release it.

What we actually want to test is that `DockingStation` does not release a bike when the bike's `broken?` method is 'truthy' (or it's `working?` method is 'falsy' depending on how you've implemented the feature in your DockingStation). So what we need is a double that returns `true` when `broken?` is called or `false` when `working?` is called. We can do this using **method stubs**. Take a look at the following code:

```
it 'does not release broken bikes' do
  bike = double(:bike)
  allow(bike).to receive(:broken?).and_return(true)
  subject.dock bike
  expect {subject.release_bike}.to raise_error 'No bikes available'
end
```

The additional line, starting with `allow`, simply tells our double to respond to a `broken?` method and return `true`.

If we know the method double should stub at the point we create the double, we can use a shorthand syntax:

```
# this is the same as the test above
it 'does not release broken bikes' do
  # see how we move the allow() statement
  # into the double creation statement
  bike = double(:bike, broken?: true)
  subject.dock bike
  expect {subject.release_bike}.to raise_error 'No bikes available'
end
```

We can mock behaviour on most of our doubles this way.

Sometimes, mocking behaviour will reveal a weakness in our design. This is a great advantage to thoroughly isolating tests.

Let's take a look at one of the last remaining reference to `Bike` in our docking station unit tests:

```
# in docking_station_spec.rb
it 'releases working bikes' do
  subject.dock Bike.new
  bike = subject.release_bike
  expect(bike).to be_working
end
```

Bizarrely, although this is a `DockingStation` unit test, our expectation is on `bike`! Because we wrote the test against the actual `Bike` class, it wasn't immediately obvious that this is a poor test. However, when we replace it with a double, it becomes more obvious:

```
it 'releases working bikes' do
  subject.dock double(:bike, broken?: false)
  bike = subject.release_bike
  expect(bike).to be_working
end
```

We've explicitly defined a bike double that returns `false` when `broken?` is called because we used the `broken?` method in our `DockingStation` to test whether a bike can be released. You may have it the other way around (you might test `working?` instead). Either way, we will eventually expose a subtle problem.

```
Failures:

  1) DockingStation releases working bikes
     Failure/Error: expect(bike.working?).to be true
       Double :bike received unexpected message :working? with (no args)
     # ./spec/docking_station_spec.rb:9:in `block (2 levels) in <top (required)>'
```

Our test is failing because in our expectation, we are calling a method on our double that is not defined (`working?`). To pass it, we would have to do define the `working?` method too, like this:

```
subject.dock double :bike, broken?: false, working?: true
```

This mocking is getting ridiculous. Now we are testing that our double returns `true` for `working?`, and `false` for `broken?`. One of those pieces of information should be enough.

What we really want to test is that if there's one working bike in the docking station, then that same bike gets returned by `release_bike`. The *feature test* takes care of also testing that the bike that comes out is working.

Rewriting our unit test:

```
it 'releases working bikes' do
  bike = double(:bike, broken?: false)
  subject.dock bike
  expect(subject.release_bike).to be bike
end
```

Of course, we are introducing doubles retrospectively and normally we would use doubles from the very start, so issues like this would be less likely to arise.

> This particular style of testing - where every dependency is mocked out using doubles is called the 'London' style. There is another style called the 'Chicago' style. Research these two styles with your pair partner and discuss each approach. Which do you prefer? Why?

Back to the Challenge Map

Walkthrough - Men with Ven

Back to the Challenge

The relevant user stories here are:

```
As a maintainer of the system,
So that I can manage broken bikes and not disappoint users,
I'd like vans to take broken bikes from docking stations and deliver them to
garages to be fixed.

As a maintainer of the system,
So that I can manage broken bikes and not disappoint users,
I'd like vans to collect working bikes from garages and distribute them to docking
stations.
```

Let's follow the technique from the previous stages to design manual feature tests and then unit tests to drive the functionality of the Van and the Garage. **The user stories above have multiple clauses in their task components**; should we break them out into separate features?

Also, before writing more code, let's think about 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.

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.

This solution will not explicitly build the Van and Garage classes for you, as they are almost identical to the DockingStation class you already have.

Once you have implemented them, you will notice that you are duplicating a lot of code across classes. This is a great time to refactor our code using a *mixin*, which can share behaviour across classes. Let's do that now.

Forward to the Challenge Map

Walkthrough - `Module`s as mixins

Back to the Challenge

Let's start by looking at the shared behaviour of docking stations, vans and garages:

- They contain bikes
- They have a capacity
- You can put bikes in
- You can take bikes out

Each object might use a different language for these (.e.g. 'dock', 'load', 'release', 'unload' etc.) but the behaviour is essentially the same. Let's test-drive the creation of a new `module` that encapsulates this behaviour.

Note: A module cannot be instantiated in the way a class can. You can't have an instance of a module. In Ruby, modules are used to define common behaviour that can then be *included* into other objects. If we are defining common behaviour for a docking station, van and garage, it follows that this behaviour will need to be included into the `DockingStation`, `Van` and `Garage` classes. We don't want to have to write the same tests three times, so we will use the shared example feature of RSpec to write the tests once, then include those tests in the tests for docking station, van and garage.

