



University of
Sheffield

COM1001 *Introduction to Software Engineering* SPRING
SEMESTER

Testing

Professor Phil McMinn

Test Cases

In the last lecture, we briefly mentioned **test cases**.

Test cases are **inputs**, or sequences of inputs, to a program or unit of a program (such as a method), along with **expected outputs**.

Test cases help us verify that a program is working correctly.

Software Testing

Software Testing

Testing is the process of writing test cases and executing them.

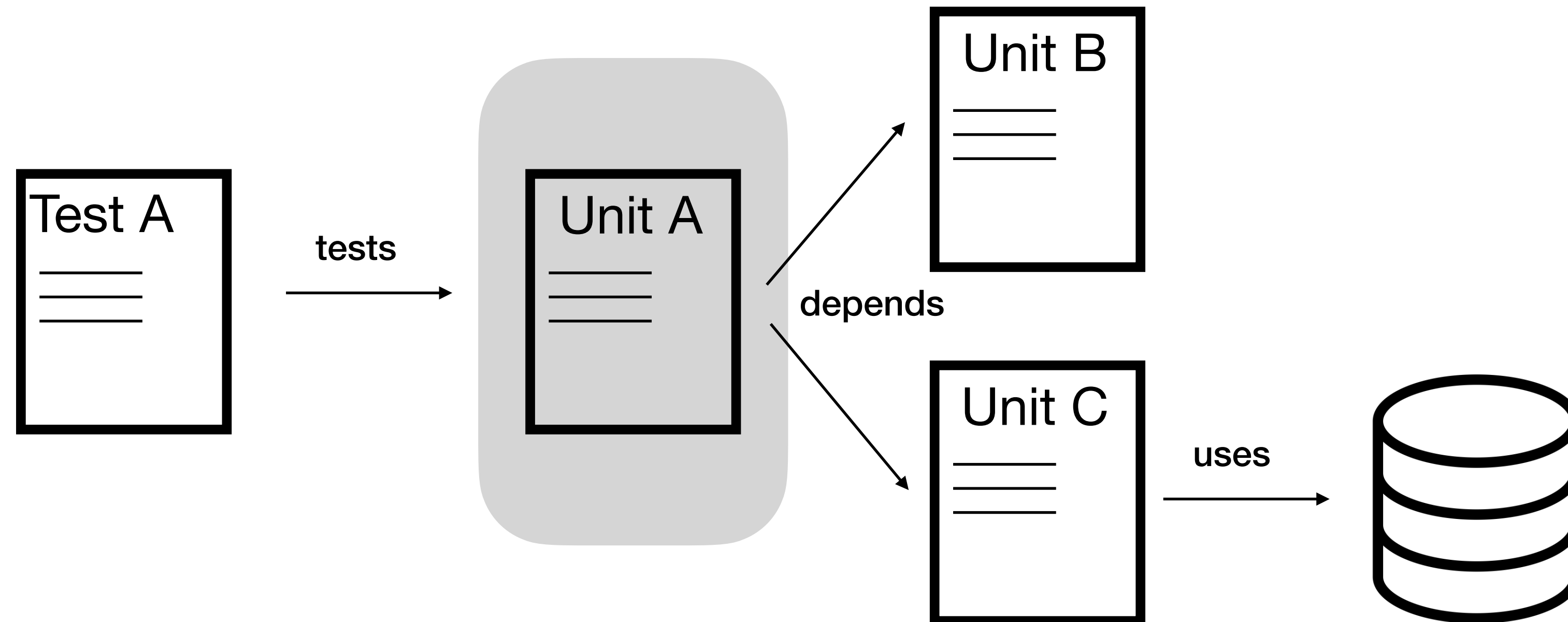
Testing is a separate activity from debugging.

Testing is a **mental discipline** that helps all software professionals develop **higher quality software**.

Unit Testing

A unit is an individual component of a system, such as a class or an individual method.

Testing units in isolation is called **unit testing**.



Unit Testing

A unit is an individual component of a system, such as a class or an individual method.

Testing units in isolation is called **unit testing**.

- **Fast**
- **Easy to control**
- **Easy to write**
- **Lack reality**
- **Cannot catch all bugs**
(e.g. interactions with other components or services)

Unit tests are a very useful type of test but are often insufficient on their own.

Integration Tests

Testing in isolation is not enough. Sometimes code goes “beyond” the system’s borders and uses other (often external) components – for example, a database.

Integration tests test the integration between our code and that of external parties.

Example: Testing methods that access a database via SQL queries. Do our methods obtain the right data from the database?

