The Single Responsibility Principle

Back to the Challenge Map

Our program is working, but it's getting complex. We need to **refactor** it into a better shape. We will do so using the first of several design principles: the **Single Responsibility Principle** ('SRP').

Consider the following lines of the `dock` method:

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

This method is doing two things:

1. Defining the capacity of the docking station (20), and
2. Docking a bike.

As it stands, `dock` *does not have a Single Responsibility*.

In this challenge, you will use **private methods** to extract method responsibilities to other methods.

> You do not need to test `private` methods.

Learning Objectives covered

- Use the Single Responsibility Principle
- Refactor code for readability

To complete this challenge, you will need to:

- [ ] Define a `full?` **predicate** method that uses some of the `dock` code to return `true` or `false` depending on whether the station is full or not
- [ ] Rewrite the guard condition of your `dock` method to incorporate your new `full?` method
- [ ] Do the same for `release_bike`, using an `empty?` method
- [ ] Use the `private` keyword to ensure these methods cannot be called from 'outside' instances of the `DockingStation` class.

Hints

▶ CLICK ME

Resources

- Single Responsibility Principle (Thoughtbot) <-- extracts classes rather than methods, but the principle still applies
- `private` methods in Ruby

Walkthrough

Previous Challenge | Next Challenge

Removing 'magic numbers'

Back to the Challenge Map

Our code is starting to look pretty. It's legible - we could pass it to another developer with little or no explanation needed - and it's following the Single Responsibility Principle pretty well. However, we're making references to the number `20` repeatedly, without explaining what it is.

Since the number `20` is fixed in stone for now, we can use a **constant** to refactor this 'magic number' out of our code.

In this challenge, you will further refactor your code, using a **constant**.

Learning Objectives covered

- Use a constant

To complete this challenge, you will need to:

- ☐ Define a constant, `DEFAULT_CAPACITY`, that stores the number `20`. Do this within the `DockingStation` class.
- ☐ Remove references to the magic number `20` in your implementation, using `DEFAULT_CAPACITY` instead.
- ☐ Refactor your tests to use this new constant instead of the magic number `20`.

Hints

▶ CLICK ME

Resources

- Ruby Constants (RubyDoc)
- Magic Numbers

Walkthrough

Previous Challenge | Next Challenge

Initialization defaults

Back to the Challenge Map

Now we have a constant, `DEFAULT_CAPACITY`, that forces all `DockingStation` instances to accept a maximum of 20 `Bike` instances, in an array stored as an instance variable `@bikes`. Nice!

Unfortunately for us, here comes another email from the client: they want system maintainers to be able to set variable capacities on new `DockingStation` instances. This should default to `20` if no capacity is supplied.

Here's a User Story for that:

```
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.
```

In this challenge, you will modify your `intialize` function to accept a `capacity` argument with a **default value** set to the `DEFAULT_CAPACITY`.

Learning Objectives covered

- Set an initial attribute value using `initialize`
- Set a default initialization value

To complete this challenge, you will need to:

- [ ] Write a feature test that allows a user to set a `@capacity` instance variable when `DockingStation.new` is called.
- [ ] Create Unit tests for this
- [ ] Implement the functionality in your code.
- [ ] Write a feature test that ensures a default capacity of `20` is set when no parameters are passed to `DockingStation.new`
- [ ] Create a unit test for this default capacity
- [ ] Use a default argument value within the `initialize` method to make this test pass.

Hints

▶ CLICK ME

Resources

- Method arguments in Ruby

Walkthrough

Previous Challenge | Next Challenge

Dealing with broken bikes

Back to the Challenge Map

Our system is looking good. We can vary the capacity of docking stations, but only at the point of creation. `DockingStation` instances will default to a capacity of 20 if none is provided. We're all tested, and our code is readable. Let's get harder!

We're now given three 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).
```

In this challenge, you will implement the above features using the skills you have gained so far.

Learning Objectives covered

- Implement a feature from scratch

To complete this challenge, you will need to:

- ☐ Complete the Red-Green-Refactor TDD cycle for each of the features above.

Hints

▶ CLICK ME

Resources

- 💊 The BDD Cycle

Walkthrough

Previous Challenge | Next Challenge

Isolating Tests with Doubles

Back to the Challenge Map

Our system can now do pretty much everything our client wants. However, our Unit Tests are not **isolated**. If we mess something up in the `Bike` class, all our tests for `DockingStation` will fail. We might spend hours looking for a problem we think resides within `docking_station.rb`, when in fact it's located in `bike.rb`.

We can use a dummy object, a **double**, Unit Tests that interact with other classes. We can define them to act predictably. They will not be affected by bugs that might arise in the classes they're 'standing in' for. You could think of them as 'stunt doubles' for the actual classes they represent.

In this challenge, you will **isolate** your Unit Tests by using **doubles**.

Your tests will be failing on completion of this challenge. Proceed to the next challenge regardless.

Challenge Setup

Here is a *non-isolated* test for `DockingStation` that relies on `Bike`:

```
# in docking_station_spec.rb
it 'releases working bikes' do
  # what if this line fails because of
  # a syntax error in our Bike class?
  # We need a way to remove this test's
  # dependency on Bike.
  subject.dock Bike.new
  bike = subject.release_bike
  expect(bike).to be_working
end
```

Realistically, it doesn't matter *what* object we pass to `dock`, as our `DockingStation` instance will happily store anything:

```
it 'releases working bikes' do
  # let's substitute our Bike.new
  # for a String.new
  subject.dock String.new("I'm not a bike!")
  # no error yet: and no problem when
  # we release the 'bike': we just
  # get the string we made
  bike = subject.release_bike
  # a problem here: strings don't
  # know how to respond_to working?
  # (we'll deal with that in the next
  # challenge: mocking behaviour).
  expect(bike).to be_working
end
```

RSpec has a `double` object which works just like the string above:

