Simplicity *itself*

Developer skills

Language-independent skills

Version control (with git)

Test Driven Development (TDD)

Building software

Continuous Integration

Language-independent skills

Version control (with git)

Test Driven Development (TDD)

Building software

Continuous Integration

VCS - Version Control System

SCM - Source Control Management

Different acronyms, basically same thing

Why?

- Know what changes were made when
- Keep source code used for a release
- Never lose changes
- Enable collaboration

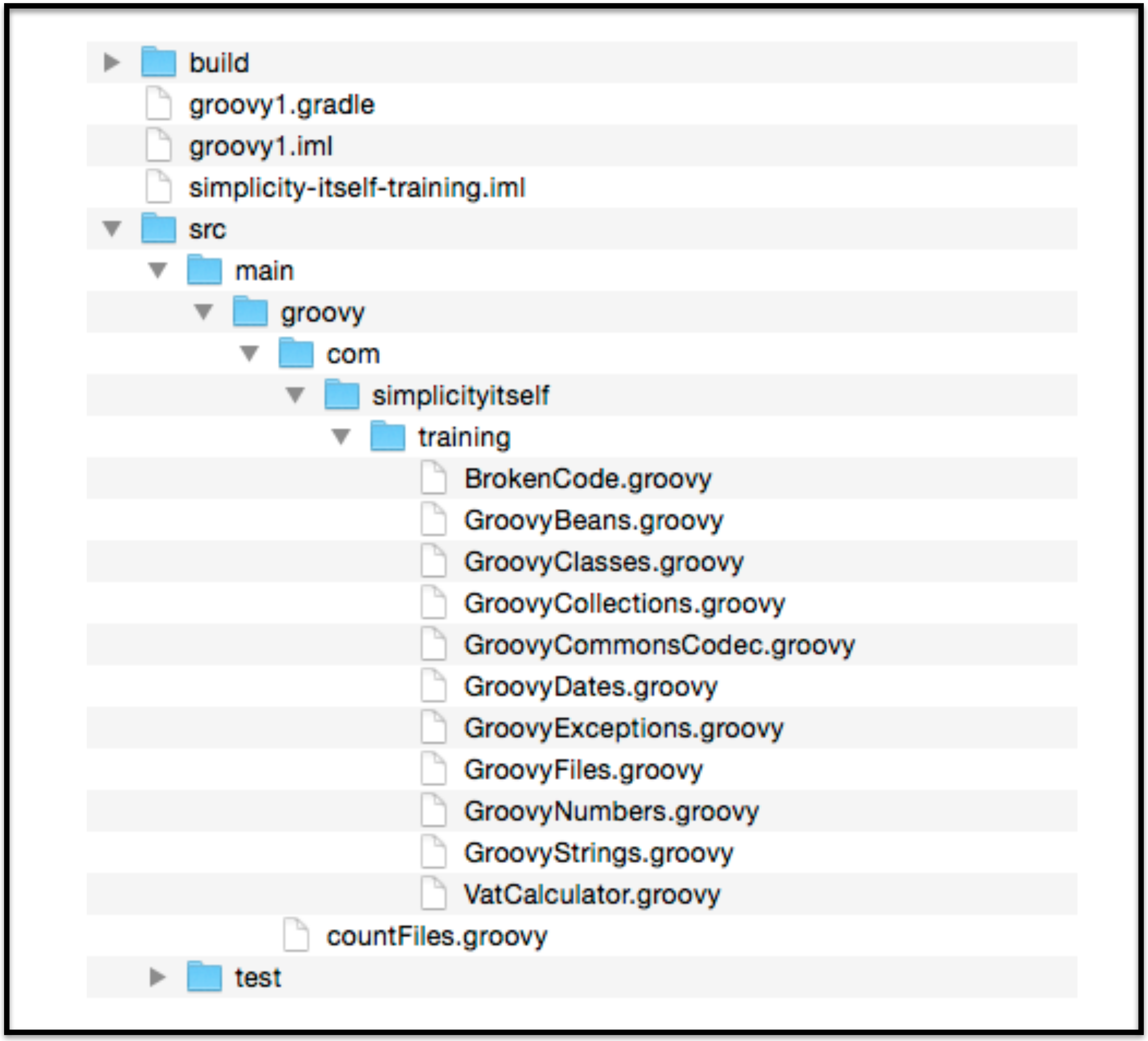
git - a *distributed* VCS

There is no central server that contains the history and authoritative state of the code base

The git model

Repositories

Repo



```
▶ build
  groovy1.gradle
  groovy1.iml
  simplicity-itself-training.iml
▼ src
  ▼ main
    ▼ groovy
      ▼ com
        ▼ simplicityitself
          ▼ training
              BrokenCode.groovy
              GroovyBeans.groovy
              GroovyClasses.groovy
              GroovyCollections.groovy
              GroovyCommonsCodec.groovy
              GroovyDates.groovy
              GroovyExceptions.groovy
              GroovyFiles.groovy
              GroovyNumbers.groovy
              GroovyStrings.groovy
              VatCalculator.groovy
        countFiles.groovy
  ▶ test
```

Repositories

- Contain the source code
- + history
- + branches
- + tags

Repositories

Each repository is self-contained!

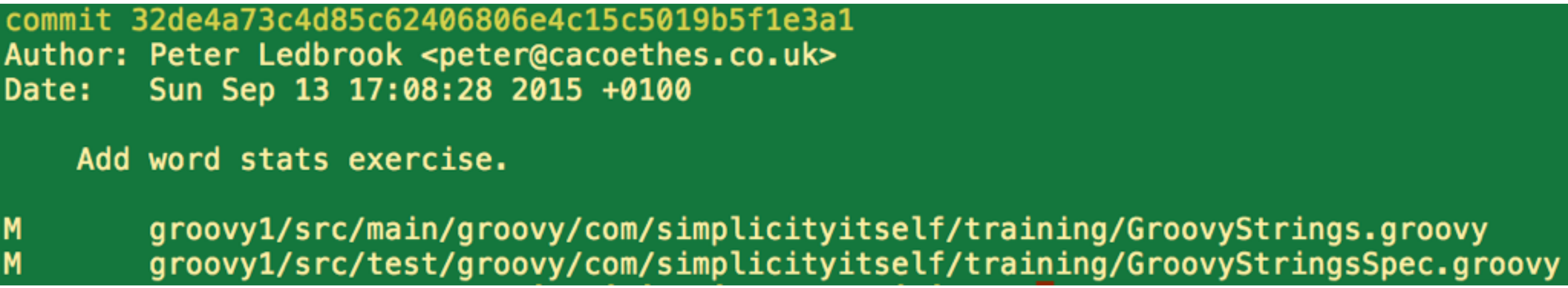
Create a repository

```
git init .
```

Doesn’t track files until you add them explicitly

Commits

Has an ID



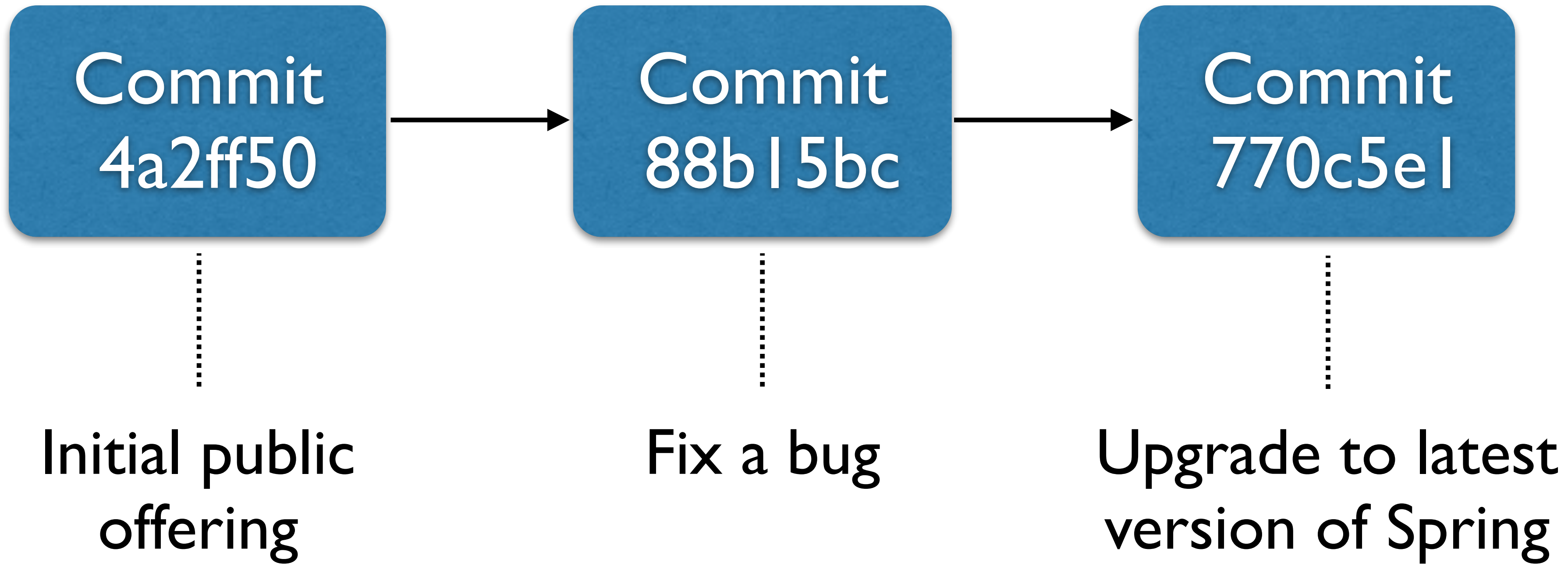
```
commit 32de4a73c4d85c62406806e4c15c5019b5f1e3a1
Author: Peter Ledbrook <peter@cacoethes.co.uk>
Date:   Sun Sep 13 17:08:28 2015 +0100

    Add word stats exercise.

M       groovy1/src/main/groovy/com/simplicityitself/training/GroovyStrings.groovy
M       groovy1/src/test/groovy/com/simplicityitself/training/GroovyStringsSpec.groovy
```