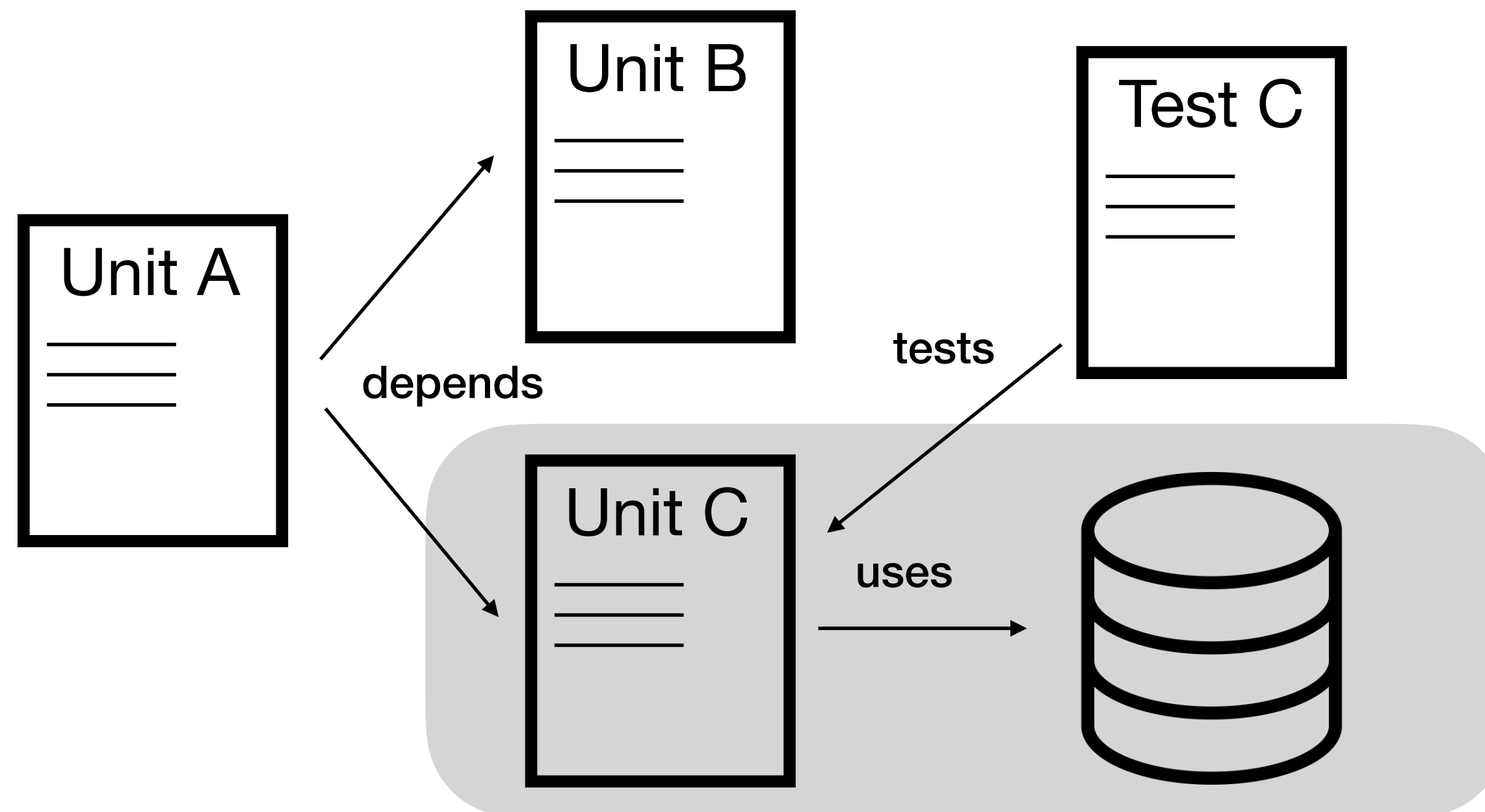
- **Can capture integration bugs**
- **Less complex than writing a system test that goes through the entire system, including components we do not care about**
- **Hard to write**, for example:
 - Need to use an isolated instance of the database
 - Put it into a state expected by the test
 - Reset the state afterwards

Integration Tests

Testing in isolation is not enough. Sometimes code goes “beyond” the system’s borders and uses other (often external) components – for example, a database.

Integration tests test the integration between our code and that of external parties.

Example: Testing methods that access a database via SQL queries. Do our methods obtain the right data from the database?