```
it 'releases working bikes' do
  # let's substitute our Bike.new
  # for a double
  subject.dock double(:bike)
  # no error yet: and no problem when
  # we release the 'bike': we just
  # get the double we made
  bike = subject.release_bike
  # a problem here: this double doesn't
  # know how to respond_to working?
  # (we'll deal with that in the next
  # challenge: mocking behaviour).
  expect(bike).to be_working
end
```

In the next challenge, we'll figure out how to get this test passing (i.e. how to tell a `double` to `respond_to` the `working?` method with the value `true`).

Learning Objectives covered

- Explain why doubles are needed to isolate unit tests
- Use a double to isolate a unit test

To complete this challenge, you will need to:

- [ ] Note down everywhere in a Unit Test where you refer to a class other than the class being tested.
- [ ] Use the RSpec `double` keyword to substitute all references to these classes with `double`s.
- [ ] Run `rspec` and see that the tests fail.
- [ ] Explain to your pair partner why the tests are failing.

Hints

▶ CLICK ME

Resources

- Doubles
- RSpec Mocks (docs)

Walkthrough

Previous Challenge | Next Challenge

Mocking Behaviour on Doubles

Back to the Challenge Map

We've isolated our tests, but now they're failing. This is happening because our `double`s can't respond to all the messages `Bike` instances can, like `broken?`.

In this challenge, you will **mock** behaviour by allowing `double`s to respond to certain methods with predefined values. These predefined method-value relationships are called **method stubs**.

Challenge Setup

Remember our isolated test from the last challenge? It looked a bit like this:

```
it 'releases working bikes' do
  # let's substitute our Bike.new
  # for a double
  subject.dock double(:bike)
  # no error yet: and no problem when
  # we release the 'bike': we just
  # get the double we made
  bike = subject.release_bike
  # a problem here: this double doesn't
  # know how to respond_to working?
  # (we'll deal with that in the next
  # challenge: mocking behaviour).
  expect(bike).to be_working
end
```

This test is failing, because the `double` doesn't know how to `respond_to` the `working?` method with the value `true`.

Let's implement that by **mocking** behaviour on the `double`:

```
# let's extract the double to a let
# statement so we can use it repeatedly
let(:bike) { double :bike }
it 'releases working bikes' do
  # let's superpower our double
  # using allow().to receive().and_return()
  allow(bike).to receive(:working?).and_return(true)
  subject.dock(bike)
  released_bike = subject.release_bike
  # Now the double responds to working?
  # with the value true
  expect(released_bike).to be_working
end
```

There are ways of DRYing the above up, as well: check the Resources for more information.

Learning Objectives covered

- Explain why method stubs are needed to isolate unit tests
- Use a method stub to isolate a unit test
- Discuss 'London' and 'Chicago' testing styles

To complete this challenge, you will need to:

- [ ] Find the first failing test
- [ ] Note down the method that the test `double` must respond to for the test to pass
- [ ] Use RSpec's `allow` syntax to permit the double to respond to methods the test requires
- [ ] Repeat for each test
- [ ] Refactor the `allow` statements to use method stubs at the point of `double` creation (see shorthand syntax)
- [ ] Ensure all your feature and unit tests are passing

Hints

▶ CLICK ME

Resources

- Mocking a simple return value
- RSpec Method Stubs (shorthand) (Github)

Walkthrough

Previous Challenge | Next Challenge

Men with Ven

Back to the Challenge Map

> You get a van, Jez. We could be men...with ven. *Super Hans, Peep Show*

Now we have a system capable of fulfilling all of our User Stories, and therefore all of our features. Our Unit Tests are isolated, and our Feature Tests are thorough.

Now, let's introduce two new objects to our domain: Vans and Garages. As you implement them you will realise how easily you can extend your domain when you follow the practices you've met this week.

Here are your User Stories:

```
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.
```

In this challenge, you will implement two User Stories with minimal scaffolding.

Learning Objectives covered

- Implement a feature from scratch

To complete this challenge, you will need to:

- ☐ Use the Feature - Unit cycle to implement the above User Stories.

Hints

▶ CLICK ME

Resources

- The BDD Cycle

Walkthrough

Previous Challenge | Next Challenge

Modules as mixins

Back to the Challenge Map

You have now built classes for `Van` and `Garage` that are very similar to the `DockingStation` class. We can tell they are similar as their is quite a lot of duplicated code between them.

When you have similar behaviours across classes, you should extract those similar behaviours to a **mixin**. Then, you can `include` the mixin within a class to gain all of the behaviour that mixin implements. Such a process is called **Object Composition**.

In Ruby, you use `Module`s as mixins.

In this challenge, you will refactor your code to **compose** objects using mixins. You will test-drive this using RSpec's **shared example** feature.

Learning Objectives covered

- Extract shared behaviour to a Module
- Test Modules
- Mix a Module into a Class using `include`

To complete this challenge, you will need to:

- ☐ Write down behaviour shared by the `DockingStation`, `Van`, and `Garage` classes (e.g. 'they all contain bikes')
- ☐ Create a spec file for a `Module` (called `BikeContainer`) that uses RSpec's shared example feature
- ☐ Extract relevant tests from your existing Unit Tests for `DockingStation`, `Garage`, and `Van` into the shared example tests for `BikeContainer`
- ☐ Implement `BikeContainer`'s behaviour via TDD
- ☐ Use `include` to mix `BikeContainer` into `DockingStation`, `Garage`, and `Van`
- ☐ Remove methods from `DockingStation`, `Garage`, and `Van` if they duplicate functionality from `BikeContainer`.

Resources

- RSpec Shared Examples (Relish)
-  Modules

Walkthrough

Previous Challenge