Pair programming

The key component of pair programming is regular driver/navigator switching 🔀. When pair programming, at any one time, one person should be the **driver**, i.e. the person actually typing, while the other pair should be the **navigator**.

The **navigator's** role is to try and think more broadly about where the code is going; to act as a sounding board to the driver and to offer suggestions on architectural design or to be looking up documentation related to the task at hand. The navigator should avoid constantly mentioning spelling mistakes and other typos unless the driver is really stuck.

A Pairing Session Template

Here's a template for your next pairing session:

1. [ ] Agree on the high-level goal out loud.
2. [ ] Break the work into a handful of tasks and prioritize them.
3. [ ] Decide your driver/navigator swapping strategy.
4. [ ] Configure git to share credit.
5. [ ] Eliminate distractions.
6. [ ] Work.
7. [ ] Analyze the session with a mini retro.

1. Agree on the high-level goal out loud

State **out loud** what you hope to accomplish at a high level.

You wouldn't think it'd be possible for two people to start pairing without agreement about where they're headed, but it's surprisingly easy.

2. Break the work into a handful of tasks (and prioritize them)

It’s worth trying to break your high-level goal into a handful of smaller steps.

This has a number of benefits:

- It makes the goal less intimidating.
- You’ll spot dead ends and pitfalls more easily.
- You can sort your task list by priority.
- You’re more likely to notice that accomplishing task C would make B easier, and reorder appropriately.
- You can decide on a task based on your current energy levels.
- It gives you a clear place to put new tasks you think of while working.

Some folks like to write each task on its own index card. The stack of them lives in front of the navigator. Each card can be a nice home for notes or ideas to bring up when there is a break in the action.

3. Decide what will trigger a driver/ navigator swap

Unless you already know what works best for you, I strongly recommend the Pomodoro Technique:

1. Code for 25 minutes.
2. Take a 5 minute break.
3. Switch drivers.

Other pair programming styles exist if you wish to try them.

4. Configure git to share credit

If two of you work on some code, both your names should appear on the commit.

Here’s a handy guide to configuring git appropriately.

Bonus: GitHub understands this natively and will give you both credit for the commit.

A few tools exist to make this even easier:

- git pair

- git duet
- git-together

5. Eliminate distractions

Show respect for your pair and the work you’re about to do.

- Don’t bring your phone. Silence it if you do.
- Disable notifications on the machine you’re using to pair.
- Close email/Slack/Twitter/IRC. Never keep something distracting on a second monitor.

6. Work

Do the work!

Don’t forget:

- *When navigating:* ask questions rather than making demands.
- *When driving:* dictate what you’re doing and why.
- Err on the side of over-communication.
- Take lots of breaks.
- Swap roles frequently.
- Do the simplest thing that could possibly work (for now).
- Avoid these pairing antipatterns.

7. Perform a mini retro

Spend a few minutes after your session reflecting on the experience.

First, discuss what went well.

Then, consider what would make the next session 1% better.

Possible areas for improvement:

- **Focus**: did distractions sneak in?
- **Communication**: were there long stretches of no talking?
- **Pacing**: did the session feel like a grind at any point?

- **Division of responsibility**: did you split the work up well?
- **Code quality**: was your end-product high-quality?

TDD Remote pairing checklist process

- **Before**
  - Direct message your pair in Slack
  - Agree a start time
  - Agree who will create the Zoom meeting
- **Kick off**
  - Meet in Zoom
  - Check your audio quality
  - Emotional check-in
  - Agree when you'll take breaks
  - Agree on a high-level goal out loud
  - Break your goal down into a handful of tasks
  - Decide your driver/navigator swapping strategy
  - Eliminate distractions
  - Agree on how often you'll stop to give each other constructive feedback (for example, you could agree to do this after every half hour or hour of pairing)
- **Pair**
  - Switch driver/navigator regularly
  - Take breaks
  - Message your pair in Slack if you get disconnected
  - Be specific, reference what line of code you are talking about by using its line number

- **Wrap up**
  - Reflect on your pairing session, give your partner an appreciation and any last nuggets of feedback

Debugging process

Finding bugs and fixing them. You have to do this as systematically as possible:

- **Tighten the loop (find the *exact line* the bug is coming from)**
- **Get visibility** (**use p** to inspect everything to help you home in on the exact line)
- Once you know the *one thing* that is wrong, out of place, misspelled, or not giving you what you expect, try to fix it.