Because we are testing a module, we won't have an actual object instance that we can test. So, while we are test-driving the module, we'll use a temporary class as something to include the module into, so it can be tested. Later, when we've test-driven the creation of this module, we could do away with the temporary class.

`spec/bike_container_test_spec.rb`:

```
# The purpose of this class is to give us an instance of
# something that includes `BikeContainer`, to initially run tests against
class BikeContainerTest
  include BikeContainer
end

# `it_behaves_like BikeContainer` imports all the tests in the shared example
describe BikeContainerTest do
  it_behaves_like BikeContainer
end
```

If you run RSpec now, it should fail with an error message like `Could not find shared examples "a bike container"`

So let's get on and define our shared examples. In `spec/support/shared_examples_for_bike_container.rb`:

```
shared_examples_for BikeContainer do
  it 'has a default capacity when initialized' do
```

```
    expect(subject.capacity).to eq BikeContainer::DEFAULT_CAPACITY
  end
end
```

Remember, we will need also to require this examples file in `bike_container_test_spec.rb`.

We've chosen the default capacity as the first behaviour to model as it feels like a suitable place to start. So how de we make this test pass? Let's create the module! `lib/bike_container.rb`:

```
module BikeContainer
  DEFAULT_CAPACITY = 20

  attr_reader :capacity

  def initialize
    @capacity = DEFAULT_CAPACITY
  end
end
```

Note that we've defined `DEFAULT_CAPACITY` in the `BikeContainer` module. That means `DockingStation`, `Van` and `Garage` will all have the same default capacity. This is probably not ideal, but we'll stick with it for now for simplicity.

Depending on how you implemented the variable capacity, you may need to specify the capacity at the initialize stage. Let's write a test for that:

```
shared_examples_for BikeContainer do
  # other tests omitted for brevity

  describe 'capacity' do
    it 'can be overridden on initialize' do
      random_capacity = Random.rand(100)
      subject = described_class.new random_capacity
      expect(subject.capacity).to eq random_capacity
    end
  end
```

Why did we use a random capacity? Discuss this idea with your pair partner - can you see any advantages and/or disadvantages with this approach? We'll make it pass:

```
module BikeContainer
  DEFAULT_CAPACITY = 20

  attr_reader :capacity

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

```
  end
end
```

Let's add another test to `shared_examples_for_bike_container.rb`:

```
shared_examples_for BikeContainer do
  # other tests omitted for brevity

  describe 'add_bike' do
    it 'receives a bike' do
      subject.add_bike double :bike
      expect(subject).not_to be_empty
    end
  end
end
```

Notice that we are not being driven by a feature test here. Our motivation is refactoring, so we are driven by a desire for clean code. Imagine every project has the unwritten user story:

```
As as conscientious software developer
So that my code is robust, readable and reusable
I want to refactor regularly
```

Let's speed things up a bit and add some related tests at the same time:

```
shared_examples_for BikeContainer do
  # other tests omitted for brevity

  describe 'add_bike' do
    it 'receives a bike' do
      subject.add_bike double :bike
      expect(subject).not_to be_empty
    end

    it 'raises an error when full' do
      subject.capacity.times { subject.add_bike double(:bike) }
      expect { subject.add_bike double(:bike) }.to raise_error "#
{described_class.name} full"
    end
  end
end
```

And let's make them pass:

```
module BikeContainer
  # other code omitted for brevity

  def initialize(capacity = DEFAULT_CAPACITY)
    @capacity = capacity
    @bikes = []
  end

  def add_bike(bike)
    raise "#{self.class.name} full" if full?
    bikes << bike
  end

  def empty?
    bikes.empty?
  end

  def full?
    bikes.count >= capacity
  end

  private

  attr_reader :bikes
end
```

Take some time to study the code above with your pair partner. What do you think `self.class.name` means? What is the class in this case? Why have we made the `bikes` attribute private?

Let's finish by enabling bikes to be taken out of a bike container:

```
shared_examples_for BikeContainer do
  # other tests omitted for brevity

  describe 'remove_bike' do
    let(:bike) { Bike.new }
    before(:each) { subject.add_bike bike }

    it 'returns a bike' do
      expect(subject.remove_bike).to eq bike
    end

    it 'removes the bike from the collection' do
      subject.remove_bike
      expect(subject).to be_empty
    end

    it 'raises an error when empty' do
      subject.remove_bike
      expect { subject.remove_bike }.to raise_error "#{described_class.name}
empty"
```

```
    end
  end
end
```

Take a good look through these tests. Notice how each test includes just one expectation and tests a very specific part of the `remove_bike` behaviour. You could imagine combining all of these into one test:

