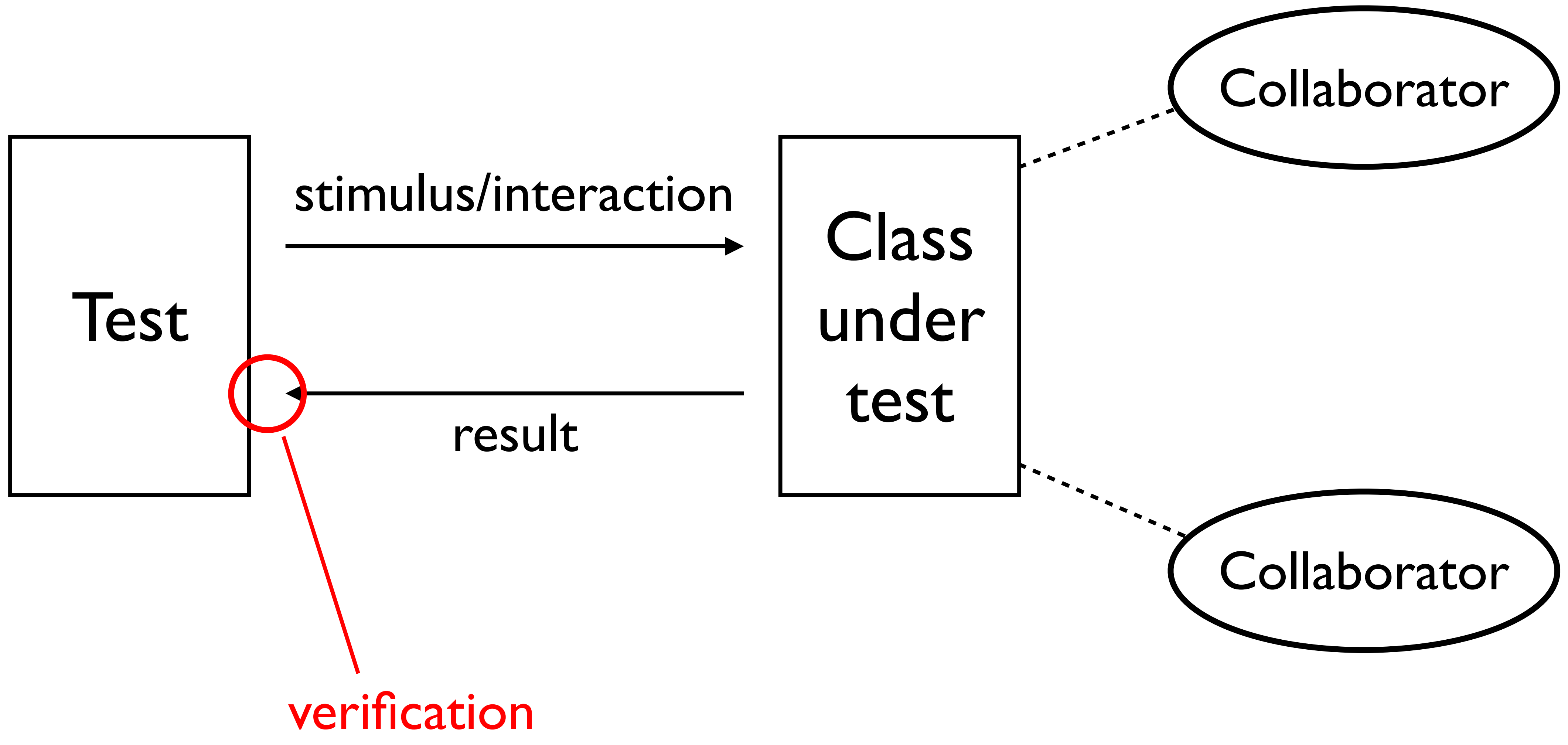