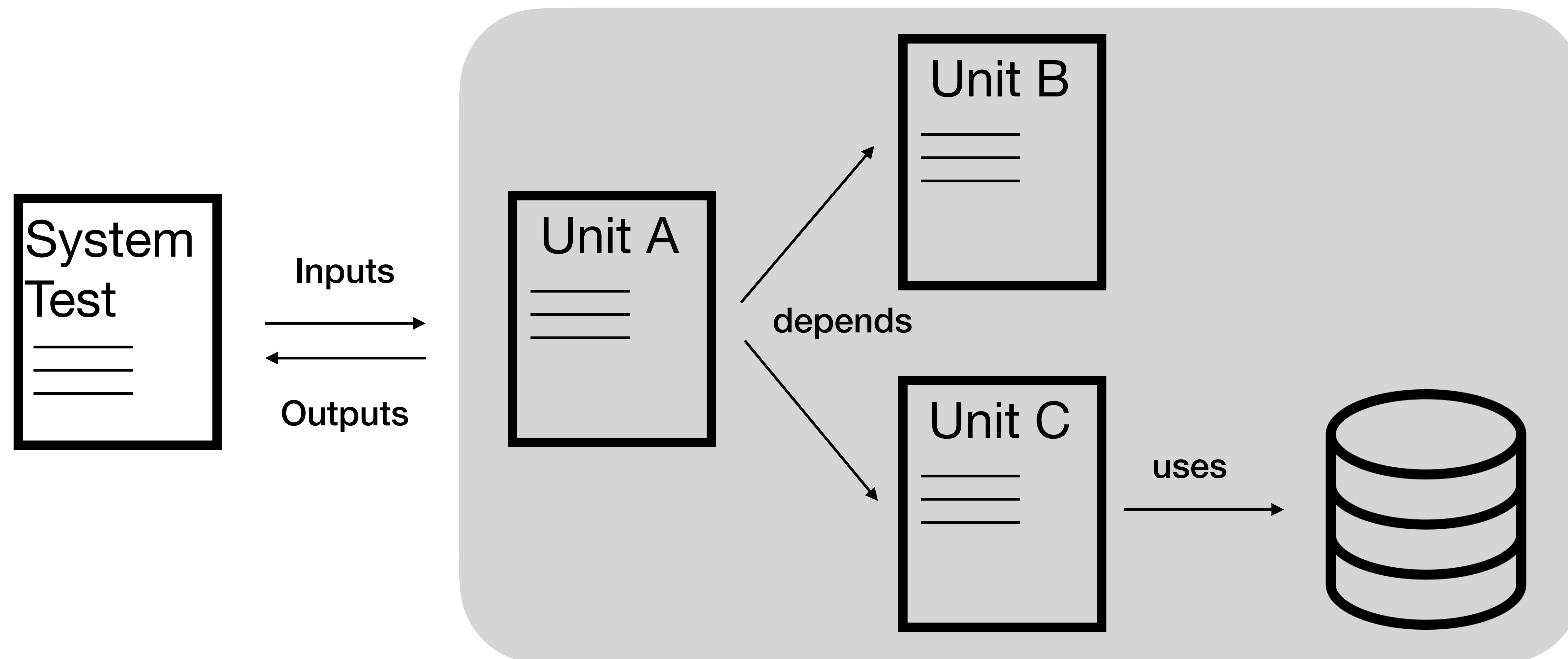


System Tests

To get a more realistic view of the software we should also perform more realistic tests with it – with all its database, front-end, and other components.

We do not care about how the system works from the inside.

We care that given certain inputs, certain outputs are provided by the system.



System Tests

To get a more realistic view of the software we should also perform more realistic tests with it – with all its database, front-end, and other components.

We do not care about how the system works from the inside.

We care that given certain inputs, certain outputs are provided by the system.

- **Realistic**

(when the tests perform similarly to the end user, the more confident we can be that the system will work correctly for all end users)

- **Slow!**

- **Hard to write**

(lots of external services to account for)