Consists of a set of changes
- a *changeset*

Commits

A commit is a *remembered* set of changes in the repository

Commit history



Every commit has a unique ID

Commits

- Contain
  - added/removed files
  - changes to files
- Do not contain
  - directory changes unless it affects files
- *Empty directories are effectively invisible to git*

How do you create commits?

File status in repository

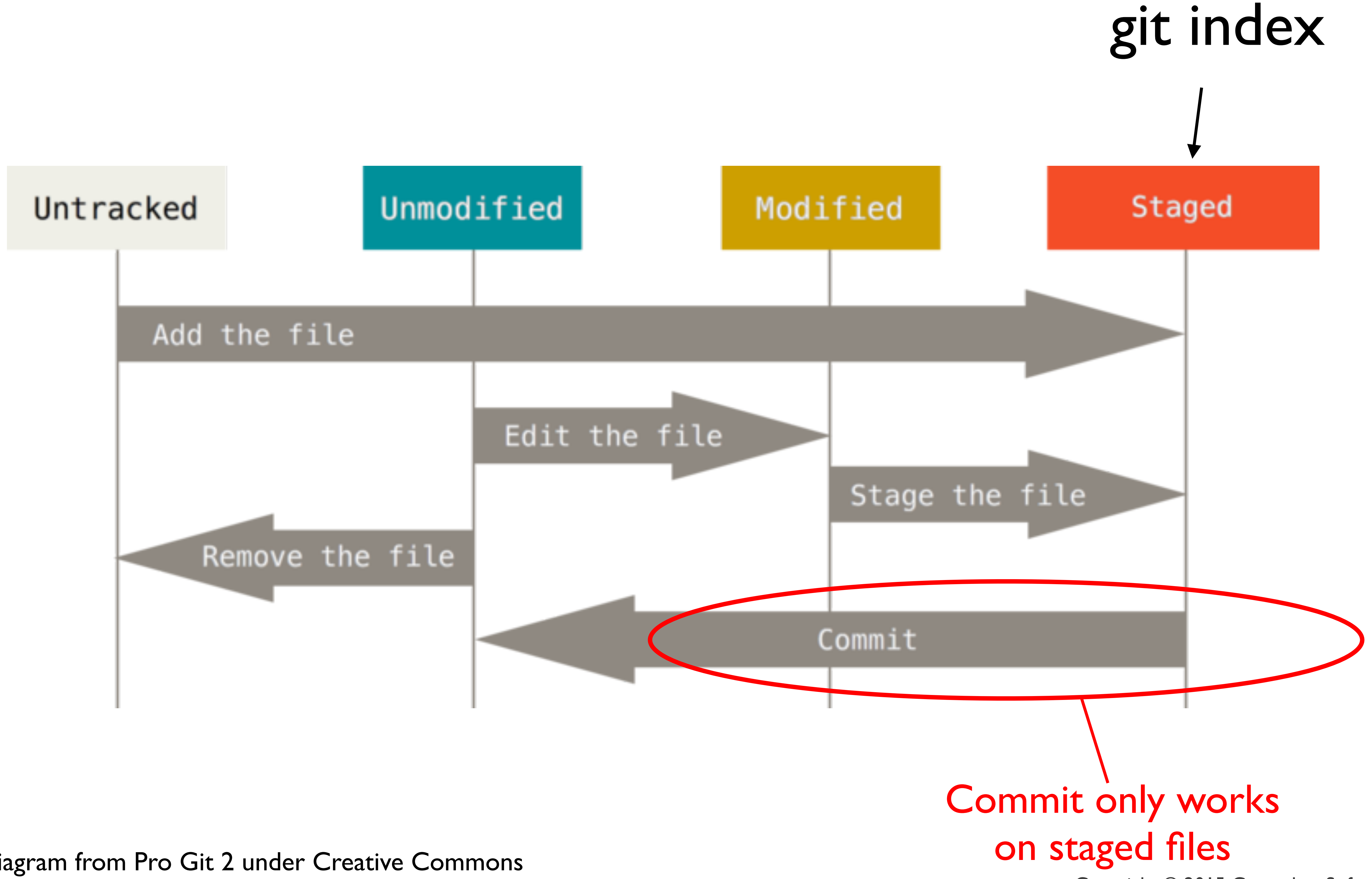
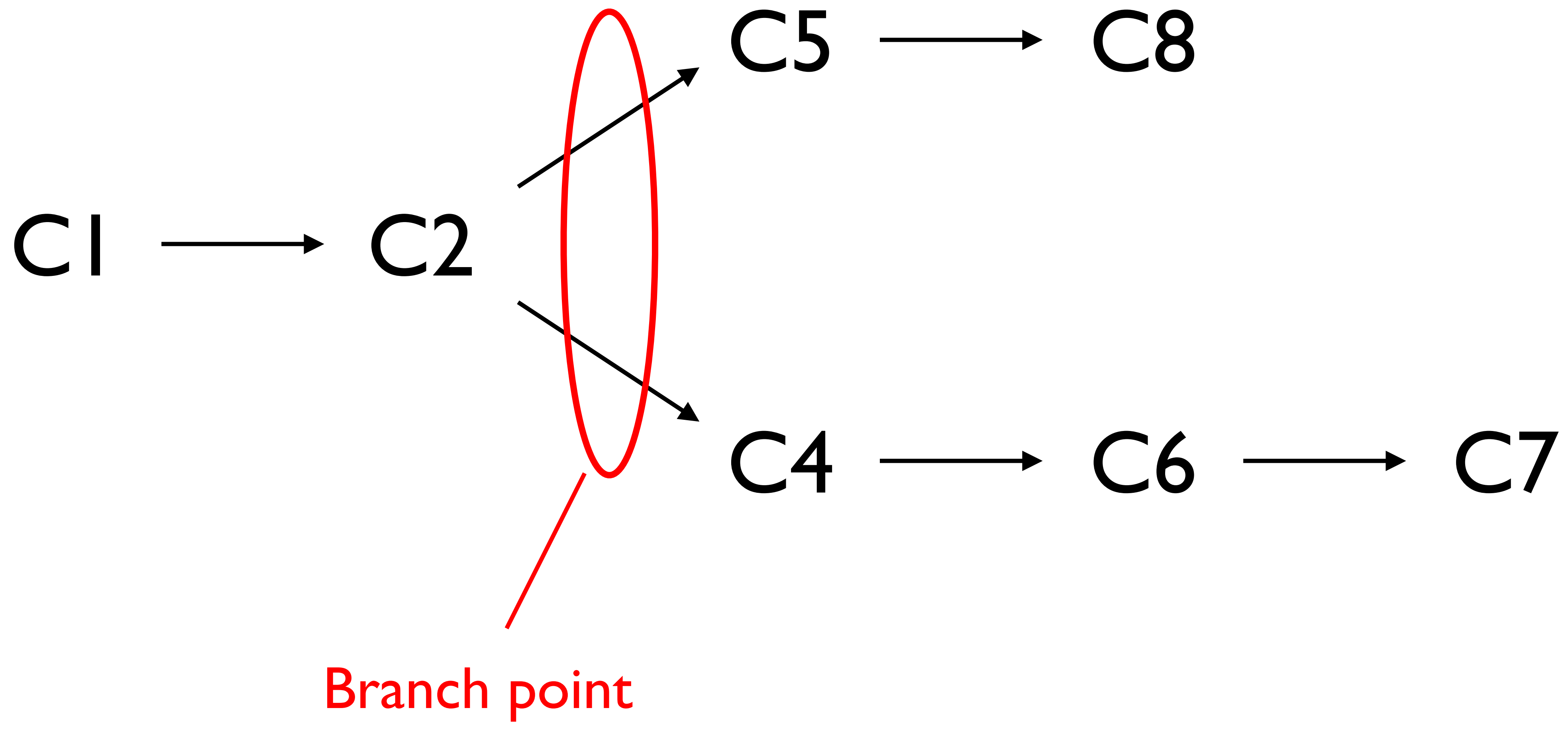