Feature testing vs Unit testing

A unit test should test a single piece of functionality (one class or one function for example).

A feature test should hit a route, make sure the response if what you want and the database reflects the changes you made (if there are).

Installing Rspec to get started

1. **gem install rspec** to install Rspec
2. **rspec --version** to verify the version of Rspec
3. cd into a project directory that you wish to configure for use with Rspec and type **rspec --init** to initialize RSpec within the project

This will create a spec folder for your tests, along with the following config files:

- a **spec** directory into which to put spec files
- a **spec/spec_helper.rb** file with default configuration options
- an **.rspec** file with default command-line flags

Rspec Syntax broken down

```
describe Calculator do
  describe "#add" do
    it "returns the sum of two numbers" do
      calculator = Calculator.new
      expect(calculator.add(5, 2)).to eql(7)
    end
  end
end
```

The describe Keyword

The word **describe** is an RSpec keyword. It is used to define an “Example Group” or a collection of tests. It takes a class name and/or string argument and is passed a block **(do/end).**

- **describe Class name**
- **Describe Method name**

The it Keyword

The **it** keyword defines an individual example (aka test or test case). **it** accepts both class name and string arguments and should be used with a block argument, designated with **do/end**. In this case, when we pass 5 and 2 to the #add method, we expect it to return 7. This is concisely expressed in our expectation clause, which uses one of RSpec’s equality matchers, eql.

The expect Keyword

The **expect** keyword is used to define an “Expectation” in RSpec. This is a verification step where we check, that a specific expected condition has been met.

The to keyword

The **to** keyword is used as part of **expect** statements. Note that you can also use the **not_to** keyword to express the opposite, when you want the Expectation to be false. You can see that to is used with a dot, **expect(message).to,** because it actually just a regular Ruby method.

The eql keyword

The **eql** keyword is a Matcher. You use Matchers to specify what type of condition you are testing to be true (or false).

```
describe Calculator do
  describe "#add" do
    it "returns the sum of two numbers" do
      calculator = Calculator.new
      expect(calculator.add(5, 2)).to eql(7)
    end
  end
end
```

1. describe the class
2. describe the method example group. Conventionally, the string argument for instance methods are written as “#method”, while string arguments for class methods are written as “.method”.
3. Write your test case/example with it.
4. Write your expectation using expect. The expect method is also chained with .to for positive expectations, or .to_not/.not_to for negative expectations. We prefer .not_to. Also, limit one expect clause per test case

Equality/Identity Matcher

Matcher	Description	Example
eq	Passes when actual == expected	expect(actual).to eq expected

eql	Passes when actual.eql?(expected)	expect(actual).to eql expected
be	Passes when actual.equal?(expected)	expect(actual).to be expected
equal	Also passes when actual.equal?(expected)	expect(actual).to equal expected

Comparison Matcher

Matcher	Description	Example
>	Passes when actual > expected	expect(actual).to be > expected
>=	Passes when actual >= expected	expect(actual).to be >= expected
<	Passes when actual < expected	expect(actual).to be < expected
<=	Passes when actual <= expected	expect(actual).to be <= expected
be_between inclusive	Passes when actual is <= min and >= max	expect(actual).to be_between(min, max).inclusive

be_between exclusive	Passes when actual is < min and > max	expect(actual).to be_between(min, max).exclusive
match	Passes when actual matches a regular expression	expect(actual).to match(/regex/)

Class/Type Matchers

Matchers for testing the type or class of objects.