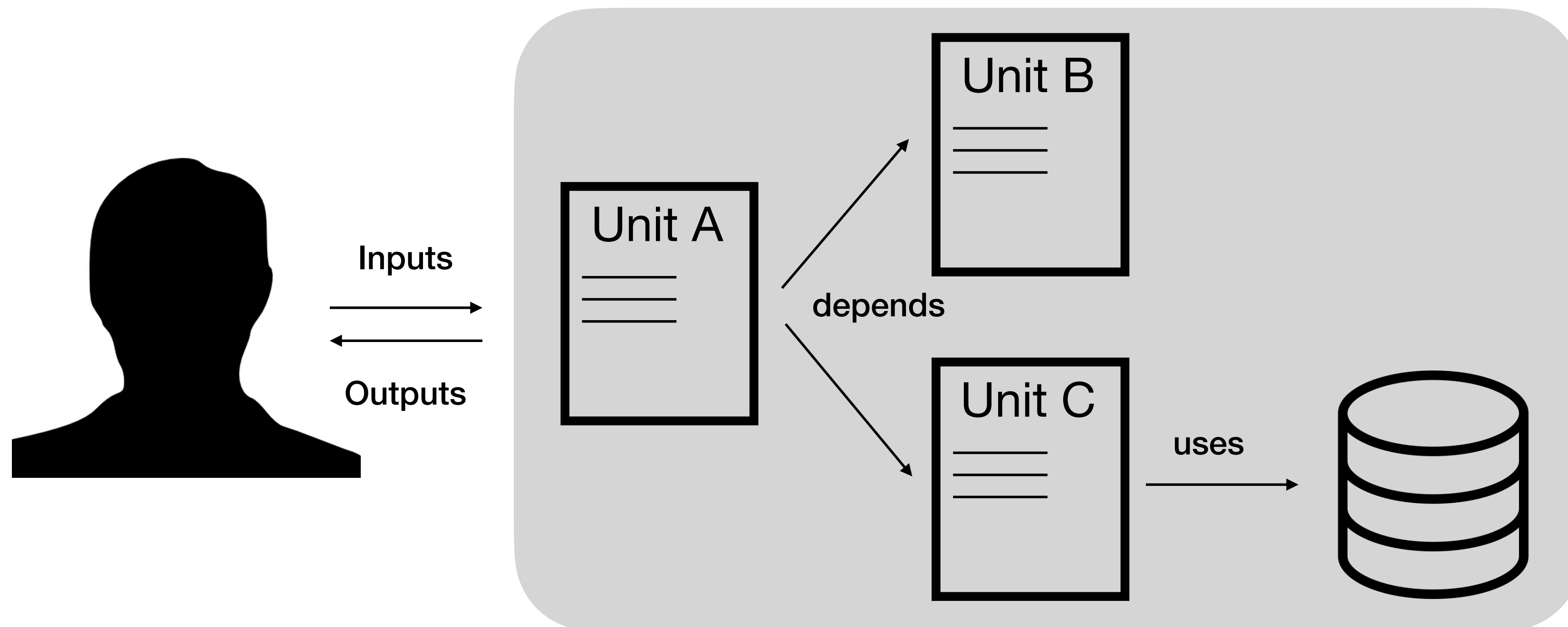
- **Prone to Flakiness**

Manual Tests

Not everything can be tested easily in an automated fashion, particularly where there are qualitative judgements (e.g., the quality of a search engine's results).

Furthermore, we may need to explore real system behaviour to know what automated tests to write.

Manual tests are system tests performed manually by a human.



Manual Tests

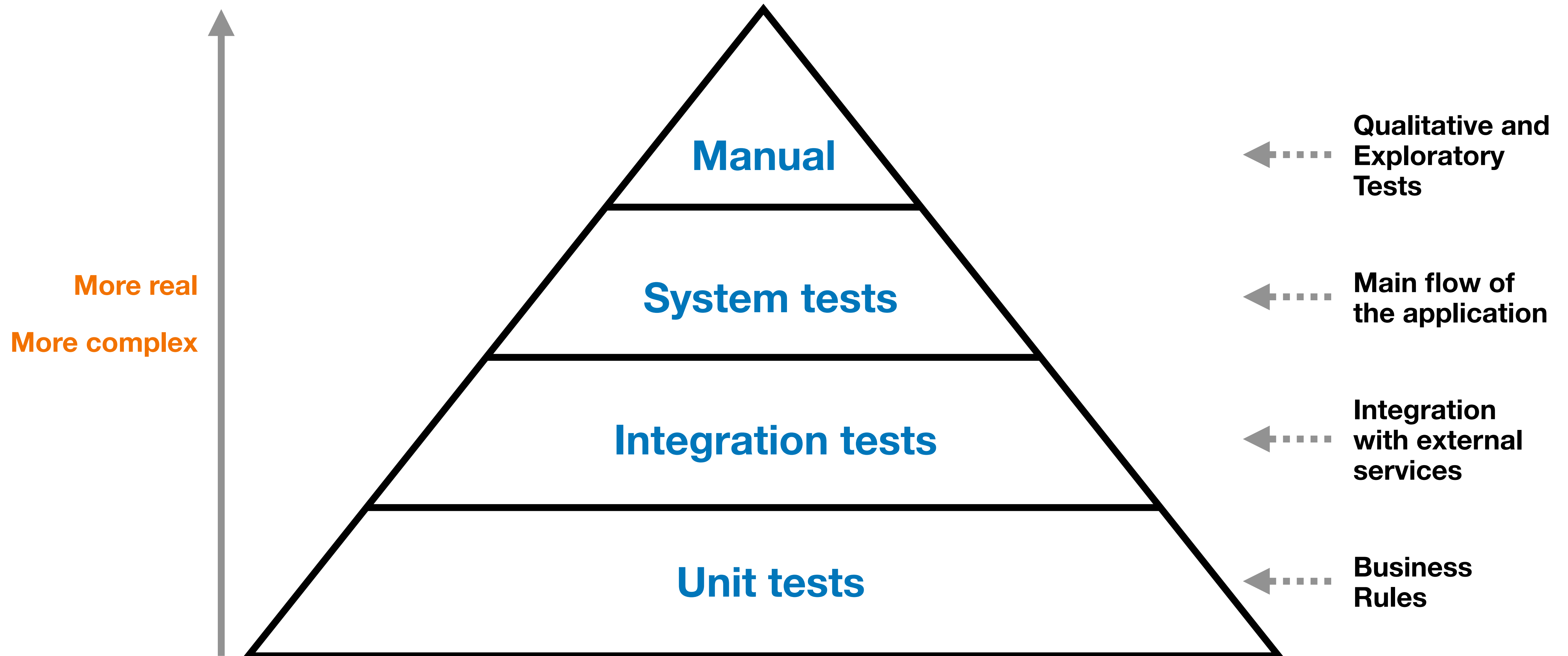
Not everything can be tested easily in an automated fashion, particularly where there are qualitative judgements (e.g., the quality of a search engine's results).