Diagram from Pro Git 2 under Creative Commons

Branches

Commit trees



A branch is a commit “pointer”

Branches

$$\begin{tikzcd}
 & & & \text{bug-fix} & & \\
 & & C5 \arrow[r] & C8 \arrow[u, dash] & & \\
C1 \arrow[r] & C2 \arrow[ur] \arrow[dr] & & & & \\
 & & C4 \arrow[r] \arrow[d, dash] & C6 \arrow[r] & C7 \arrow[d, dash] \arrow[dr, dash] & \\
 & & \text{v1.0.1 (tag)} & & \text{HEAD} & \text{master}
\end{tikzcd}$$

Branches

$$\begin{tikzcd}
 & & & \text{bug-fix} \arrow[d, dash] & & \\
 & & C5 \arrow[r] & C8 \arrow[r] & C9 & \\
C1 \arrow[r] & C2 \arrow[ur] \arrow[dr] & & & & \\
 & & C4 \arrow[r] \arrow[d, dash] & C6 \arrow[r] & C7 \arrow[d, dash] \arrow[dr, dash] & \\
 & & \text{v1.0.1 (tag)} & & \text{HEAD} & \text{master}
\end{tikzcd}$$

Branches

$$\begin{tikzcd}
& & & & \text{bug-fix} \arrow[d, dash] & \\
& & C5 \arrow[r] & C8 \arrow[r] & C9 & \\
C1 \arrow[r] & C2 \arrow[ur] \arrow[dr] & & & & \\
& & C4 \arrow[r] \arrow[d, dash] & C6 \arrow[r] & C7 \arrow[d, dash] \arrow[dr, dash] & \\
& & \text{v1.0.1 (tag)} & & \text{HEAD} & \text{master}
\end{tikzcd}$$

Guidelines

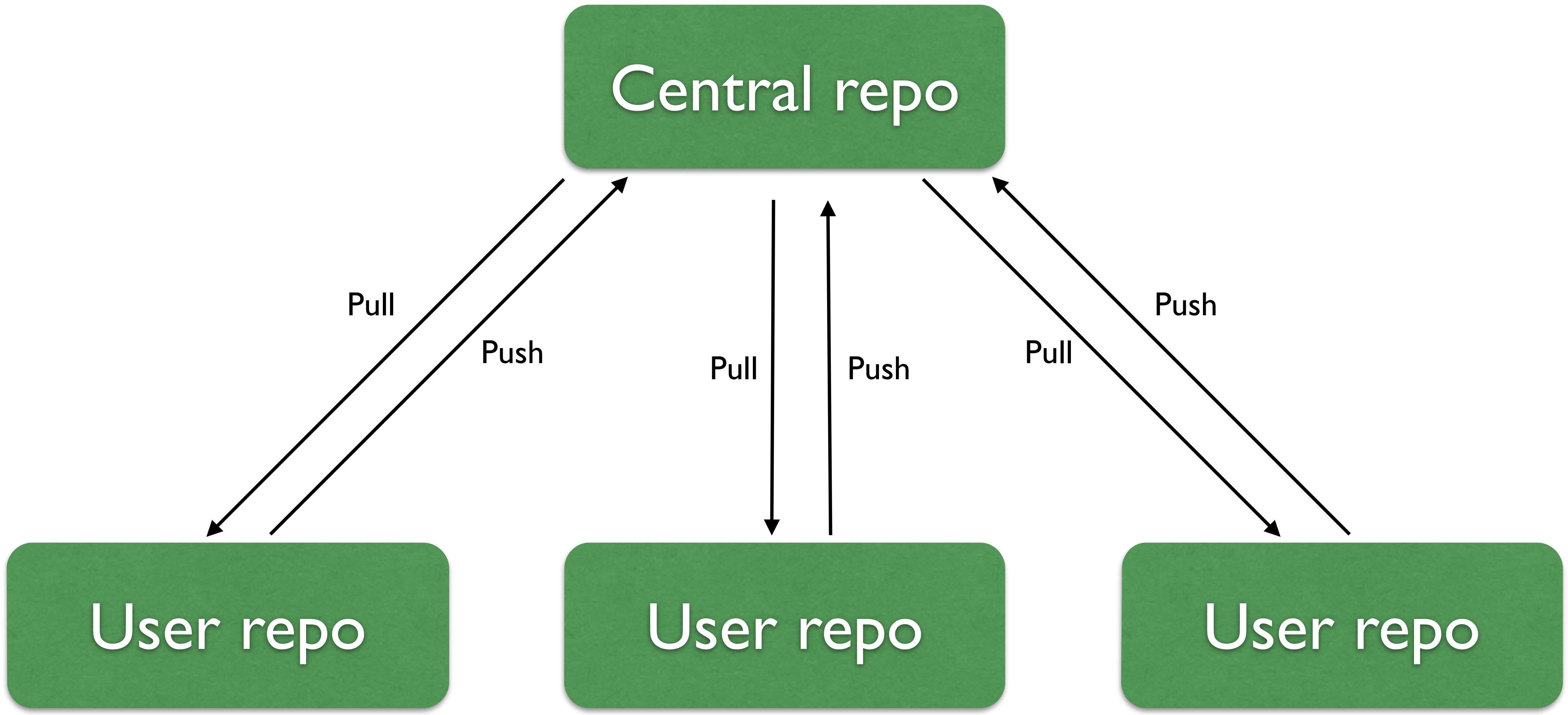
- "default" branch is called *master*
- Branches are cheap, use them!
- Use them to commit frequently
- Your branches are private to your repo
- …unless you publish them
- Branches can be merged with one another

Merge

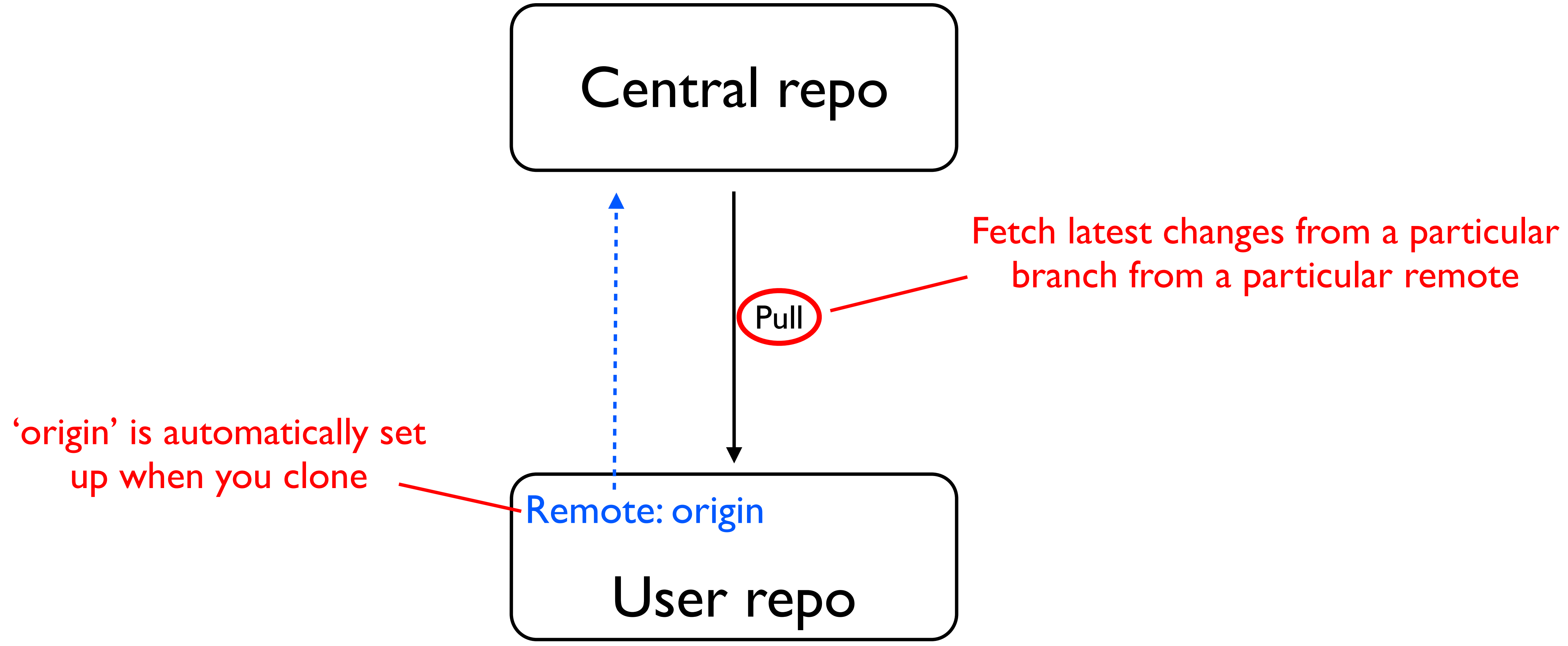
$$\begin{tikzcd}
 & & & & \text{bug-fix} \arrow[d, dash] & \\
 & & C5 \arrow[r] & C8 \arrow[r] & C9 \arrow[ddr] & \\
C1 \arrow[r] & C2 \arrow[ur] \arrow[dr] & & & & \text{Merge commit} \arrow[d, dash] \\
 & & C4 \arrow[r] & C6 \arrow[r] & C7 \arrow[r] \arrow[d, dash] \arrow[dr, dash] & C10 \\
 & & & & \text{HEAD} & \text{master}
\end{tikzcd}$$