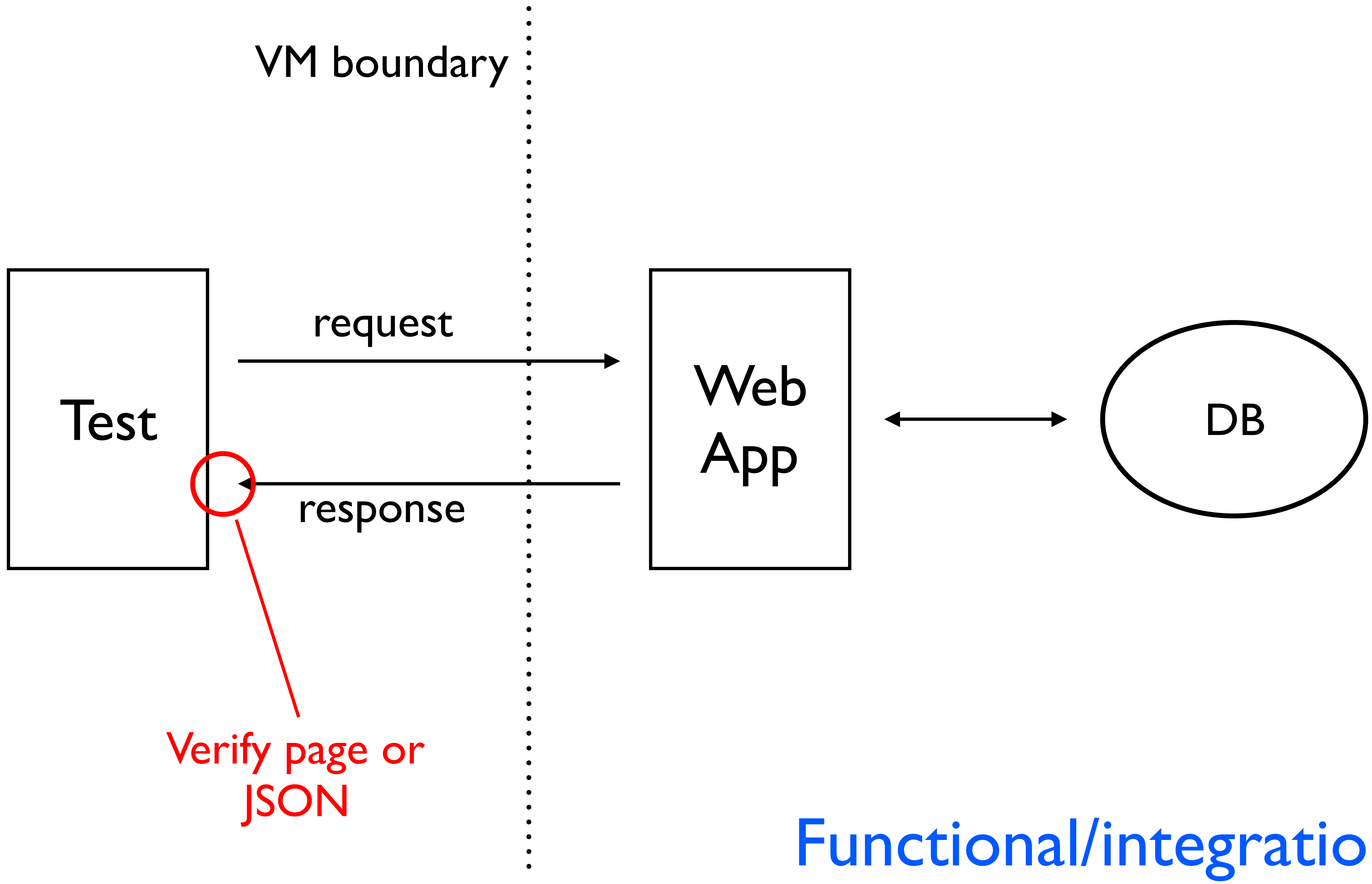
Tests