Furthermore, we may need to explore real system behaviour to know what automated tests to write.

Manual tests are system tests performed manually by a human.

- **Real**
(The tester is acting as an end-user, actually using the system)
- **Time-consuming**
- **Difficult to reproduce**
- **Tedious**

The Test Triangle



What Test am I?



What Test am I?



I am an automated test that interacts directly with code that uses a database

**Is this an example of a
unit, integration, system or a manual test?**

I am an automated test that interacts directly with code that uses a database

Is this an example of a
unit, integration, system or a manual test?

INTEGRATION

I test a web application. I load up the web page, automatically fill out forms, click buttons, and check the resulting web page.

**Is this an example of a
unit, integration, system or a manual test?**

I test a web application. I load up the web page, automatically fill out forms, click buttons, and check the resulting web page.

Is this an example of a
unit, integration, system or a manual test?

SYSTEM

I am essentially a series of inputs that a human inputs into a terminal. I don't check the answers, I leave that to my human. I don't really "exist" in any tangible form, but some humans write me into documents so that they know how to reproduce me.

**Is this an example of a
unit, integration, system or a manual test?**

I am essentially a series of inputs that a human inputs into a terminal. I don't check the answers, I leave that to my human. I don't really "exist" in any tangible form, but some humans write me into documents so that they know how to reproduce me.

**Is this an example of a
unit, integration, system or a manual test?**

MANUAL

RSpec

In this module, we're going to be using **RSpec** to write test cases.

RSpec can be used for all types of testing. We're going to use it predominantly to write **unit tests** and **system tests** (a type of system test).

The remainder of this lecture will focus on **unit tests** – i.e., **Ruby methods** and **individual web pages**.

Later in the module, we will look at some more advanced testing techniques, including testing sequences of web page interactions (end-to-end testing).

RSpec in Practice

```
def string_comparison(str1, str2)
  (1..[str1.length, str2.length].min).each do |i|
    return i if str1[i] != str2[i]
  end
  -1
end
```

debugging/string_comparison.rb

RSpec in Practice

```
def string_comparison(str1, str2)
  (1..[str1.length, str2.length].min).each do |i|
    return i if str1[i] != str2[i]
  end
  -1
end
```

```
require "rspec"
require_relative "../debugging/string_comparison"

RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

unit_testing/string_comparison_simple_spec.rb

RSpec – Behaviour-Driven

RSpec adopts a “**behaviour-driven**” approach to testing. It requires us to describe the behaviour of the code we’re testing in plain text. These descriptions serve as useful documentation, and ensure we have useful error messages if a test should fail.

We could just write out these descriptions without any test code to begin with, planning out our testing once some code has been written.

Or, we could write the descriptions first, as a **specification** (hence the name **RSpec**) for the code we are *about* to write. This is an established style of software development in industry, known as **test-first development**.

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

Describing

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

Describing

RSpec tests are grouped by the aspect of the system we are testing. They begin with an `RSpec.describe` block.

Inbetween the “`RSpec.describe`” and the “do” we would normally write the module or class name. Here, however, the method we’re testing is standalone.

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

Describing

RSpec tests are grouped by the aspect of the system we are testing. They begin with an `RSpec.describe` block.

Inbetween the “`RSpec.describe`” and the “do” we would normally write the module or class name. Here, however, the method we’re testing is standalone.

Nested within this is a `describe` block for each unit that we’re testing.

When unit testing a method, it is conventional to write the name of the method, prefixed with a hash.

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

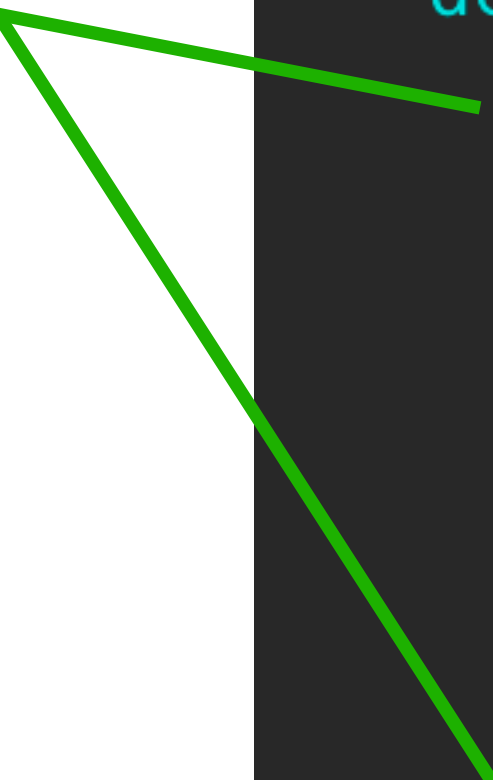
    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```


The Importance of the context

We then describe the different scenarios that we're testing the unit, with context blocks.

Here, we're supplying different arguments to the method, so we write about them.

It is RSpec convention to begin a context block description with the words “when”, “with”, or “without”.