Between repositories

Widely used structure



Remotes


Central repo
Pull
Fetch latest changes from a particular branch from a particular remote
'origin' is automatically set up when you clone
Remote: origin
User repo

You can define many remotes

Before you start

Configuration

```
git config --global user.name "John Doe"

git config --global user.email johndoe@example.com

git config --global alias.co checkout

git config --global alias.br branch

git config --global alias.st status

git config --global alias.ci commit
```

Configuration

.gitignore (root of repository)

```
build/
out/
*.iws
*.ipr
*.iml
```

Workflows

Starting

Create a new local repository:

```
git init .

git add .

git commit -m "Initial public offering"
```

Required when using a central repository

Copy an existing repository:

```
git clone <url> <dir>
```

Can be http:, https:, or git with ssh

Local development

Start a local dev branch:

Name of new branch

Where our branch starts

```
git checkout -b dev master
```

Check you are working on the right branch:

```
git branch
```

Local development

After saving local edits, check file status:

```
git status
```

Add the changes you want commited to the index:

```
git add [--patch] <file path>
```

Interactively add only some of the changes in a file to the commit

Can be a directory path too (add all untracked and modified files in that dir)

Local development

Commit those changes:

```
      git commit -m <commit message>
or    git commit -F <file>
or    git commit
```

Opens configured editor so you can
write a longer commit message

Repeat local development flow for each commit

Synchronise to remote

Switch to master branch:

```
git checkout master
```

Fetch and merge any changes others have published:

```
git pull
```

Merge changes into your dev branch:

```
git checkout dev
git merge master
```

If you're comfortable with rebasing, use `git rebase master`

Publish changes

Synchronise first!

Switch to master branch and merge your changes:

```
git checkout master
git merge dev
```

Publish your changes:

```
git push
```

Remote branches

Create new local branch:

```
git checkout -b featureA master
```

Push and track:

```
git push -u origin featureA
```

Sync with remote branch:

```
git checkout featureA
git pull
git push
```

Earlier `-u ` means these don't need arguments

Information

What changes are in a given commit?

```
git show <commit ref>
```

What's the difference between two branches or tags?

```
git diff <branch|tag> [<branch|tag>]
```

Default is current branch

What changes are in the index?

```
git diff --cached
```

What's the commit history look like?

```
git log
```

Oops, …

Stuff in the index you don’t want to commit?

```
git reset [--hard]
```

Dangerous! You lose all changes.

Revert local changes to files?

```
git checkout <file>
```

Doesn’t work for files in the index. Use `reset` instead.

Want to tidy up commits?

```
git rebase -i <commit ref>
```

Do not use on any commits that have been published!

Resources

http://sixrevisions.com/resources/git-tutorials-beginners/

http://git-scm.com/book/

https://training.github.com/kit/downloads/github-git-cheatsheet.pdf

Play time!

Language-independent skills

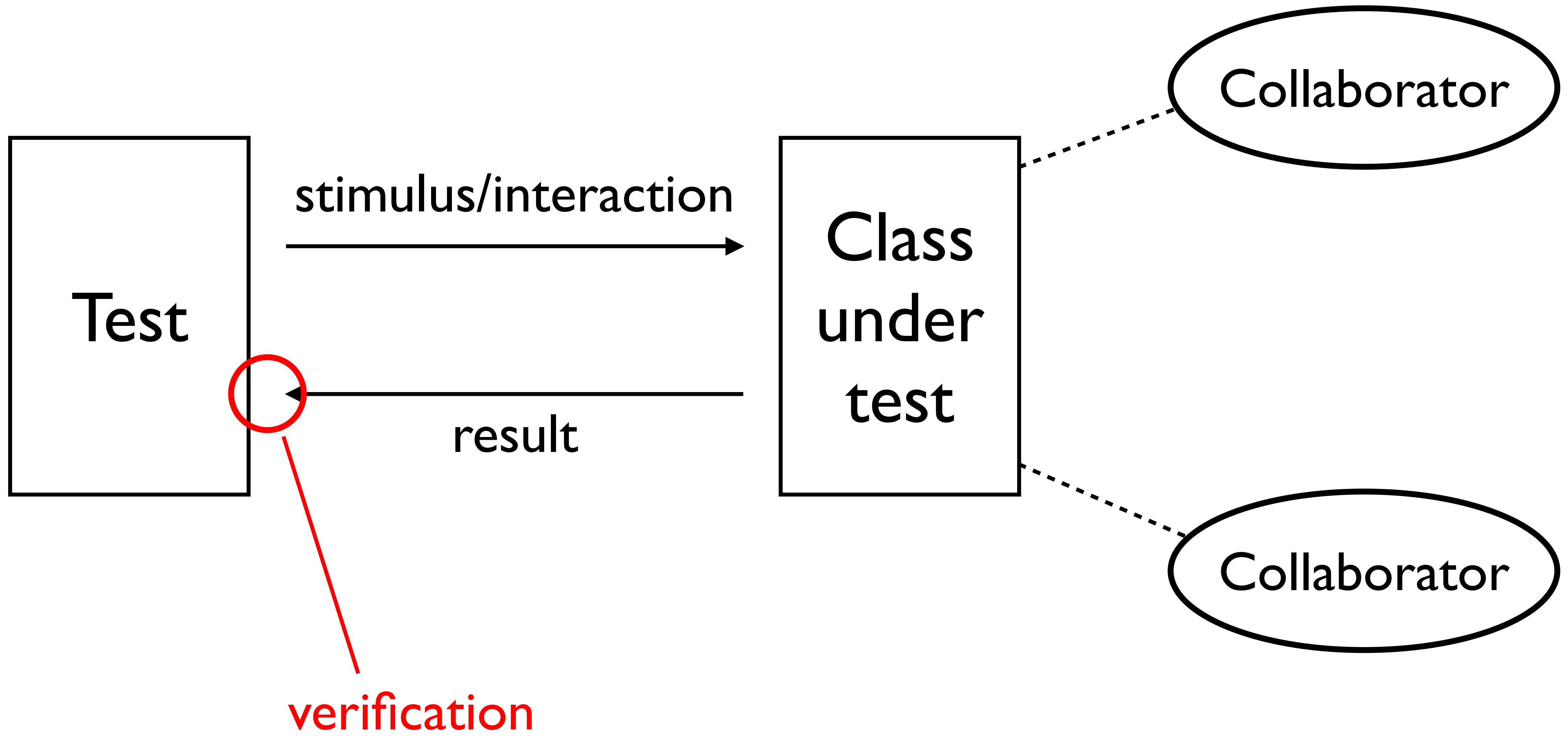
Version control (with git)

Test Driven Development (TDD)