Test
stimulus/interaction
result
Class under test
Collaborator
Collaborator
verification

Tests give you

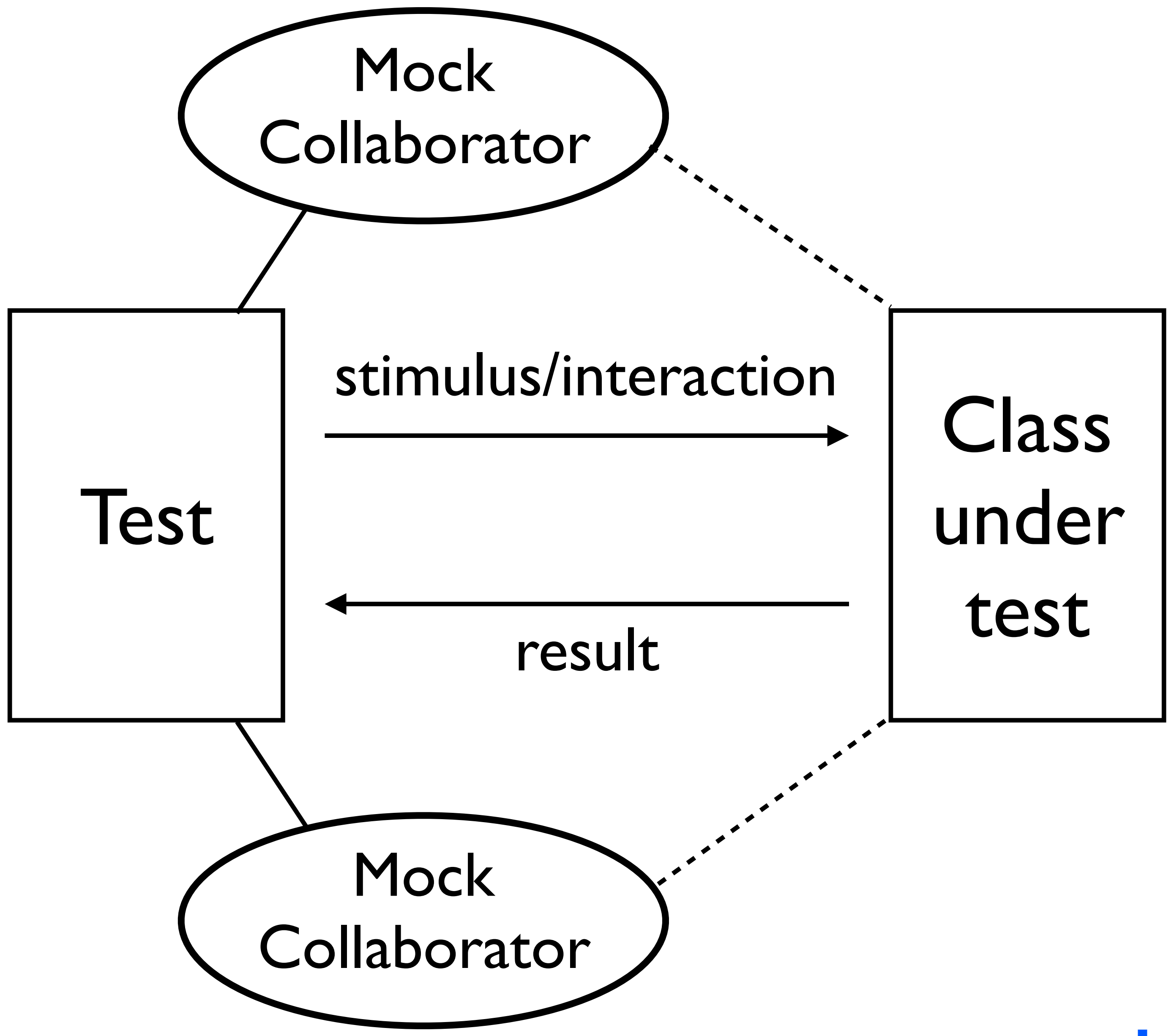
- Software reliability
- Confidence
- Safety when refactoring
- A codified specification

Tests at different depths



Functional/integration test

Tests at different depths



Unit test

Why?

- Unit tests:
  - quick to run
  - identify a broad range of bugs
- Higher level tests:
  - verify user-expected behaviour
  - test interactions between components

Two principles of testing

Invest time in making things easy to test

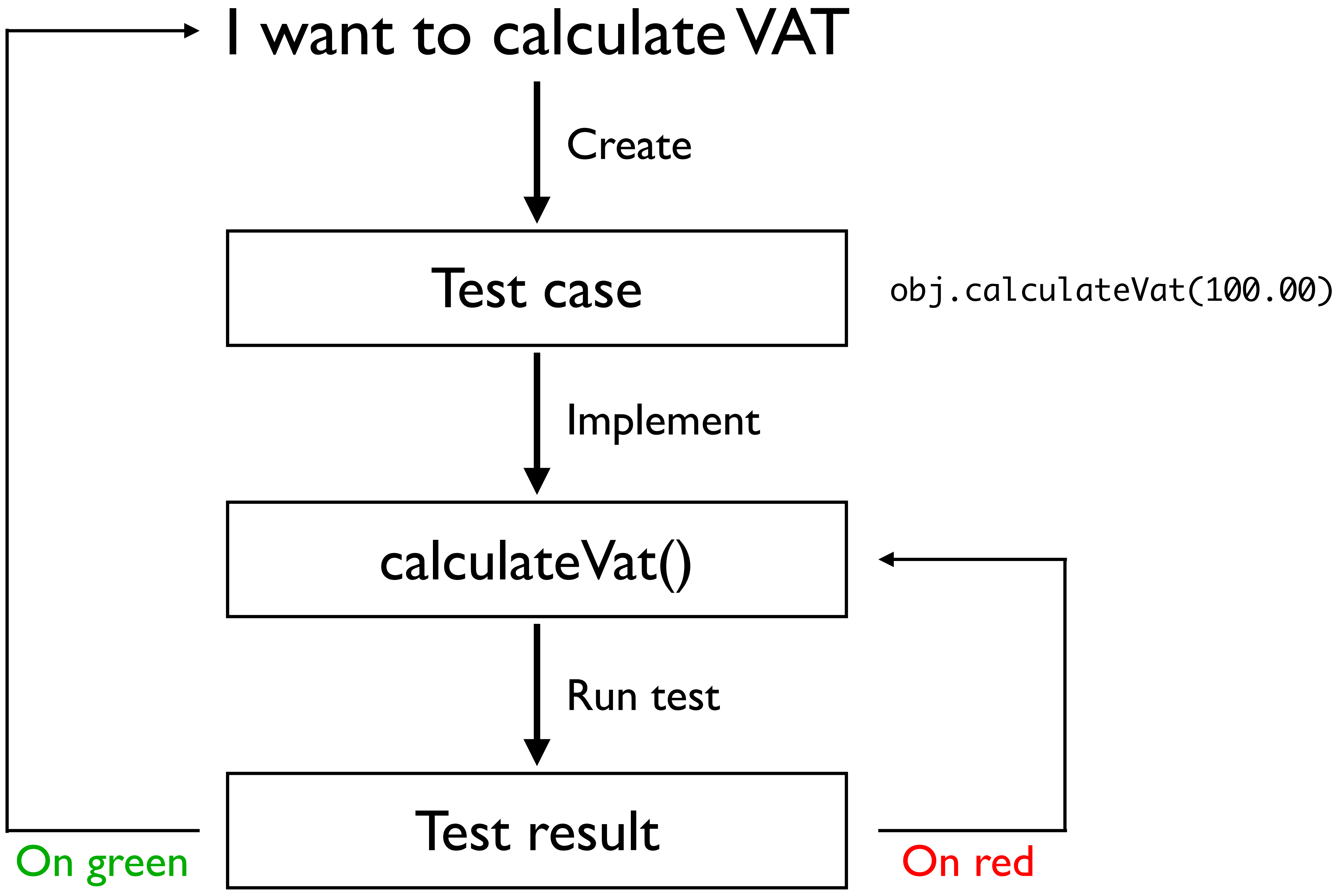
Practise, practise, practise

Test Driven Development

TDD gives you

- Guaranteed tests
- Classes that are easy to test
- Design through what you want, not how

Example


I want to calculate VAT
Create
Test case
obj.calculateVat(100.00)
Implement
calculateVat()
Run test
Test result
On green
On red

Focus on behaviour!

Behaviour Driven Development

BDD

- Evolution of TDD
- Dedicated “vocabulary”
- Structure for test cases
- Not specific to tests at a particular depth

BDD origins

http://dannorth.net/introducing-bdd/

Example

Scenario Should set start date when enrolling new student

Given A new student

When I enroll the student

Then Their start year becomes the current year

The Groovy solution

Spock Framework

https://github.com/spockframework/spock

http://docs.spockframework.org/

Example

```
import spock.lang.Specification

class EnrollmentSpec extends Specification {
    def "Should set start date when enrolling new student"() {
        given: "A new student"
        def student = new Student(name: "Joe Bloggs")

        when: "I enroll that student"
        student.enroll()

        then: "Their start year becomes this year"
        student.startYear == new Date()[Calendar.YEAR]
    }
    ...
}
```

Spock test cases

- Must extend `spock.lang.Specification`
- Should have *Spec* suffix
- Must have when + then or expect
- May be documented
- Can be run as JUnit tests

Basic example

Feature method

```
def "Make names all upper case"() {
    given: "The beans exercise"
    def exercise = new GroovyBeans()

    and: "An initial person"
    def person = new Person(firstName: "Joe", lastName: "Bloggs")

    when: "I try to upper cast the names of a given person"
    exercise.namesToUpperCase(person)

    then: "The first and last names are updated appropriately"
    person.firstName == "JOE"
    person.lastName == "BLOGGS"
}
```

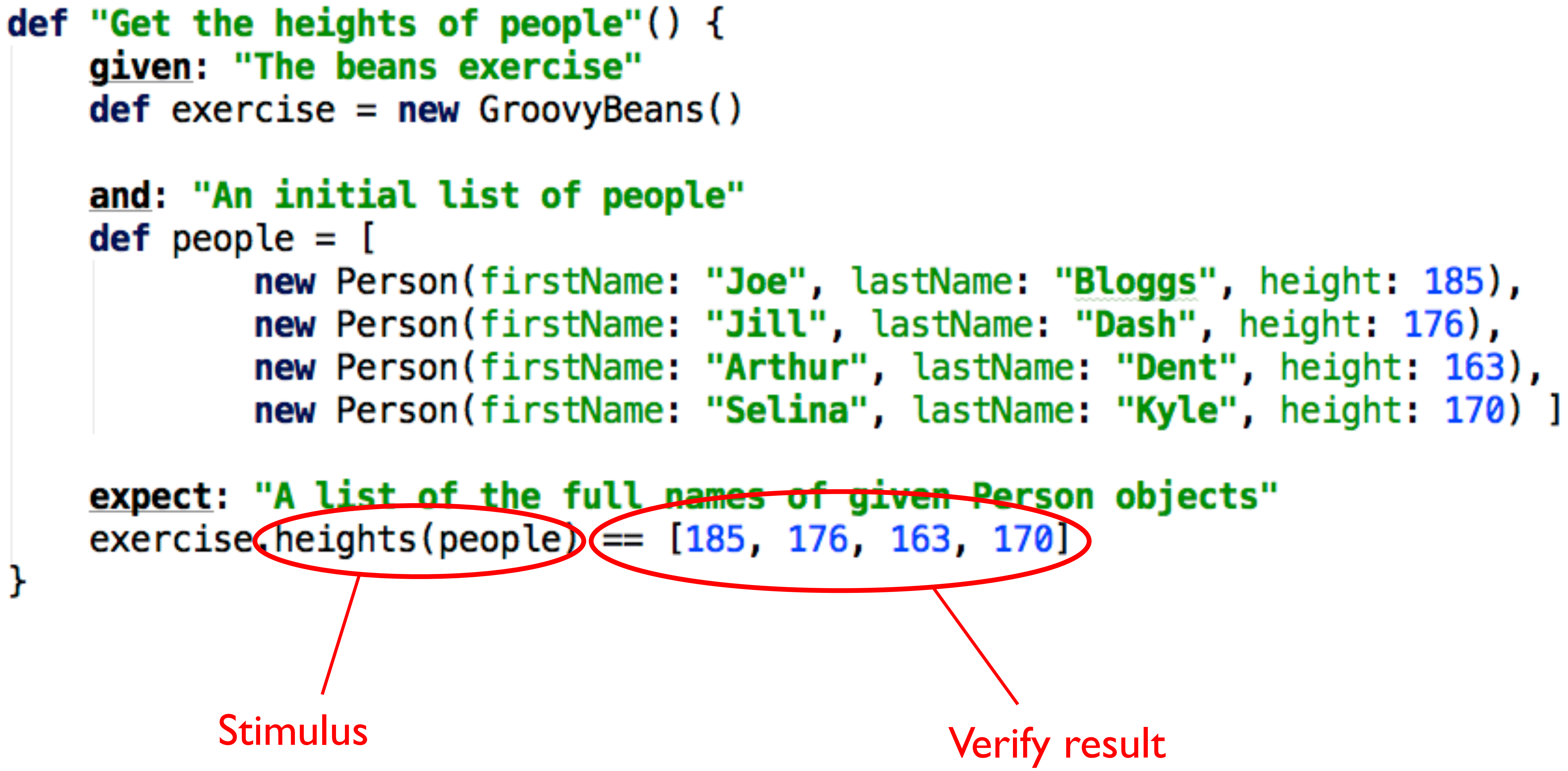
Local variables accessible from when & then blocks

Stimulus

Verify result (implicit assert)

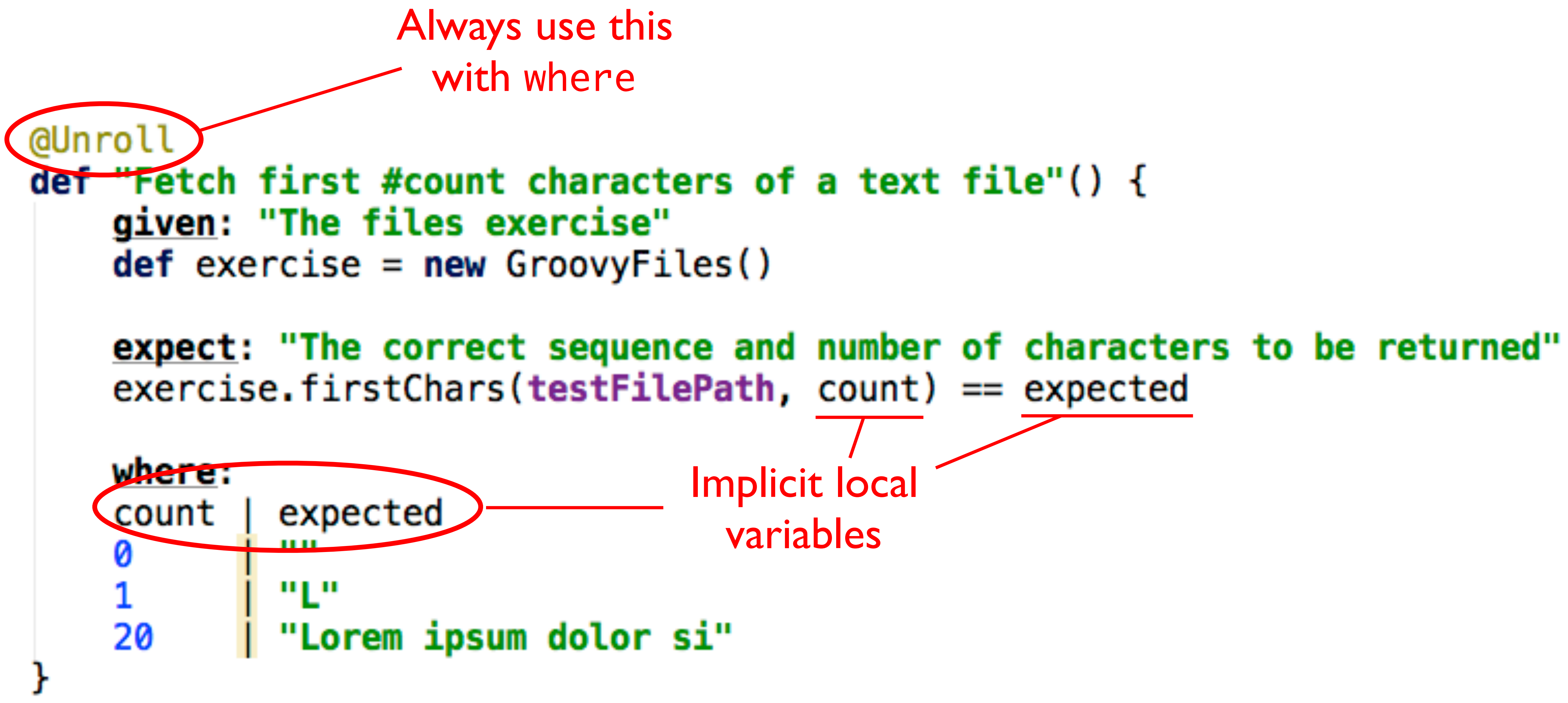
Expect

Combined when & then



```
def "Get the heights of people"() {
    given: "The beans exercise"
    def exercise = new GroovyBeans()

    and: "An initial list of people"
    def people = [
            new Person(firstName: "Joe", lastName: "Bloggs", height: 185),
            new Person(firstName: "Jill", lastName: "Dash", height: 176),
            new Person(firstName: "Arthur", lastName: "Dent", height: 163),
            new Person(firstName: "Selina", lastName: "Kyle", height: 170) ]

    expect: "A list of the full names of given Person objects"
    exercise.heights(people) == [185, 176, 163, 170]
}
```

Data-driven tests



```
@Unroll
def "Fetch first #count characters of a text file"() {
    given: "The files exercise"
    def exercise = new GroovyFiles()

    expect: "The correct sequence and number of characters to be returned"
    exercise.firstChars(testFilePath, count) == expected

    where:
    count | expected
    0     | ""
    1     | "L"
    20    | "Lorem ipsum dolor si"
}
```

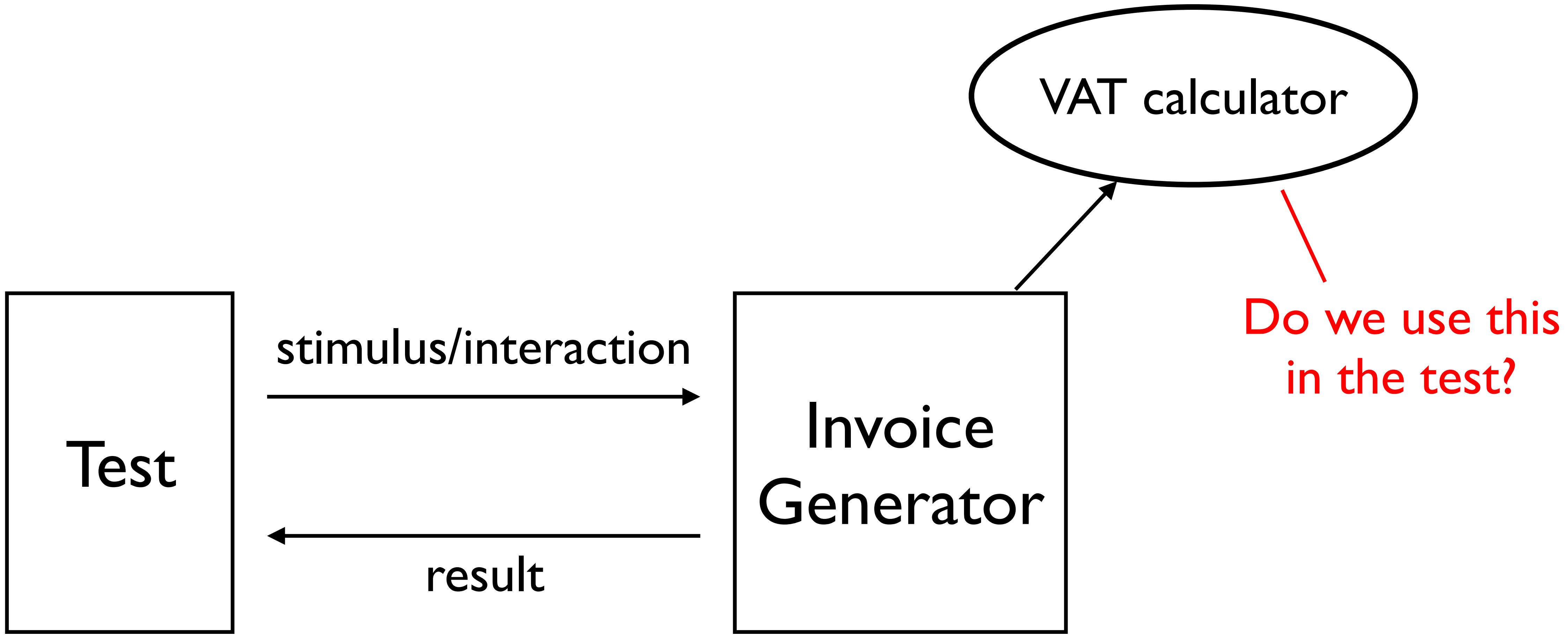
Testing exceptions

```
def "Handle errors when calculating the byte size of a file"() {
    given: "The exceptions exercise"
    def exercise = new GroovyExceptions()

    when: "I try to find the size of a null or empty path"
    exercise.characterCount(testFilePath)

    then: "The appropriate exception is thrown"
    def ex = thrown(IllegalArgumentException)
    ex.message == "Path is null or empty: '${value}'"

    where:
    testFilePath | value
    null         | 'null'
    ""           | ''
}
```

Expect exception of particular type

Mocks

Collaborators

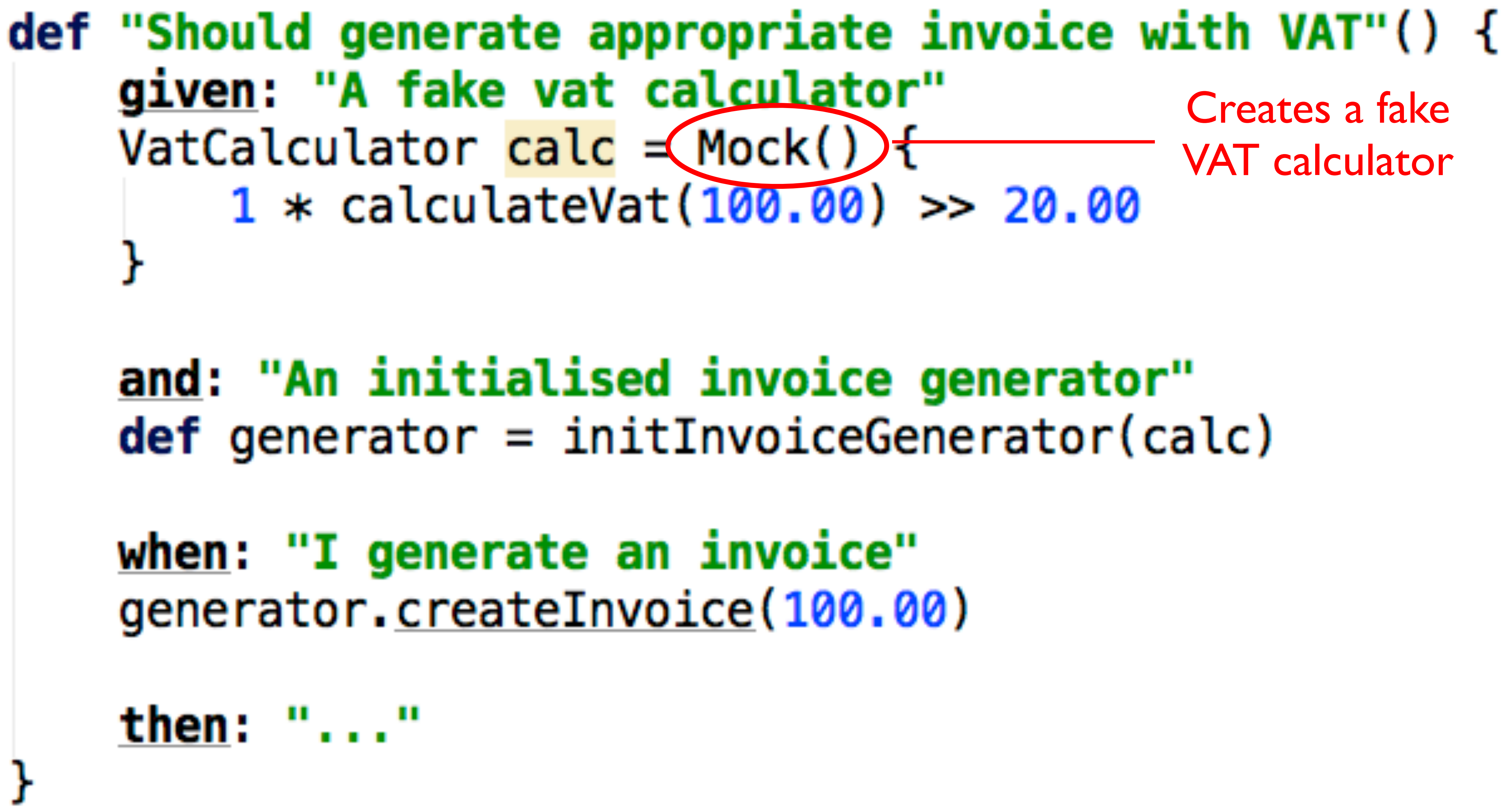


For unit tests

- Collaborators shouldn’t interact with the environment (file system, databases, etc.)
- Bugs in collaborator shouldn’t affect the test case

Use fake objects!

Mocking in Spock



```
def "Should generate appropriate invoice with VAT"() {
    given: "A fake vat calculator"
    VatCalculator calc = Mock() {
        1 * calculateVat(100.00) >> 20.00
    }

    and: "An initialised invoice generator"
    def generator = initInvoiceGenerator(calc)

    when: "I generate an invoice"
    generator.createInvoice(100.00)

    then: "..."
}
```

Creates a fake VAT calculator

Guidelines

- Mocking concrete types is hard
  - prefer interfaces
- Abstract out environmental interaction
  - put file system and DB access behind a few interfaces
- Potentially leave out explicit types if it makes for easier testing

Mocks vs stubs

Do you care which collaborator methods are called?

Do you care in which order or how many times?

Do you care what arguments are passed in?

Mocks vs stubs

You need a mock!

Mocks vs stubs

Otherwise a stub will do

Mocks vs stubs

- Mocks verify interactions
- Mocks lead to fragile tests
  - internal refactoring may change interactions
- Stubs don’t care about the interactions
- Favour stubs over mocks where possible

Caution

If your test mostly involves setting up mock objects and there isn't much logic in the method under test, skip the unit test and make sure your code is covered by a higher level test.

Caution

If tests aren’t easy to write, they won’t get written.

http://spockframework.github.io/spock/docs/1.0/interaction_based_testing.html