```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end


    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

it

We then describe, given the context, how the method should behave, using an `it` block.

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```



Great expect-ations

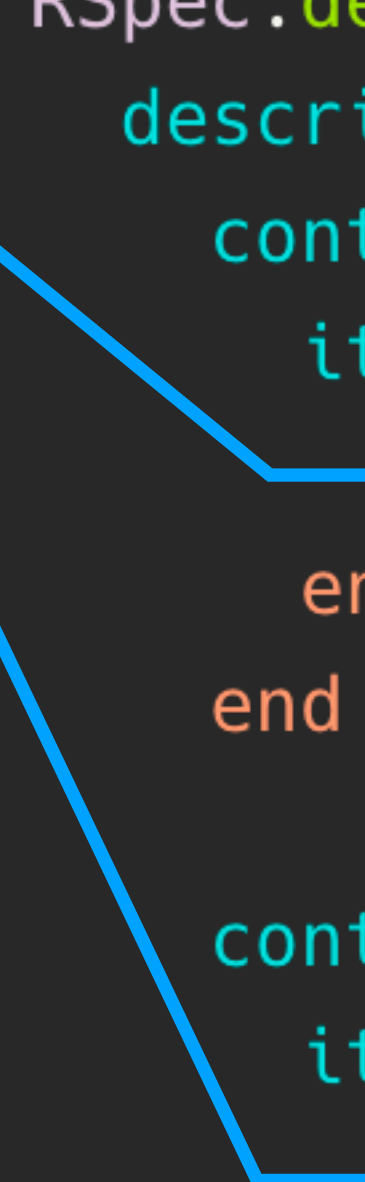
```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

Great expect-ations

And finally, we write these behaviours in terms of expect statements in Ruby code.

Here we're saying that we expect the result of the `string_comparison` method to be equal to the value in brackets



```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```


Great expect-ations

And finally, we write these behaviours in terms of expect statements in Ruby code.

Here we're saying that we expect the result of the `string_comparison` method to be equal to the value in brackets

Each expect statement involves a particular instance of a **matcher**.

For example, in this statement we **expect** the **actual result** (`result`) to **match** (be equal to) the **expected result** (`0`).

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two strings, 'aardvark' and 'aargh!'" do
      it "returns 3" do
        expect(string_comparison("aardvark", "aargh!")).to eq(3)
      end
    end

    context "when given two strings, 'sought' and 'bought'" do
      it "returns 0" do
        expect(string_comparison("sought", "bought")).to eq(0)
      end
    end
  end
end
```

Matchers

Expectations are expressed in the form:

```
expect(actual_result).to/not_to matcher(expected_result)
```

e.g., `expect(result).to eq(0)`

Other examples of **matchers** include:

Matcher	Passes if...
<code>be true</code>	<code>actual_result == true</code>
<code>be false</code>	<code>actual_result == false</code>
<code>be_nil</code>	<code>actual_result.nil?</code>
<code>be < expected_result</code>	<code>actual_result < expected_result</code>
<code>be > expected_result</code>	<code>actual_result > expected_result</code>
<code>be <= expected_result</code>	<code>actual_result <= expected_result</code>
<code>be >= expected_result</code>	<code>actual_result >= expected_result</code>
<code>be_between(x, y).inclusive</code>	<code>actual_result >= x && actual_result <= y</code>
<code>start_with(x)</code>	<code>actual_result.start_with?(x)</code>
<code>include(x)</code>	<code>actual_result.include?(x)</code>

For more information, see the RSpec documentation at <https://rspec.info/documentation>

Running RSpec

RSpec needs the **rspec** gem to be installed.

RSpec tests are then run using the **rspec** command. We can run the **rspec** command with a specific file, or point it at a directory.

When given a directory instead of a file, RSpec will run all the tests it finds in files ending with “**_spec.rb**”

```
codio@north-mister:~/workspace/com1001-code/unit_testing$ rspec string_comparison_simple_spec.rb
..

Finished in 0.00306 seconds (files took 0.08153 seconds to load)
2 examples, 0 failures
```

When tested with the corrected version of string_comparison, both tests **pass**.

```
codio@north-mister:~/workspace/com1001-code/unit_testing$ rspec string_comparison_simple_spec.rb
```

```
.F
```

Failures:

1) #string_comparison when given two strings, 'sought' and 'bought' returns 0

Failure/Error: expect(string_comparison("sought", "bought")).to eq(0)

expected: 0

got: -1

(compared using ==)

./string_comparison_simple_spec.rb:14:in `block (4 levels) in <top (required)>'

Finished in 0.02429 seconds (files took 0.08043 seconds to load)

2 examples, 1 failure

Failed examples:

rspec ./string_comparison_simple_spec.rb:13 # #string_comparison when given two strings, 'sought' and 'bought' returns 0

Here's what happens when they're run on the original buggy version of string_comparison

Why Test?

Why Test?

Testing helps us think, in advance of releasing our software, where it might fail.

Some of these failures could be quite costly.

Testing is quite challenging and sometimes requires more skill than development.

Why use a Testing Tool Like RSpec?

It helps automate aspects of the testing process

- We can keep re-running the tests after we small changes to our code, to ensure it is still working as we intended.
- This is called **Regression Testing**.

It helps with debugging

- It automates execution of the failing test case
- It tells us when the bug is fixed (the failing test case passes)
- It tells us whether we've broken anything else while fixing the bug (our other test cases pass too)

What Should We Test?

How do we go about testing?

How do we know what to test?

There are many strategies and techniques

Here are some general examples of test cases...

The Positive Test Case

Key Idea

Establish some intended behaviour of the code happens as expected

General Examples

- Testing the system **accepts correct, well-formed inputs**
- Testing the system **produces the right outputs** given those inputs

The Negative Test Case

Key Idea

Establish some unintended behaviour of the code does not happen

General Examples

- Testing the system **does not accept incorrect inputs**
- Testing the system **handles exceptional situations correctly**, by failing gracefully, reporting problems to the user etc.
- Testing the system is **secure** – for example, it does not reveal sensitive information inadvertently

The State Change Test Case

Key Idea

Establish the system changes state correctly.

Example

- Testing the system **logs in** and **logs out** correctly

The Boundary Test Case

Key Idea

Programmers typically make mistakes around the boundaries of conditions in a program.

Thoroughly test the system around boundaries involved in inputs or key conditions.

Example

Test the following method with $x=0$ and $x=1$

```
def positive?(x)
  return true if x >= 0

  false
end
```

Did the programmer mean to use “>” instead of “>=”?

Equivalence Partitioning

Key Idea

Systematically divide up the input space according to the expected behaviours of the system.

Group different inputs that result in the same or “equivalent” behaviours.

Choose one value from each group to test the program.

Example

Test the following method with $\text{day} = -10$, $\text{day} = 10$, $\text{day} = 40$ as examples of inputs that are **too small**, **in range**, and **too large** respectively.

```
def valid_day?(day, month, year)
  if (day < 1 || day > 31)
    # ...
```

The Magic Test Case

Key Idea

Test using domain knowledge about the program and certain inputs/scenarios that may cause it to fail

Example

Test the following method with `day=29, month=2, year=2000`

```
def valid_day?(day, month, year)
  # ...
```

A Better Test Suite for string_comparison

Note the different test case types – e.g., **positive** and **boundary** (on the length of the string) etc.

Take a look it in the com1001 code examples repository.

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two empty strings" do
      it "returns -1" do
        expect(string_comparison("", "")).to eq(-1)
      end
    end

    context "when given two strings that are different single characters" do
      it "returns 0" do
        expect(string_comparison("a", "b")).to eq(0)
      end
    end

    context "when given two strings of the same single character" do
      it "returns -1" do
        expect(string_comparison("a", "a")).to eq(-1)
      end
    end

    context "when given two strings that the same, of length > 1" do
      it "returns -1" do
        expect(string_comparison("ab", "ab")).to eq(-1)
      end
    end

    context "when given two strings of the same length, differing by the last character" do
      it "returns the length of the string minus 1" do
        expect(string_comparison("abc", "abz")).to eq(2)
      end
    end

    context "when given two strings of different length, first shorter than the second" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abc", "abcdef")).to eq(3)
      end
    end

    context "when given two strings of different length, second shorter than the first" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abcdef", "abc")).to eq(3)
      end
    end
  end
end
```

unit_testing/string_comparison_spec.rb

A Better Test Suite for string_comparison

Note the different test case types – e.g., **positive** and **boundary** (on the length of the string) etc.

Take a look at it in the `com1001` code examples repository.

Which test might have caught the original bug discussed in the debugging lecture?

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two empty strings" do
      it "returns -1" do
        expect(string_comparison("", "")).to eq(-1)
      end
    end

    context "when given two strings that are different single characters" do
      it "returns 0" do
        expect(string_comparison("a", "b")).to eq(0)
      end
    end

    context "when given two strings of the same single character" do
      it "returns -1" do
        expect(string_comparison("a", "a")).to eq(-1)
      end
    end

    context "when given two strings that are the same, of length > 1" do
      it "returns -1" do
        expect(string_comparison("ab", "ab")).to eq(-1)
      end
    end

    context "when given two strings of the same length, differing by the last character" do
      it "returns the length of the string minus 1" do
        expect(string_comparison("abc", "abz")).to eq(2)
      end
    end

    context "when given two strings of different length, first shorter than the second" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abc", "abcdef")).to eq(3)
      end
    end

    context "when given two strings of different length, second shorter than the first" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abcdef", "abc")).to eq(3)
      end
    end
  end
end
```

unit_testing/string_comparison_spec.rb

A Better Test Suite for string_comparison

Note the different test case types – e.g., **positive** and **boundary** (on the length of the string) etc.

Take a look at it in the `com1001` code examples repository.

Which test might have caught the original bug discussed in the debugging lecture?

What test cases do you think are missing, or could be added?

```
RSpec.describe do
  describe "#string_comparison" do
    context "when given two empty strings" do
      it "returns -1" do
        expect(string_comparison("", "")).to eq(-1)
      end
    end

    context "when given two strings that are different single characters" do
      it "returns 0" do
        expect(string_comparison("a", "b")).to eq(0)
      end
    end

    context "when given two strings of the same single character" do
      it "returns -1" do
        expect(string_comparison("a", "a")).to eq(-1)
      end
    end

    context "when given two strings that are the same, of length > 1" do
      it "returns -1" do
        expect(string_comparison("ab", "ab")).to eq(-1)
      end
    end

    context "when given two strings of the same length, differing by the last character" do
      it "returns the length of the string minus 1" do
        expect(string_comparison("abc", "abz")).to eq(2)
      end
    end

    context "when given two strings of different length, first shorter than the second" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abc", "abcdef")).to eq(3)
      end
    end

    context "when given two strings of different length, second shorter than the first" do
      it "returns the length of the shorter string" do
        expect(string_comparison("abcdef", "abc")).to eq(3)
      end
    end
  end
end
```

`unit_testing/string_comparison_spec.rb`

Unit Testing a Route

We can write automated RSpec tests to test Sinatra code too!

In order to do this we need to use the `rack-test` gem. This makes functionality available to us that we need to test web pages in addition to just methods.

First note that there's a lot of setup code we need to write to get this to work. This code tends to be repeated each time we write an RSpec test suite for a Sinatra web application. We will discuss ways to hide this away later in the module

We then request the page, check the HTTP status code – **200 OK**, and check the contents of the body of the page are as expected

```
require "rspec"
require "rack/test"

require_relative "hello_world"

RSpec.describe "Hello World Sinatra Example" do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  describe "GET /hello-world" do
    it "has a status code of 200 (OK)" do
      get "/hello-world"
      expect(last_response.status).to eq(200)
    end

    it "says 'Hello, World!'" do
      get "/hello-world"
      expect(last_response.body).to eq("Hello, World!")
    end
  end
end
```

basics/hello_world_spec.rb

```
describe "GET /first-route" do
  it "has a status code of 200 (OK)" do
    get "/first-route"
    expect(last_response).to be_ok
  end

  it "outputs the correct message for first-route" do
    get "/first-route"
    expect(last_response.body).to eq("This code is run when first-route is invoked")
  end
end

describe "GET /second-route" do
  it "has a status code of 200 (OK)" do
    get "/second-route"
    expect(last_response).to be_ok
  end

  it "outputs the correct message for second-route" do
    get "/second-route"
    expect(last_response.body).to eq("This code is run when second-route is invoked")
  end
end
```

basics/multiple_routes_spec.rb

Checking and re-checking each route after each change is time consuming.

RSpecs allow us to do this checking automatically from the terminal.

An alternative to checking the HTTP status code explicitly is to use the matcher **be_ok**

In general, it is not feasible to check the entire body of the web page matches our expectations. Typically we only check a part of it, or check that particular specifics are present.

```
describe "GET /times-table" do
  it "Says '1 times 3 = 3'" do
    get "/times-table"
    expect(last_response.body).to include("1 times 3 = 3")
  end

  it "Says '2 times 3 = 6'" do
    get "/times-table"
    expect(last_response.body).to include("2 times 3 = 6")
  end
end
```

basics/times_table_spec.rb