Building software

Continuous Integration

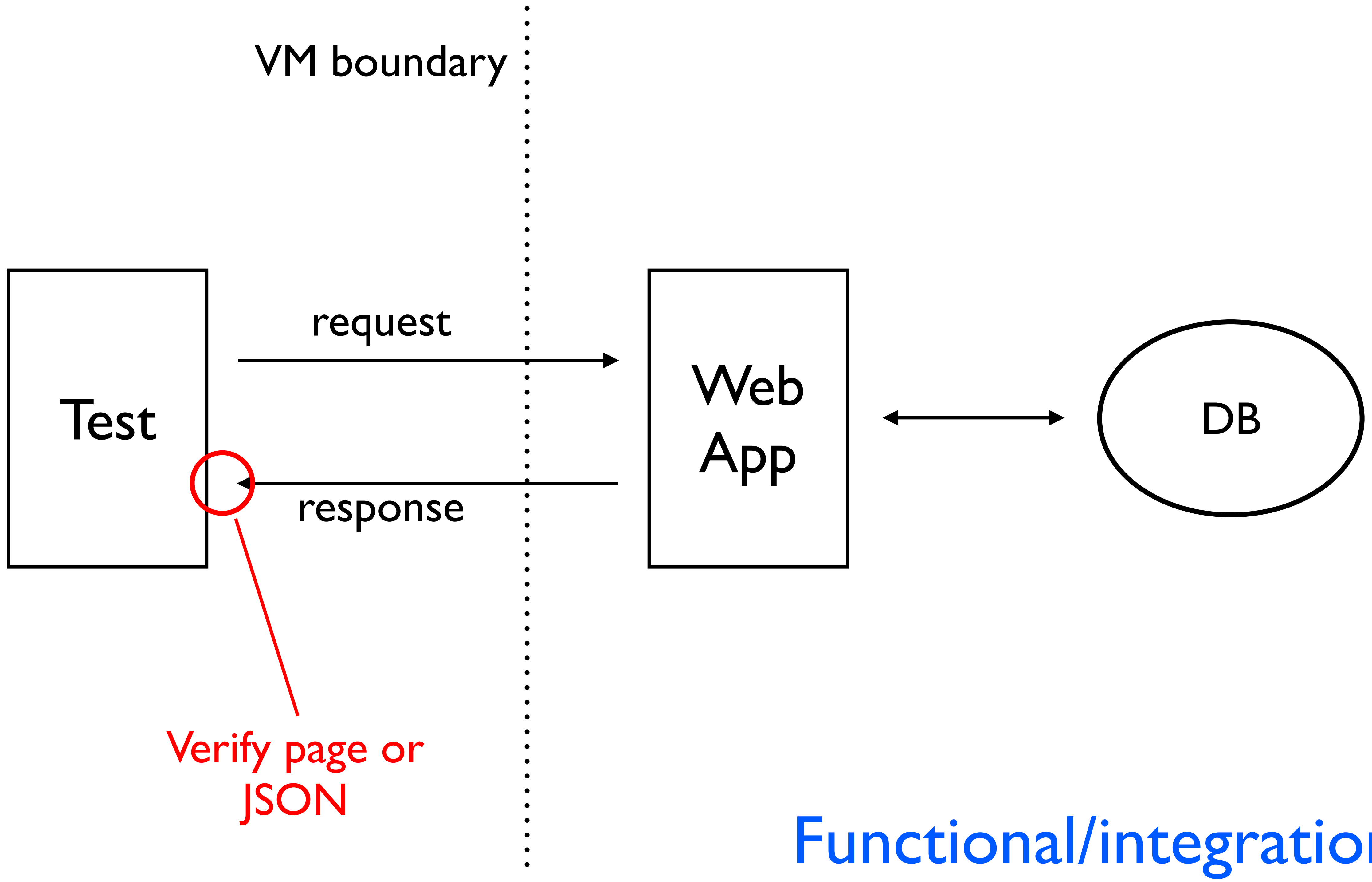
Tests


Test
stimulus/interaction
result
Class under test
Collaborator
Collaborator
verification

Tests give you

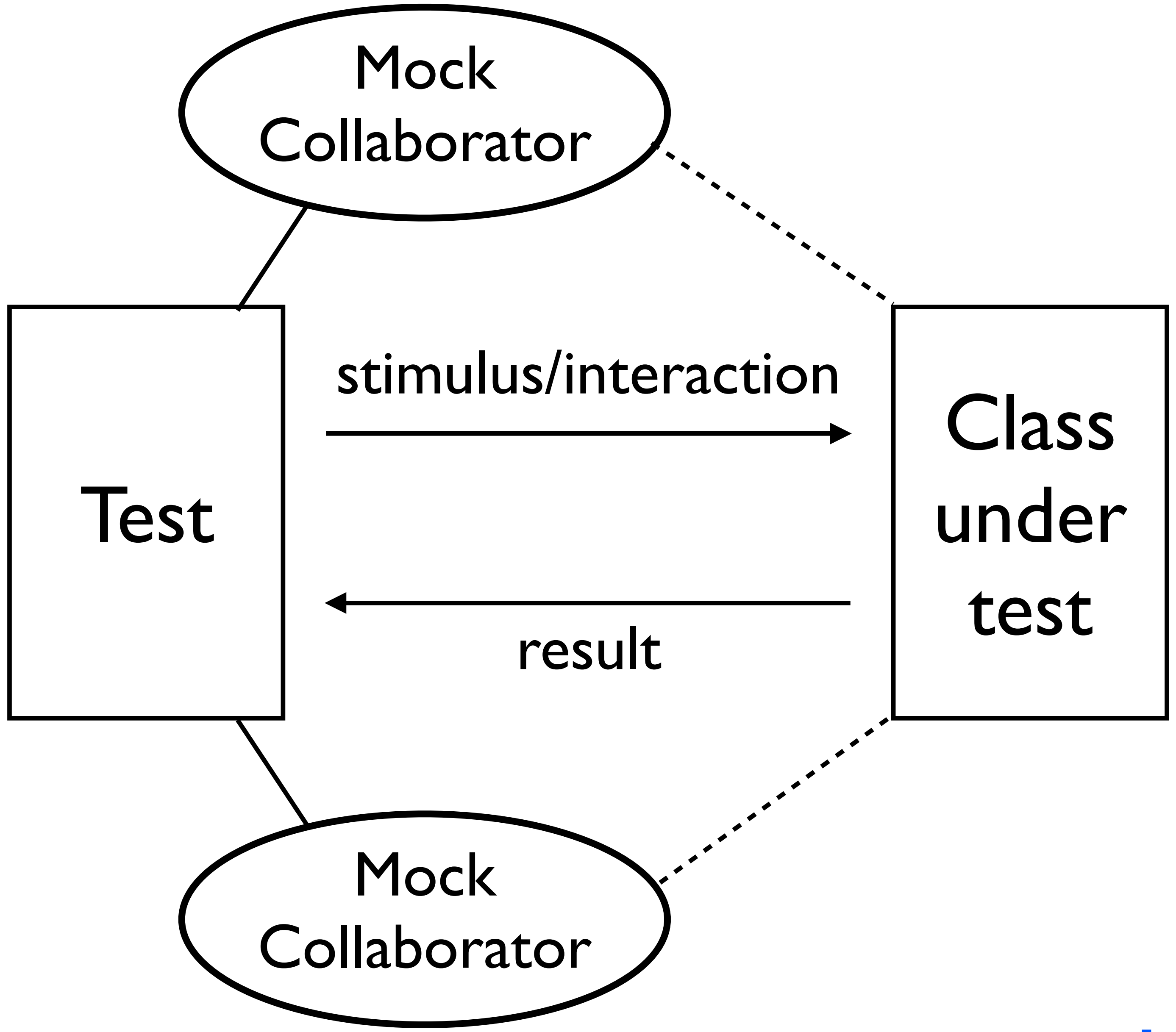
- Software reliability
- Confidence
- Safety when refactoring
- A codified specification

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

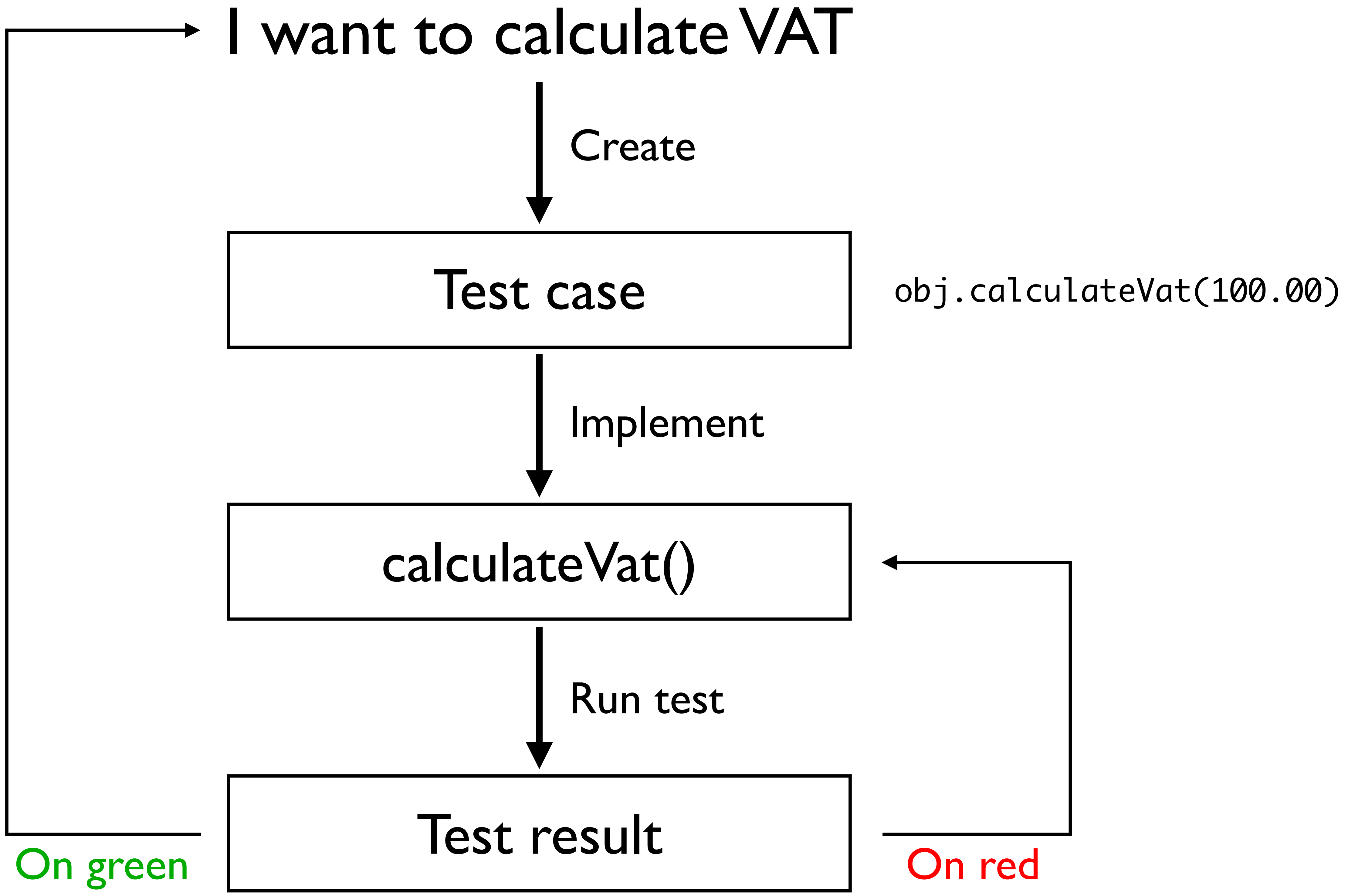
Practise, practise, practise

Test Driven Development

TDD gives you

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

Example



```
obj.calculateVat(100.00)
```

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"
}
```

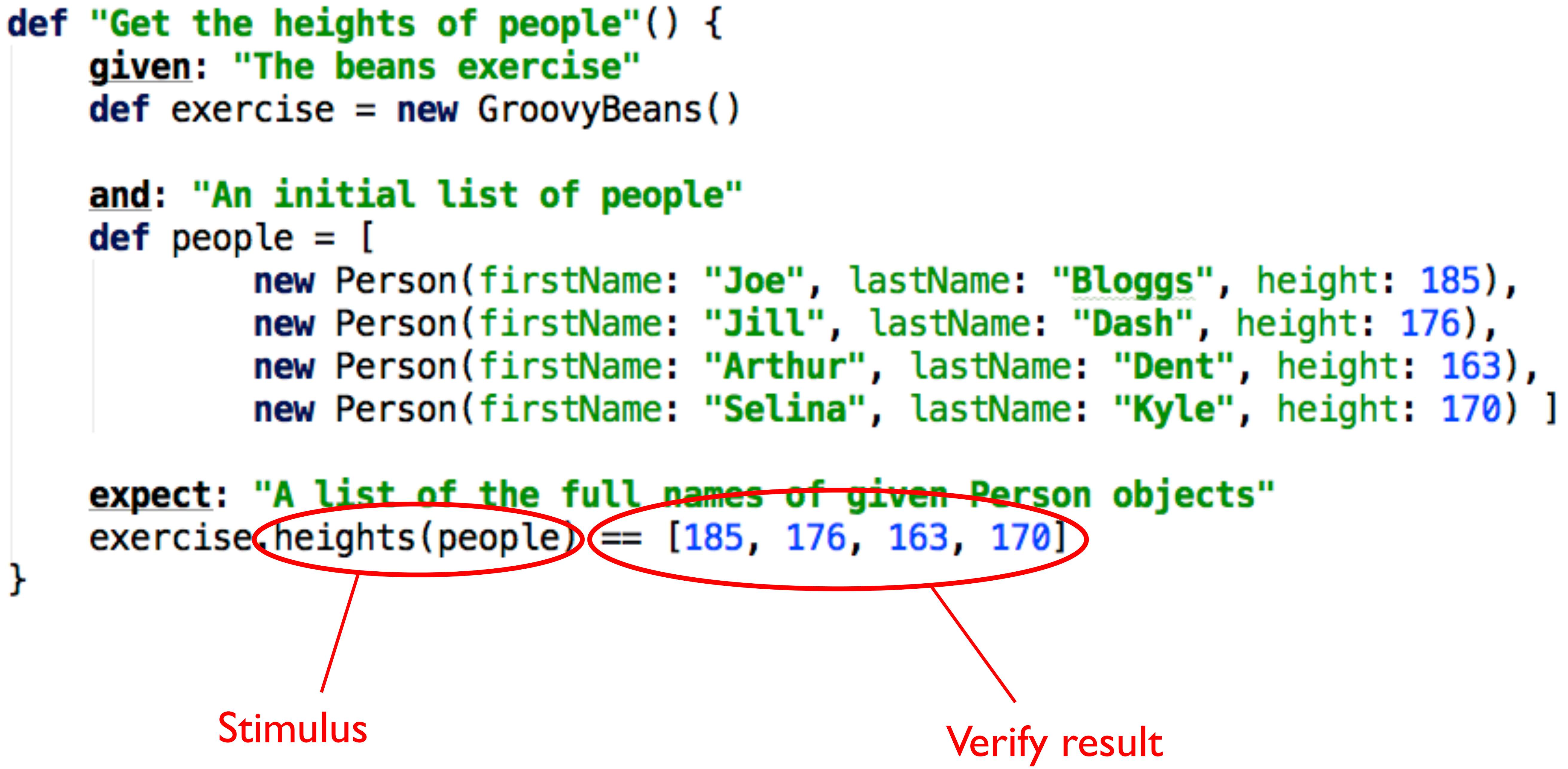
Local variables accessible from when & then blocks

Stimulus

Verify result (implicit assert)

Expect

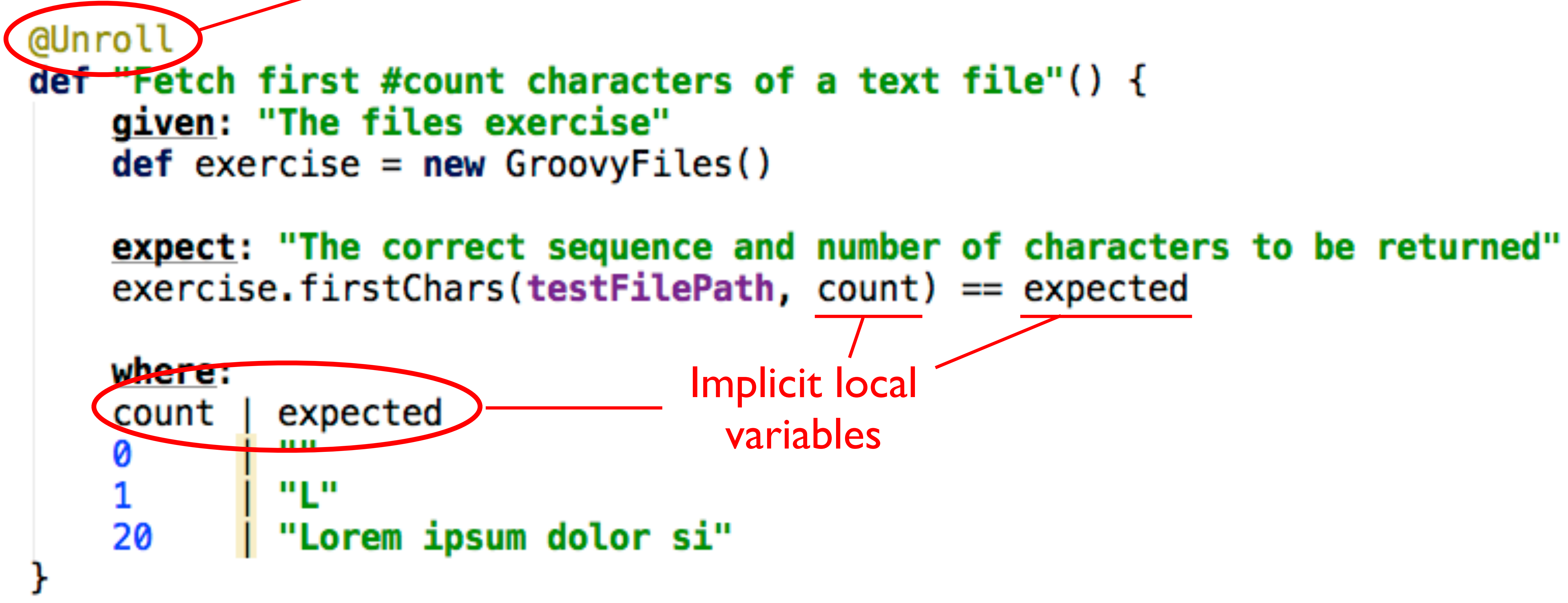
Combined when & then



```
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

Always use this with where



```
@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"
}
```

Implicit local variables

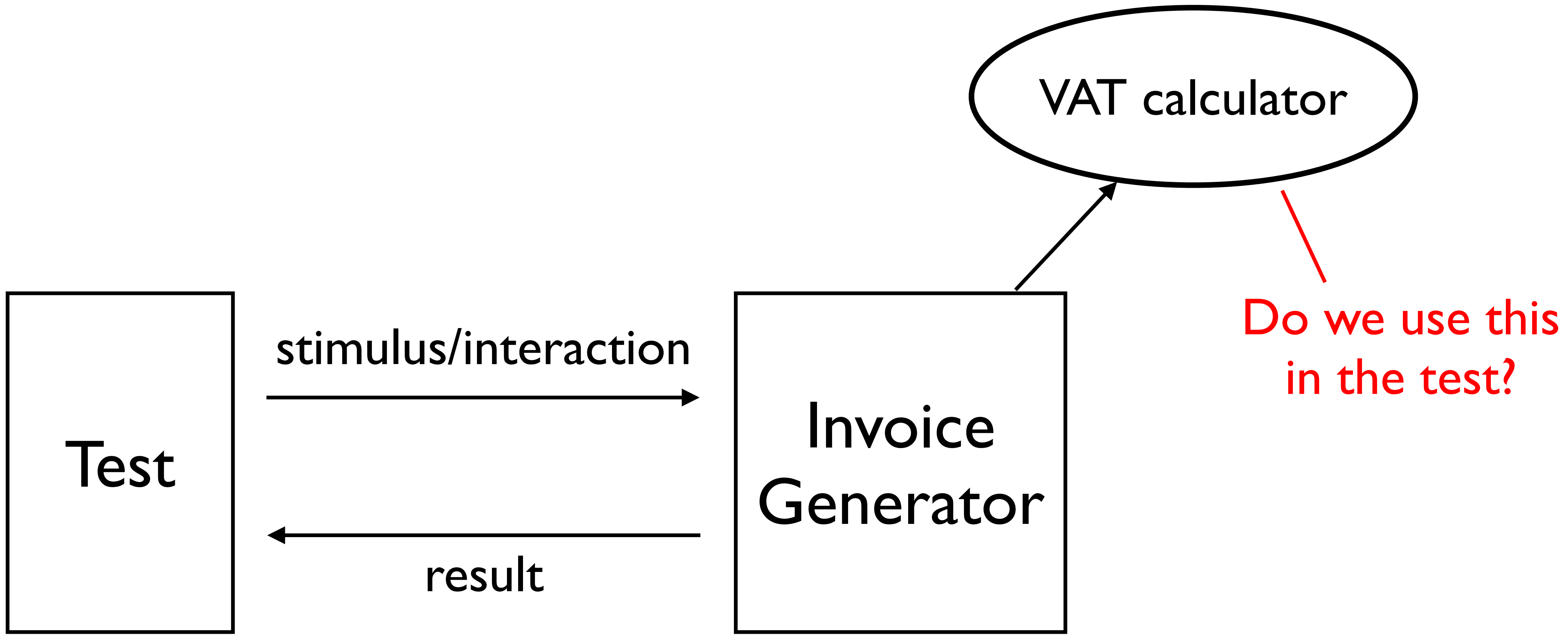
Testing exceptions

```groovy
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

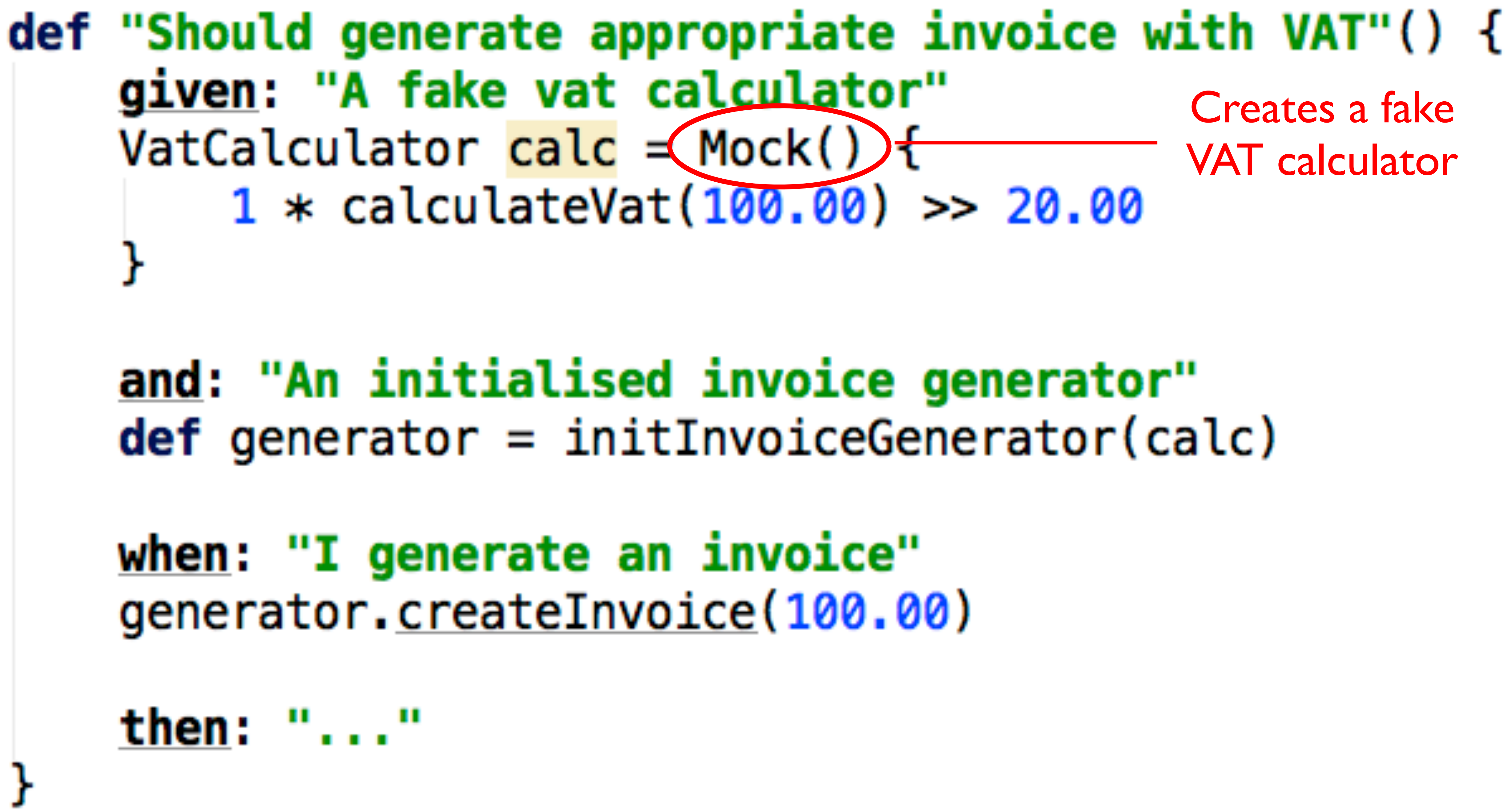


For unit tests

- 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

Play time!

Language-independent skills

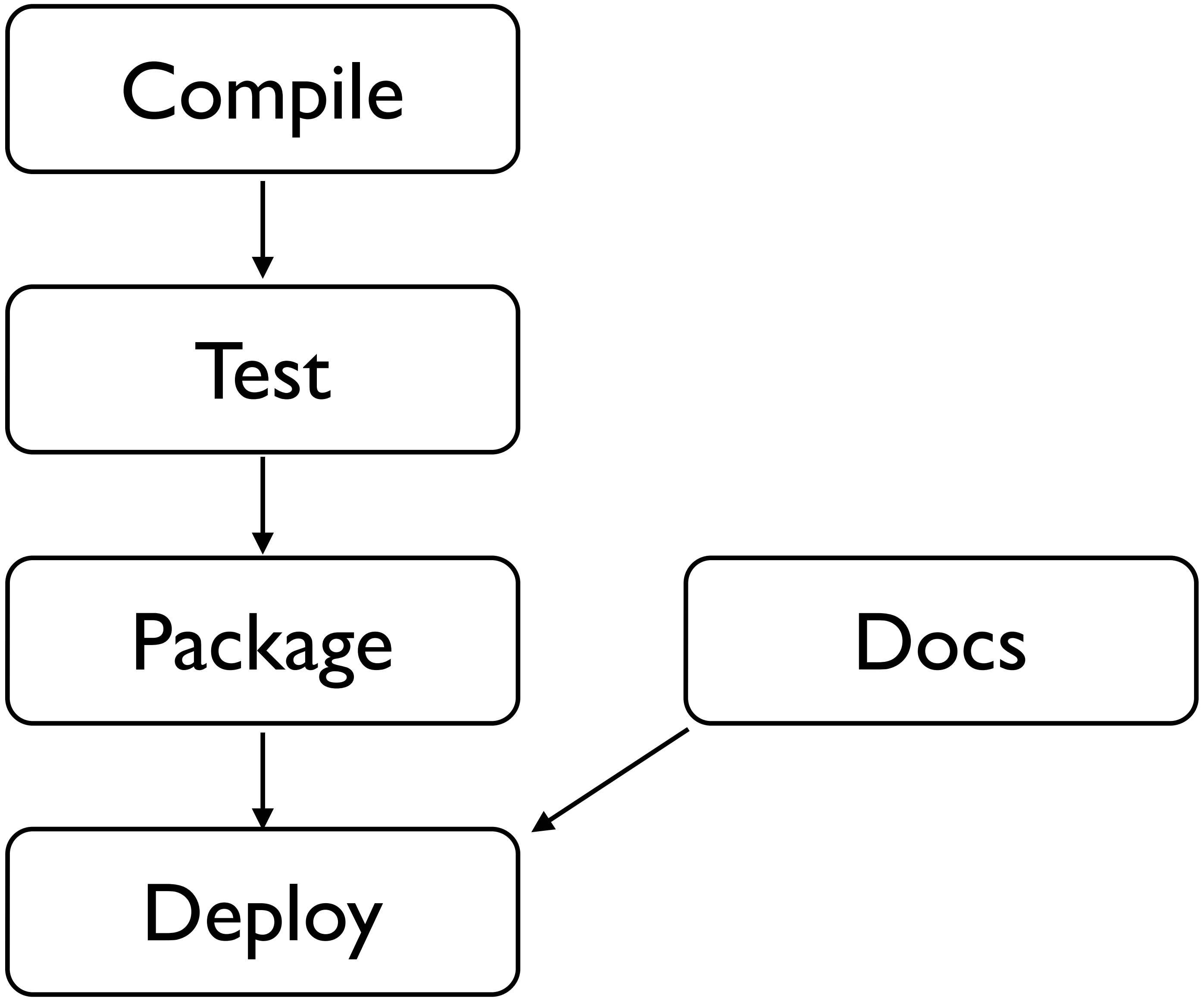
Version control (with git)

Test Driven Development (TDD)

Building software

Continuous Integration

Building software is a process



Humans are error-prone

Automate for

- Faster process
- Better reproducibility
- Fewer mistakes
- Greater confidence

Build tools are designed to do this job

Examples on the JVM

- Apache Ant
- Apache Maven
- Sbt
- Gradle
- Grails 1.x/2.x

Language-independent skills

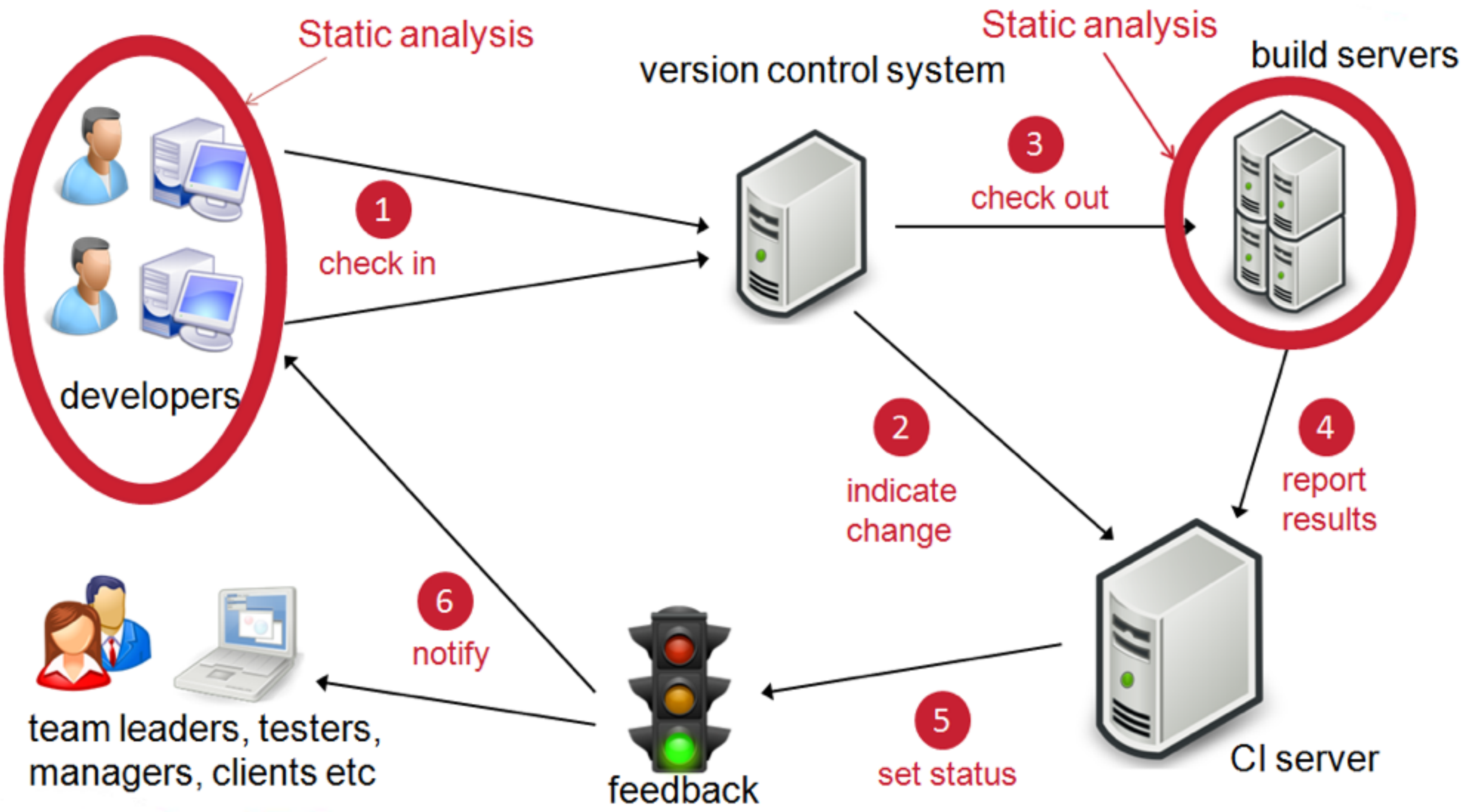
Version control (with git)

Test Driven Development (TDD)

Building software

Continuous Integration

CI



Static analysis
version control system
Static analysis
build servers
1
check in
3
check out
developers
2
indicate change
4
report results
6
notify
5
set status
team leaders, testers, managers, clients etc
feedback
CI server

Why?

- Fix "integration" issues quickly
- Notification of "works for me" issues
- Find out if you broke the build
- "master" should always build
  - or equivalent

Options

- Run against different environments
- VM per build?
- Run different tasks