```
shared_examples_for BikeContainer do
  # other tests omitted for brevity

  it 'enables bikes to be removed' do
    bike = Bike.new
    subject.add_bike bike

    expect(subject.remove_bike).to eq bike
    expect(subject).to be_empty

    expect { subject.remove_bike }.to raise_error "#{described_class.name} empty"
  end
end
```

But this is considered bad practice as the test is responsible for testing 3 different things, which violates the *Single Responsibility Principle (SRP)*. Furthermore, the RSpec output `remove_bike enables bikes to be removed` doesn't really tell us how the method behaves - we would still have to read the code.

Can you get these tests to pass? It should be as simple as the following:

```
def remove_bike
  raise "#{self.class.name} empty" if empty?
  bikes.pop
end
```

Let's look at our shared requirements for docking station, van and garage again:

- They contain bikes
- They have a capacity
- You can put bikes in
- You can take bikes out

Have we fulfilled them? It feels like we have, so how do we go about implementing this new behaviour. First, we should test drive it. Add the following to `spec/docking_station_spec.rb`:

```
require 'docking_station'
require 'support/shared_examples_for_bike_container'

describe DockingStation do
```

```
  # other code omitted for brevity

  it_behaves_like BikeContainer
end
```

Run RSpec and make sure the new tests are failing. Now let's fix it in `lib/docking_station.rb`

```
require_relative 'bike'
require_relative 'bike_container'

class DockingStation
  include BikeContainer

  def release_bike
    fail 'No bikes available' if working_bikes.empty?
    bikes.delete working_bikes.pop
  end

  def dock(bike)
    add_bike bike
  end

  private

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

Whoa! Where has all the other code gone? Well, everything we've defined in `BikeContainer` is now also included in `DockingStation`, so we can remove all of the duplicate code. Notice how we have kept `dock` and `release_bike` though. Releasing a working bike from a docking station is not the same as removing the next bike along, so we still need this specialized method. `dock` is really now just an alias for 'add_bike' as it delegates directly to that method. This is a common approach in Ruby and improves the readability and domain-relevance of our code. `docking_station.dock` is more intuitive than `docking_station.add_bike`.

You may have implemented things slightly differently, but take a look at the code for `release_bike`. What's happening in the line `bikes.delete working_bikes.pop`? Discuss this with your pair partner and discuss other ways to approach this problem.

Now you can create refactor your `Van` and/or `Garage` classes so that they will reuse `BikeContainer`. You'll need to think how to extend the functionality of the existing methods. For example, will you create aliases for `add_bike` and `remove_bike`? Maybe 'load' and 'unload' for `Van`. The garage must fix the bikes as they arrive. However, the `add_bike()` method knows nothing about fixing bikes:

```
def add_bike(bike)
  raise "#{self.class.name} full" if full?
```

```
  bikes << bike
end
```

There are a number of ways you might approach this. One would be to define a separate method, which fixes the bike, then delegates to `add_bike`:

```
def accept(bike)
    bike.fix
    add_bike bike
end
```

The advantage is the simplicity. The disadvantage is that this isn't quite consistent with the real domain. A bike does not get fixed simply by being put in the garage - this is a separate stage. We have tightly coupled the fixing of bikes with the adding of bikes. It might be preferable to create a `fix_bikes` method which, for now, simply fixes all bikes in the garage:

```
require 'garage'
require 'support/shared_examples_for_bike_container'

describe Garage do
  it_behaves_like BikeContainer

  it 'fixes broken bikes' do
    bike = double :bike
    subject.add_bike bike
    expect(bike).to receive :fix
    subject.fix_bikes
  end
end
```

Notice how this test is structured. This problem is similar to the `release_bike` test for docking station. It is tempting to test that the bike is not broken after we've called `fix_bikes`. However, by using doubles, we can see that this would be a vacuous test:

```
require 'garage'
require 'support/shared_examples_for_bike_container'

describe Garage do
  it_behaves_like BikeContainer

  it 'fixes broken bikes' do
    bike = double :bike, fix: nil, broken?: false
    subject.add_bike bike
    subject.fix_bikes
    expect(bike).not_to be_broken
```

```
    end
  end
```

We have to stub `broken?` because `bike` is a double; but we are then testing that the stubbed method returns `false`. So this is no longer a test of `Garage`, but a test of RSpec doubles! One of the reasons the London style is so effective is it helps ensure we are testing the behaviour of the correct object.

By testing `expect(bike).to receive :fix`, we are ensuring the `fix_bikes` method has the correct behaviour. The fact that the bike subsequently becomes fixed is the responsibility of the `fix` method in `Bike` and should be unit tested in `bike_spec.rb` accordingly.

Nevertheless, we should still have a test that creates a garage, adds bikes, fixes them and then tests that the bikes are no longer broken. But what sort of test would this be?

Try using what we've learned in this stage to complete all the feature and unit tests for the docking station, van and garage, refactored to use `BikeContainer`.

Once we're done, let's go for some extensions!

Forward to the Challenge Map