Matcher	Description	Example
be_instance_of	Passes when actual is an instance of the expected class.	expect(actual).to be_instance_of(Expected)
be_kind_of	Passes when actual is an instance of the expected class or any of its parent classes.	expect(actual).to be_kind_of(Expected)
respond_to	Passes when actual responds to the specified method.	expect(actual).to respond_to(expected)

True/False/Nil Matchers

Matchers for testing whether a value is true, false or nil.

Matcher	Description	Example
be true	Passes when actual == true	expect(actual).to be true
be false	Passes when actual == false	expect(actual).to be false
be_truthy	Passes when actual is not false or nil	expect(actual).to be_truthy
be_falsey	Passes when actual is false or nil	expect(actual).to be_falsey
be_nil	Passes when actual is nil	expect(actual).to be_nil

Error Matchers

Matchers for testing whether a block of code raises an error.

Matcher	Description	Example
raise_error(ErrorClass)	Passes when the block raises an error of type	expect {block}.to

raise_error("error message")	Passes when the block raise an error with the message “error	expect {block}.to raise_error(“err
raise_error(ErrorClass, "error	Passes when the block raises an error of type ErrorClass with the	expect {block}.to raise_error(Err

Test Doubles/RSpec Mocks

A Double is an object which can “stand in” for another object

As you can see, using a **test double** allows you to test your code even when it relies on a class that is undefined or unavailable. Also, this means that when there is a test failure, you can tell right away that it’s because of an issue in your class and not a class written by someone else.

In RSpec, a stub is often called a Method Stub, it’s a special type of method that “stands in” for an existing method, or for a method that doesn’t even exist yet.

The most common hooks used in RSpec are before and after hooks. They provide a way to define and run the setup and teardown code we discussed above.

```
describe "Before and after hooks" do
   before(:each) do
      puts "Runs before each Example"
   end

   after(:each) do
```

```
      puts "Runs after each Example"
   end

   before(:all) do
      puts "Runs before all Examples"
   end

   after(:all) do
      puts "Runs after all Examples"
   end

   it 'is the first Example in this spec file' do
      puts 'Running the first Example'
   end

   it 'is the second Example in this spec file' do
      puts 'Running the second Example'
   end
end
```

When you run the above code, you will see this output −

```
Runs before all Examples
Runs before each Example
Running the first Example
Runs after each Example
.Runs before each Example
Running the second Example
Runs after each Example
.Runs after all Examples
```

RSpec - Subjects

The `subject` method is exactly like `let`, except for a few things:

1.) When using a class name for the example group(the describe block), you don't have to define the name of the method using let. The name of the method will be the argument of the describe block, so you don't have to name the method:

```
subject { Person.new }
```

2.) If you are keeping it simple, subject already made a helper method for you. So you wouldn't have to define the object either.

```
describe Person do
  context "attributes" do
    it "#name" do
      subject.name = "Ethan"
      expect(subject.name).to eq("Ethan")
    end
  end
end
```

See how I didn't have to define subject at all? The argument name of the example group was Person. subject already created a helper method with a simple new instance, Person.new.

Subject is what is under test, usually an instance or a class you are testing.

Let is a way for you to create instance variables in your test that are available between tests. The other thing is it's lazy, meaning it's not actually assigned until it's called in a test.

let is a method that takes an argument, (:card) in the example. By calling let(:card) you have a method card available for you to use. You do not have a method let available to you. In fact, you'd likely get an error in your test. You could get it to work with let(:let) {...} but it would be weird and hard to read 😛.The purpose of using lets to help isolate behavior.

The thing with subject is this, RSpec knows more about subject. It's special. On of the ways that it's special is found in this implicit example in the docs. You can see it { is_expected.to be_empty } is creating an expectation on subject without you having to write subject anywhere. It just knows that that the object under test is whatever subject is. let does not provide this.

I agree, they have similarities under the hood because it's really just ruby, but their intended use is different.

Just incase someone wants to read the docs in the future:
subject in docs
let in docs

1. subject will automatically be assigned as an instance of the class being tested.
2. It allows for the use of "one liner" tests: