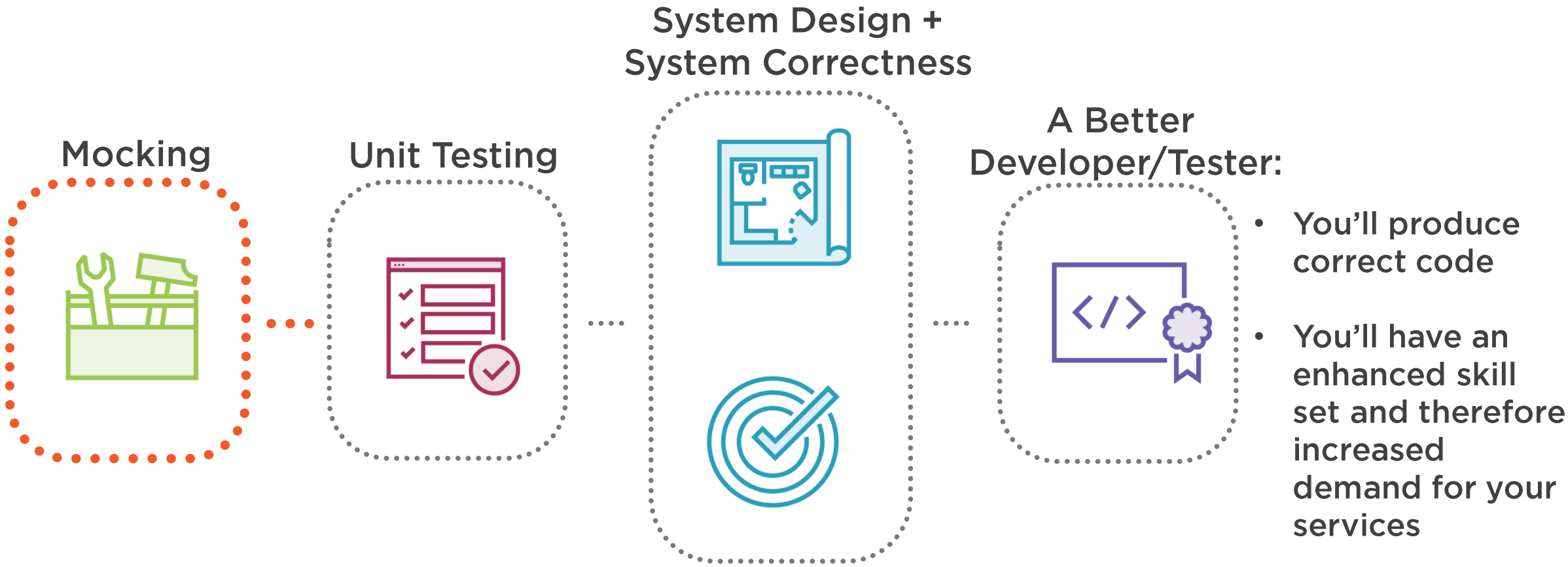
Getting Started with EasyMock

Nicolae Caprarescu
FULL-STACK SOFTWARE DEVELOPMENT CONSULTANT

www.properjava.com

Why Learn How to Mock Using EasyMock?



Mocking gives you the ability to focus on the unit you're trying to test, by giving you full control over the behavior of its dependencies.

Suggested Prerequisites

http://bit.ly/psjunit

or

http://bit.ly/psjunit5

Overall Course Outline

Getting Started with EasyMock

Verifying What Methods are Called Using EasyMock

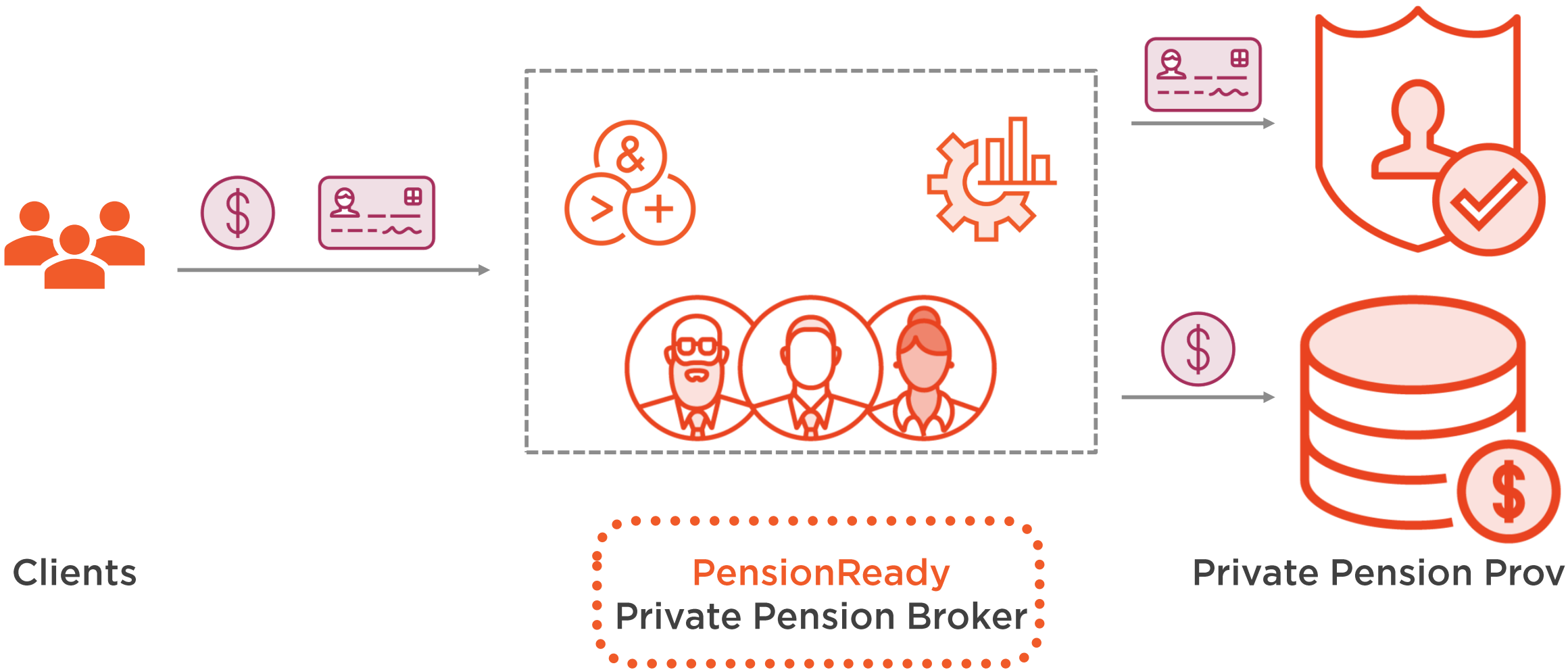
Configuring Behavior of Mocks

Advanced Topics

Module Overview: Getting Started with EasyMock

- Demo Application Overview
- What Is Mocking
- Writing Tests With Mocks 
- Writing Tests Without Mocks 
- Why Use Mocking
- Advanced Topics and Test Doubles

Demo Company: PensionReady

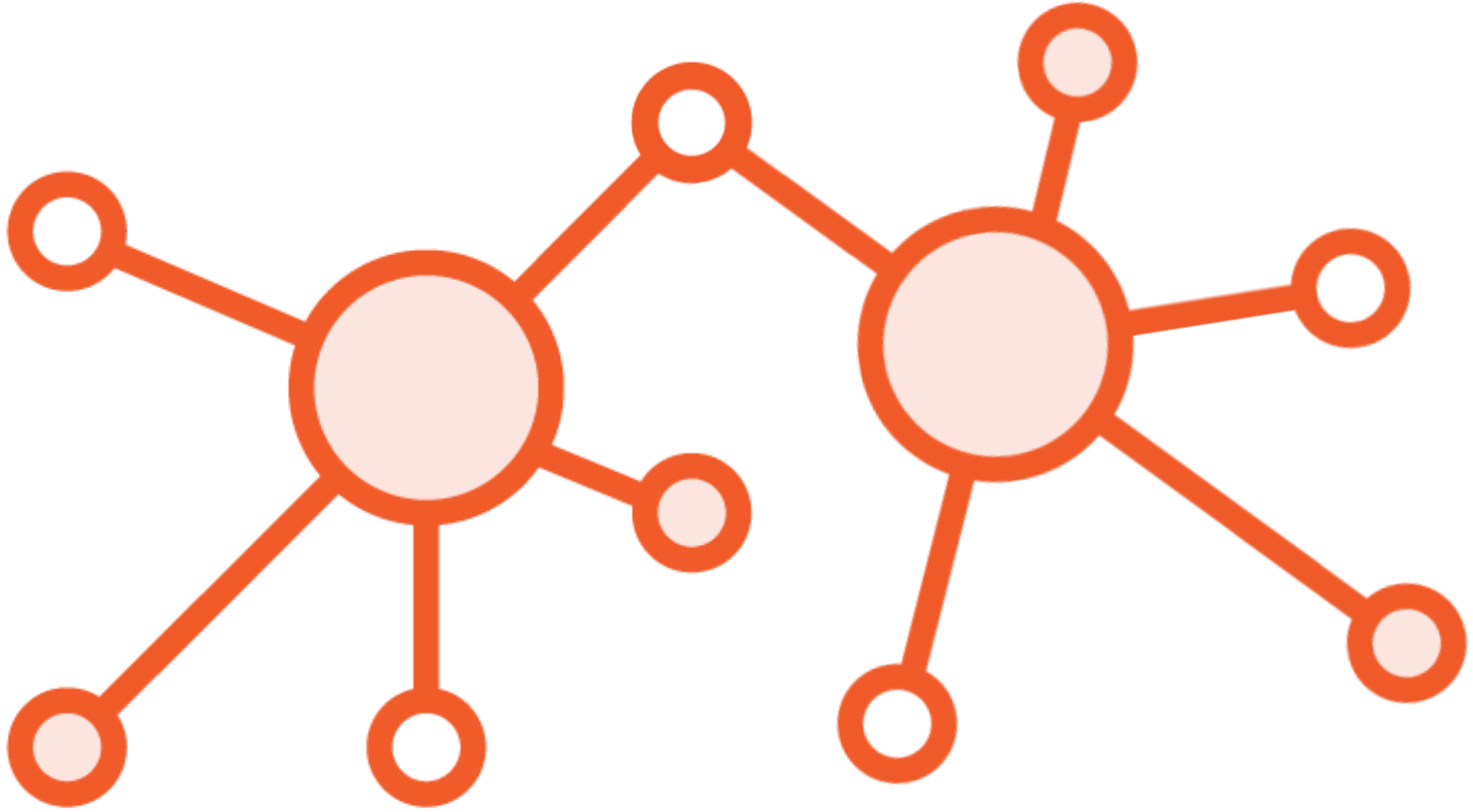


Demo

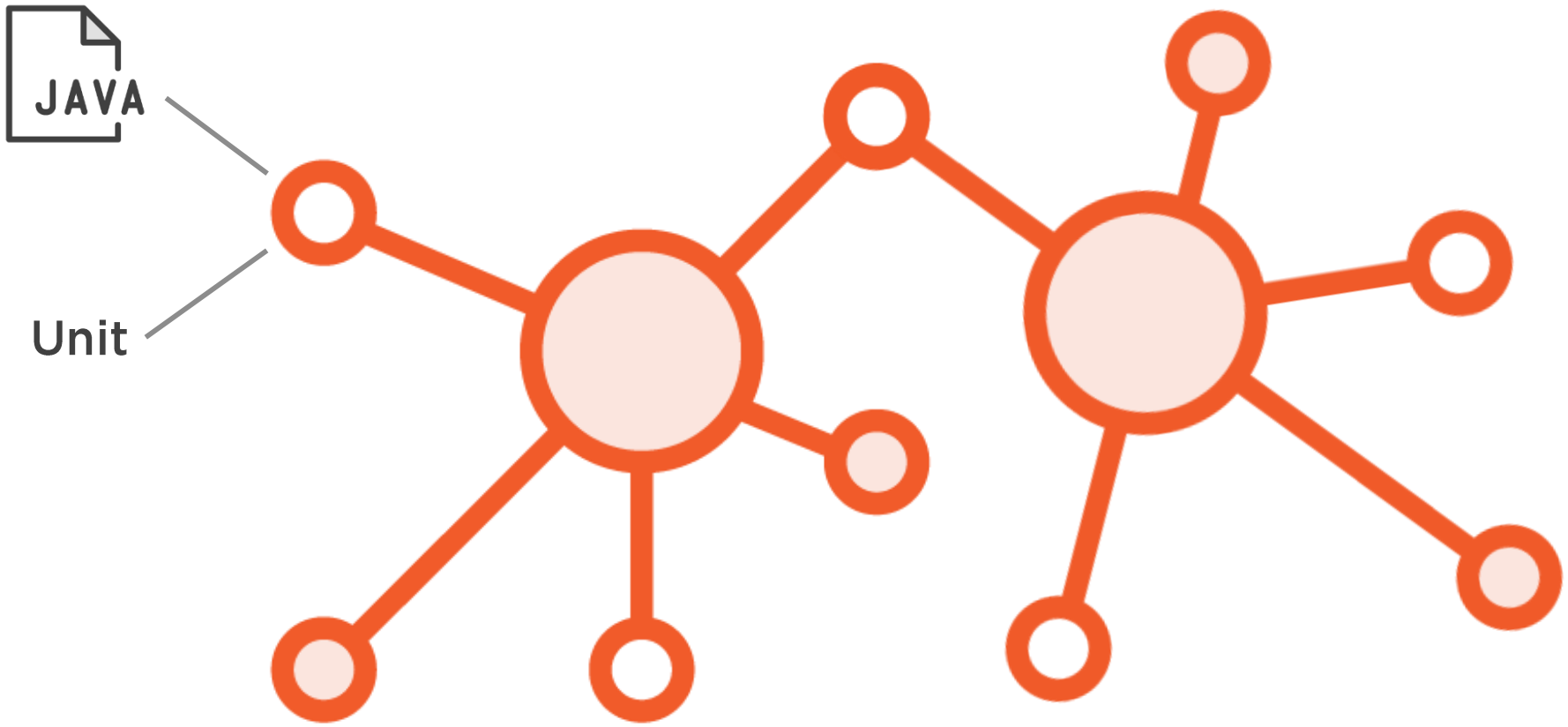
PensionReady: application overview

The first test (without mocks)

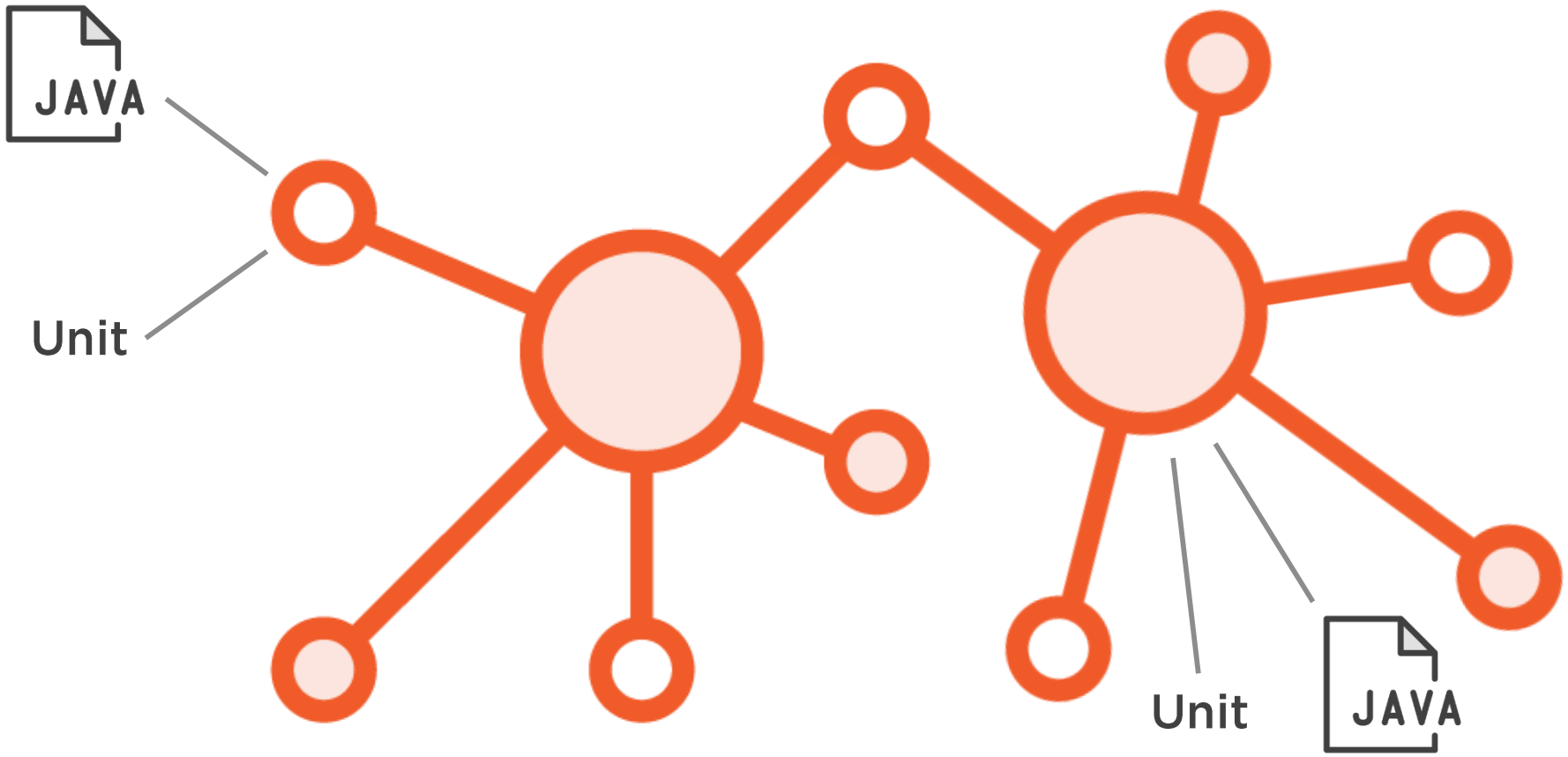
Unit Testing



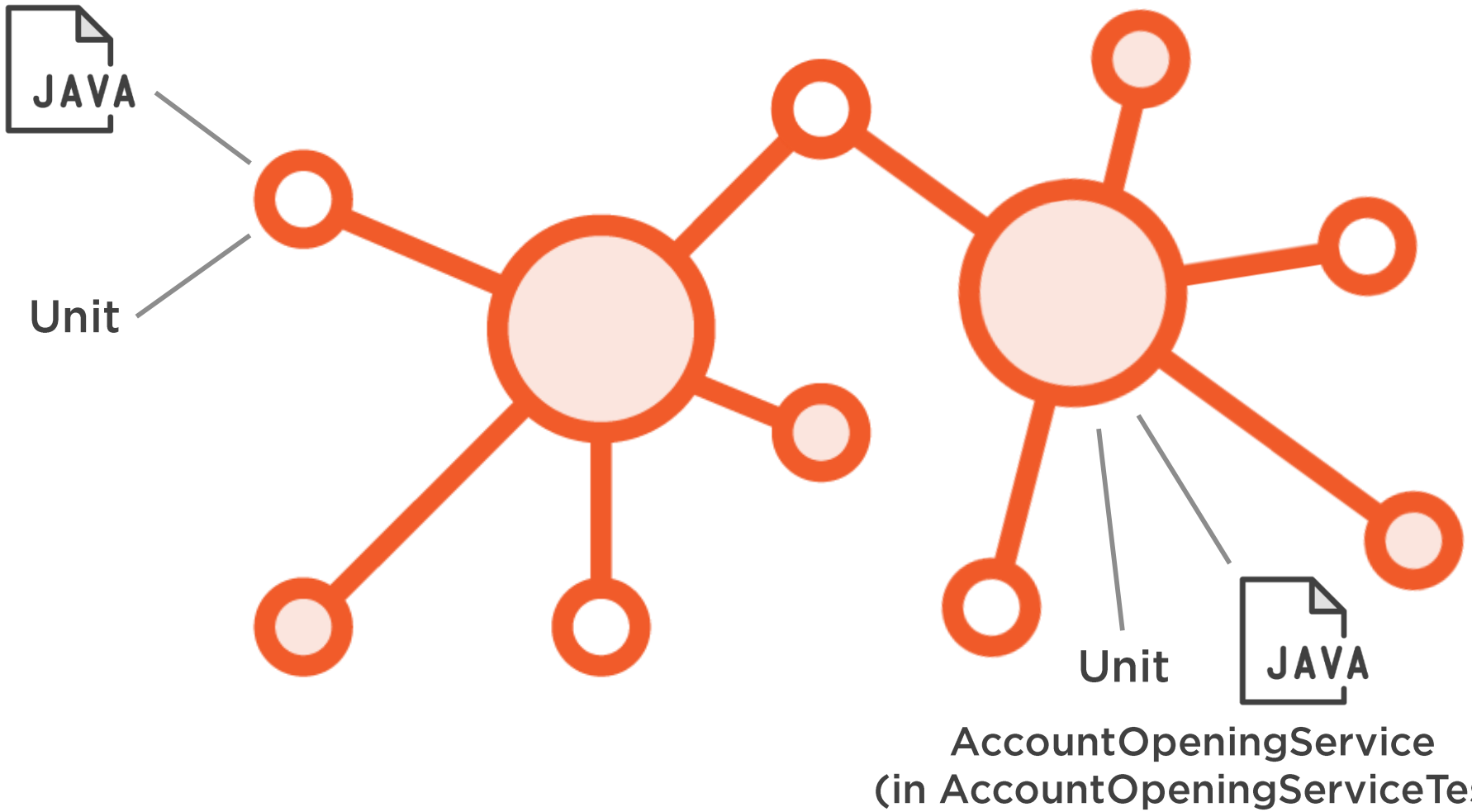
Unit Testing



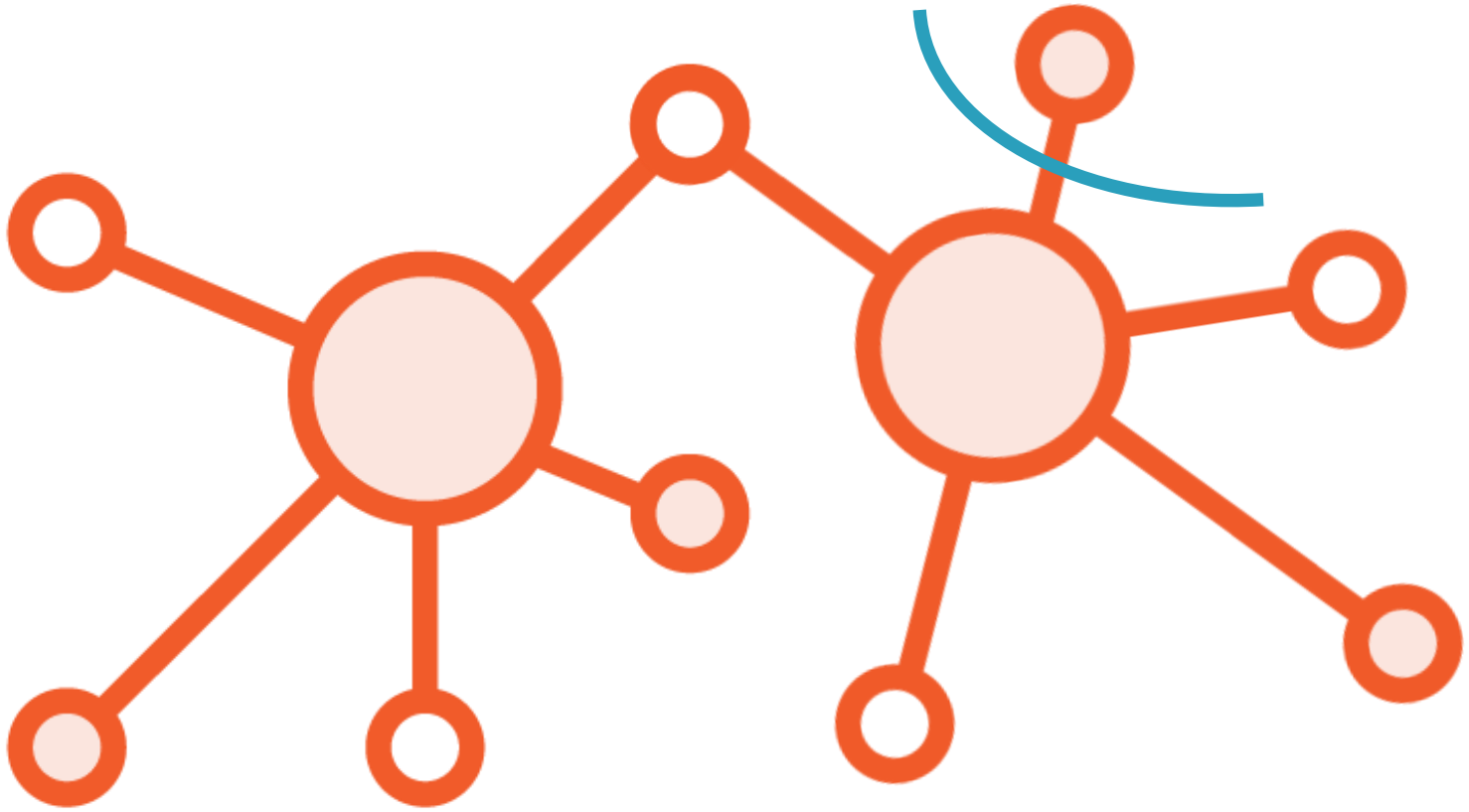
Unit Testing



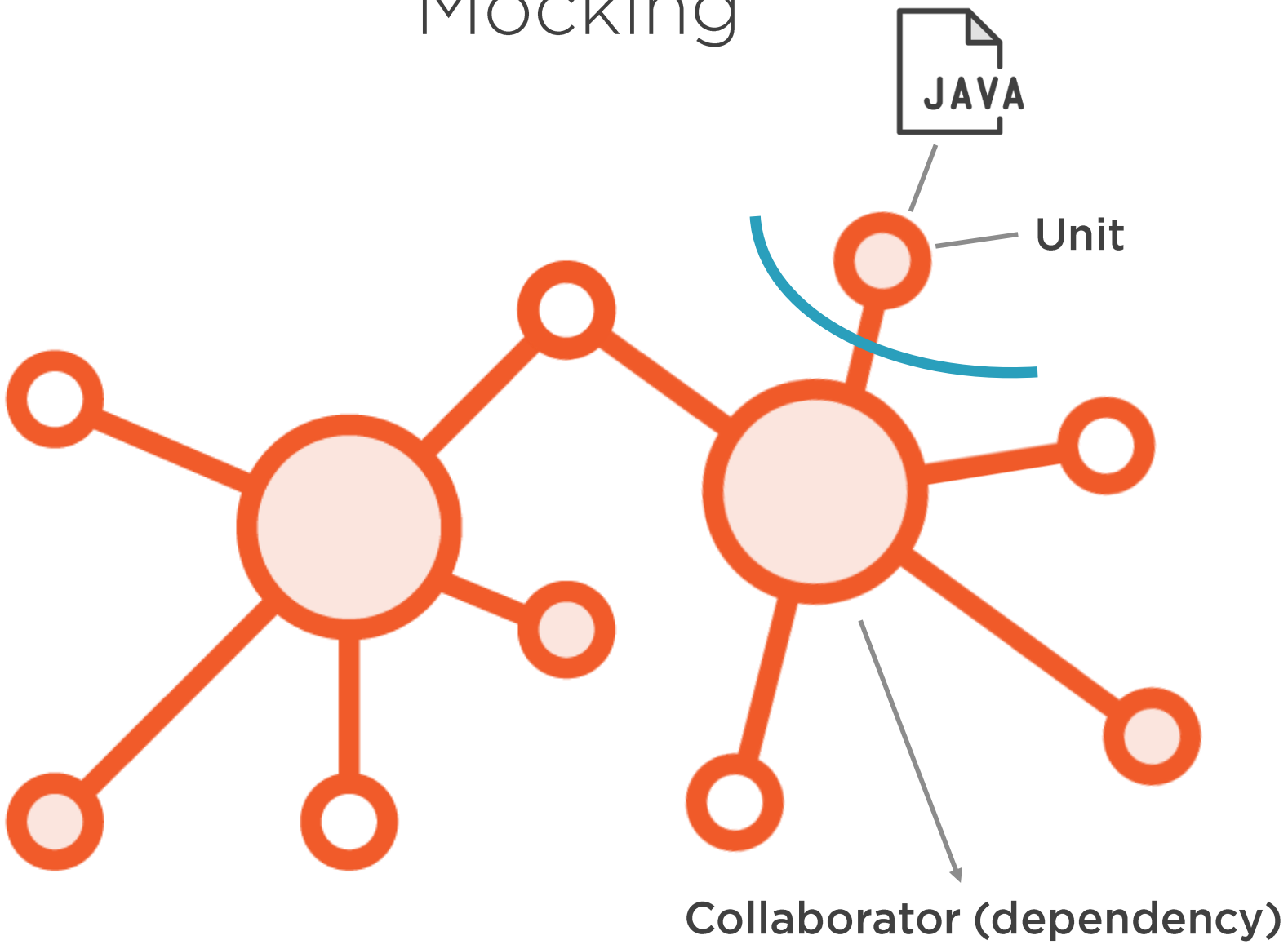
Unit Testing



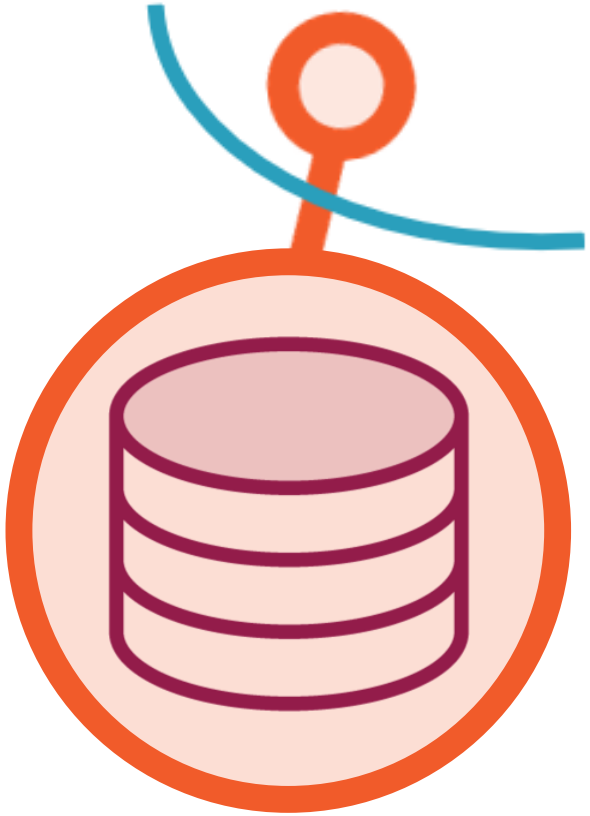
Mocking



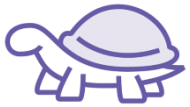
Mocking



Most Popular Collaborators for Mocking



Why Not to Use Databases in Your Unit Tests



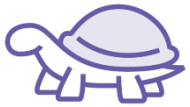
Databases are slow

What if your database is down when you try to run your tests?

What happens if your tests accidentally pick-up a production database config?

Most Popular Collaborators for Mocking

Why Not to Use Web Services in Your Unit Tests

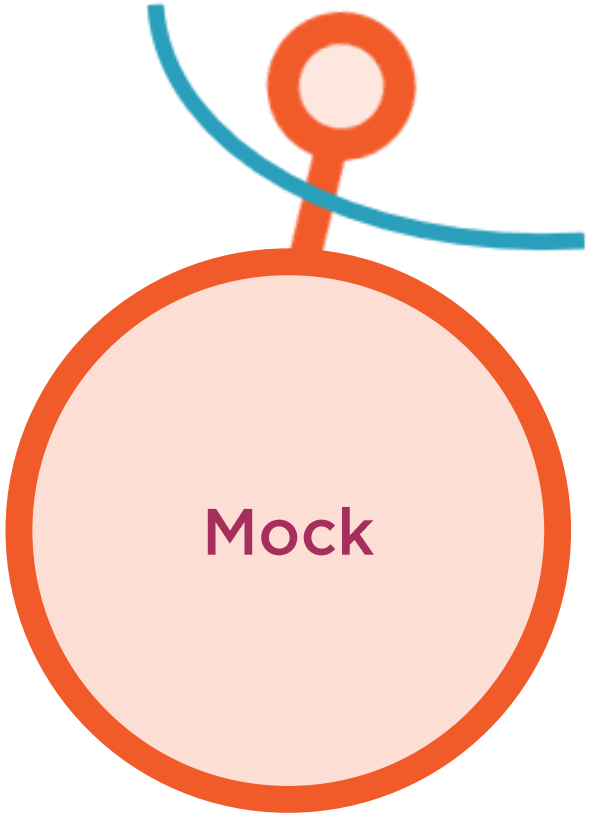


Using the network is slow

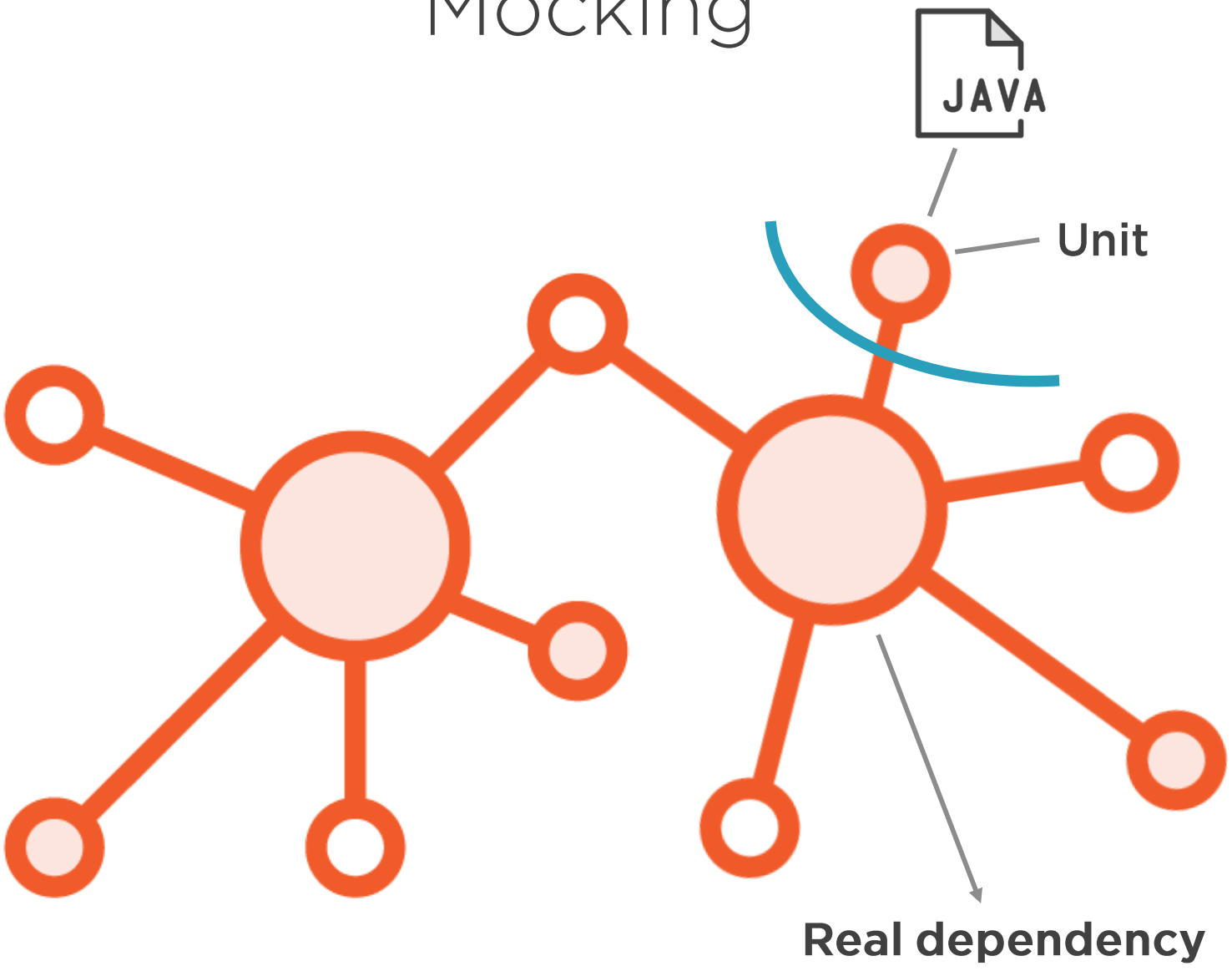
Writing a standalone web server process to interact with your tests, and imitate the real external service provider, takes time and effort to maintain

What happens if your tests accidentally pick-up a production server config and hit a real external service provider endpoint?

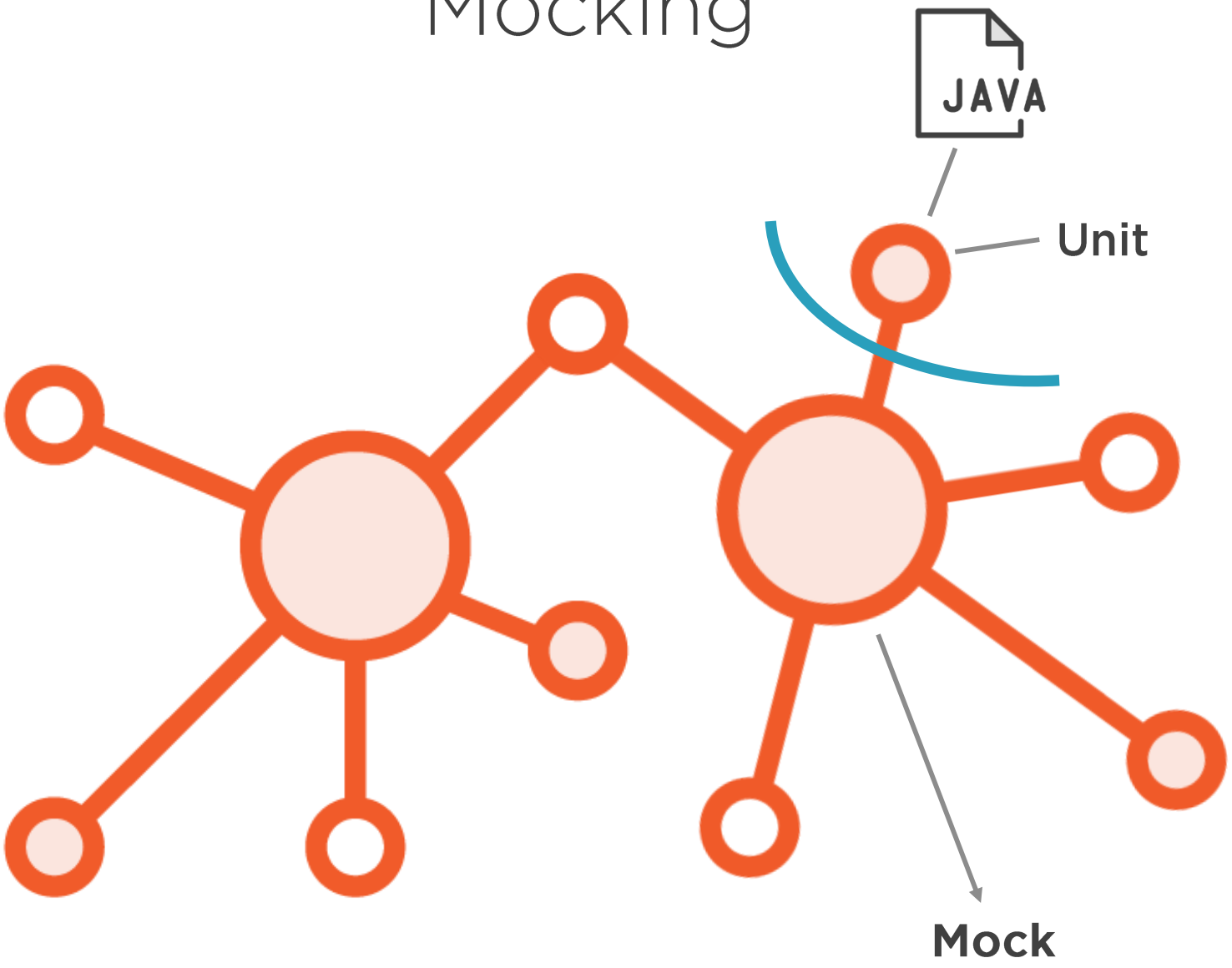
Most Popular Collaborators for Mocking



Mocking



Mocking



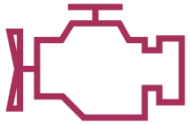
Mocking

Mocking allows you to focus on the unit you're trying to test by replacing the unit's real dependencies with test-only collaborators. This allows you to reason about the unit in isolation, without having to deal with the rest of the codebase at the same time.

Mocking Analogy: Electric Car

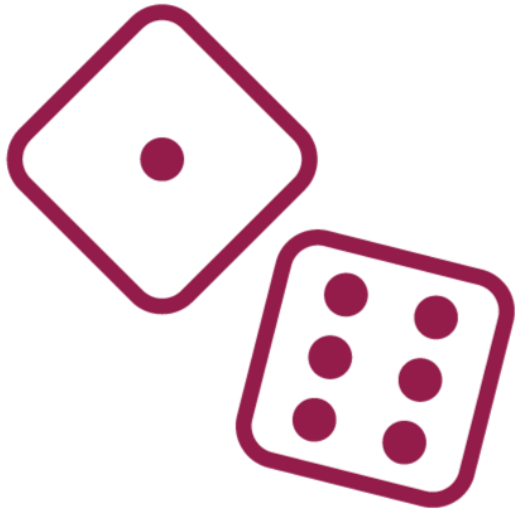
Imagine you're building an electric car

You want to test how the working interior of the car behaves when used



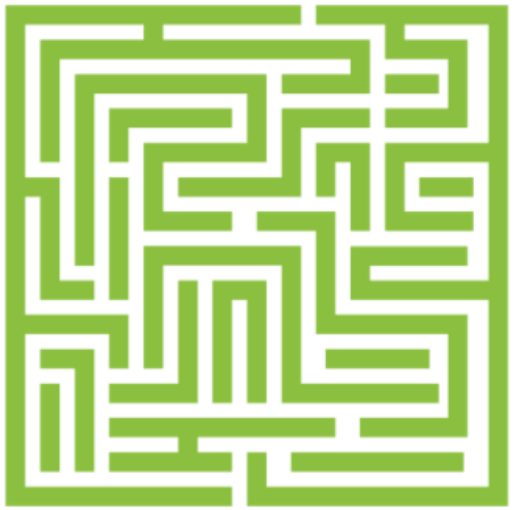
But you don't want to install the production-quality power system for this: you just install a 'mock' one, enough to verify that the interior components correctly interact with the power system

Why Use Mocking?



Eliminates non-determinism

- And randomness



Reduces complexity

- Increases flexibility

Improves test execution:

- Speed
- Reliability

Supports collaboration

Why Use Mocking: When Things Go Wrong

Knight Capital, a leading financial company, lost >400 million USD due to wrong execution of client orders

This happened due to: not removing dead code, introducing confusion by feature flag repurposing, and incorrect code refactoring

Mocking can be used to prevent such disasters. When code refactoring happens, mock-enabled tests can catch incorrect interactions between components

Demo

Installing EasyMock

Adding mocks to our first PensionReady unit test

“Double is a generic term for any case where you replace a production object for testing purposes.”

Martin Fowler

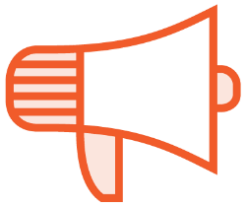
Test Doubles

Dummy

Fake

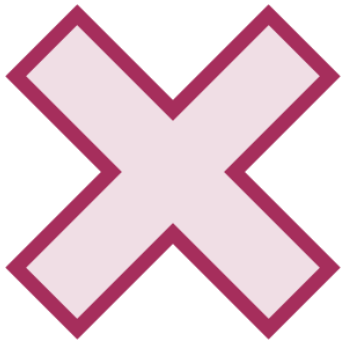
Stub

Spy



Mock

Test Doubles: Dummy

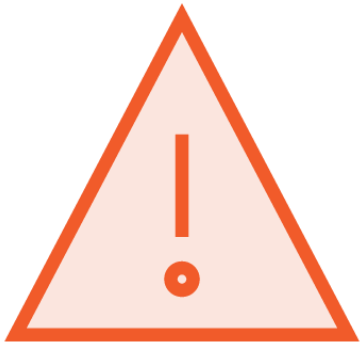


Not used in the test

Fills-in holes

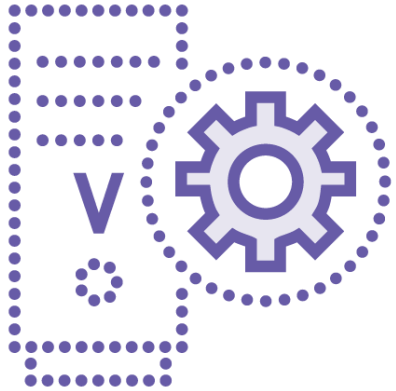
Just keeps the compiler happy

Test Doubles: Fake



Implementation that is unsuitable for production

In-memory DB



Fake web service

Test Doubles: Stub

Provides ‘canned’ answers

Not intelligent enough to respond with anything else

Multiple variations are possible

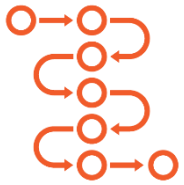
Test Doubles: Spy

Like a more intelligent Stub

Keeps track of how it was used

Also helps with verification

Test Doubles: Mock



Uses expectations

Can fail the test if unexpected calls are made

The focus is on behavior verification

State Verification Versus Behavior Verification

State Verification

All other Test Doubles do it

Verify resulting state

Exception: Spies can verify behavior

Behavior Verification

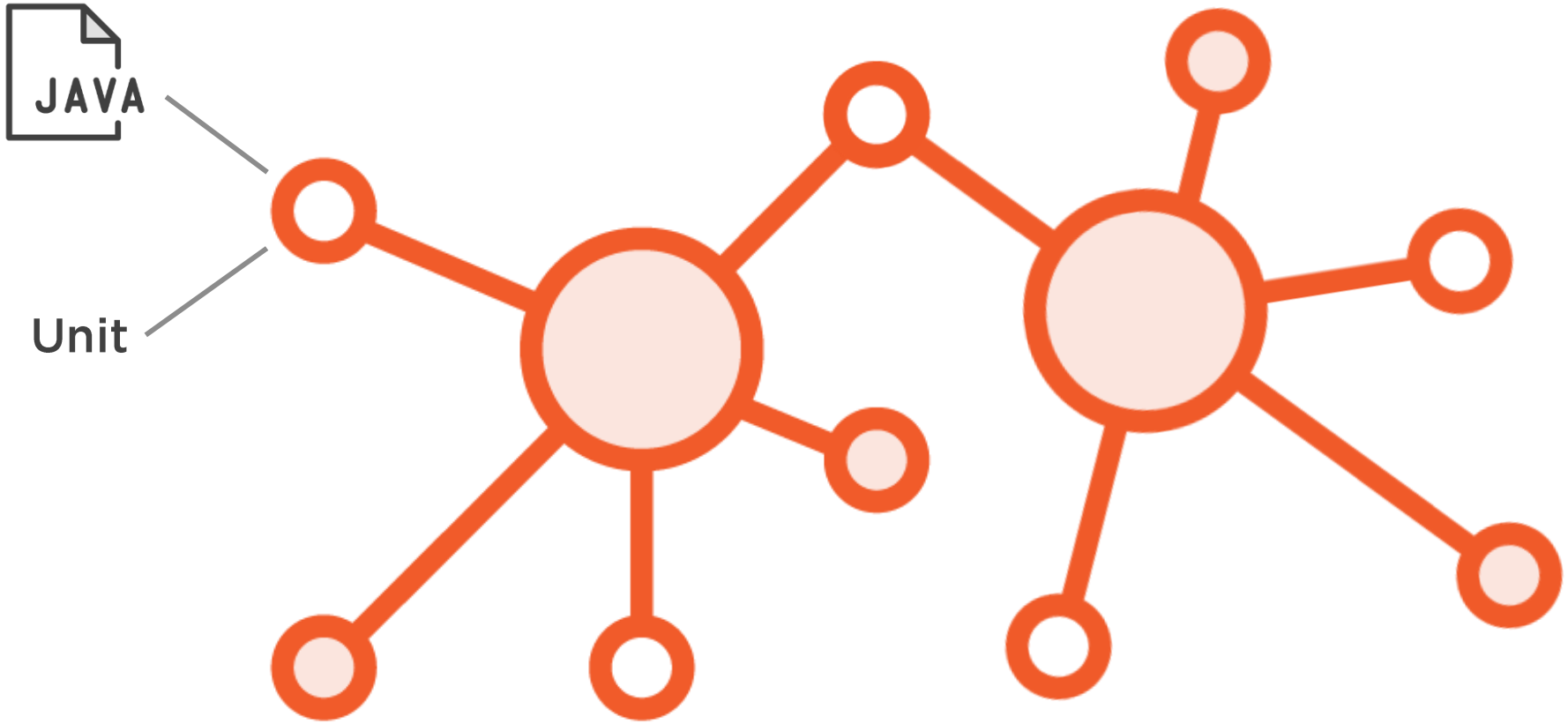
Mocks always use it

Verify interactions

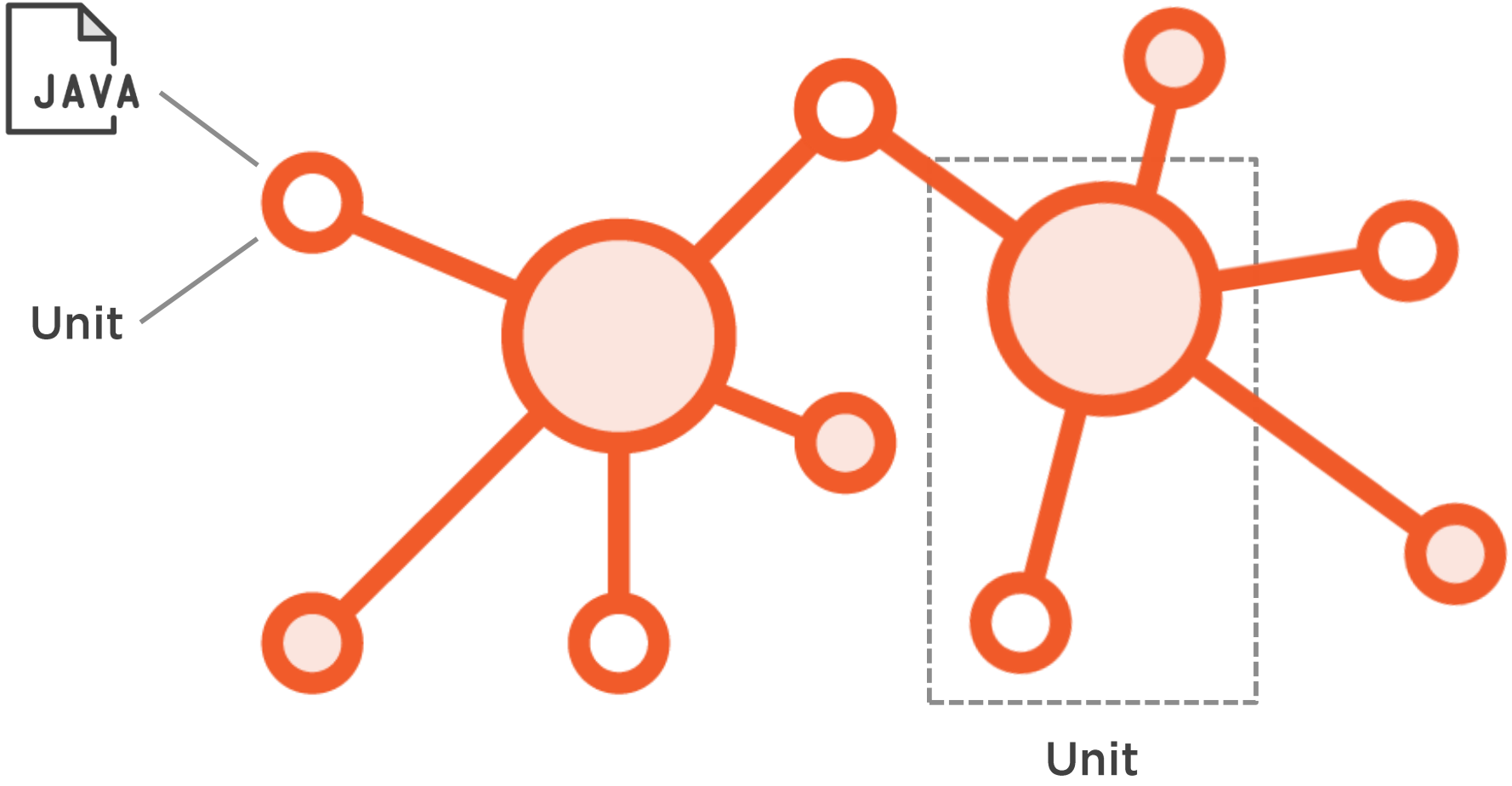
Can add state verification on top

For ease of learning, we'll focus on Spies and Mocks

A Few More Words on Unit Testing



A Few More Words on Unit Testing



“But really it’s a situational thing - the team decides what makes sense to be a unit for the purposes of their understanding of the system and its testing. ”

Martin Fowler

A Unit Can Consist of More Than One Class

1-class unit

```
private AccountOpeningService underTest;
private BackgroundCheckService backgroundCheckService =
                                        mock(BackgroundCheckService.class);
private ReferenceIdsManager referenceIdsManager =
                                        mock(ReferenceIdsManager.class);
private AccountRepository accountRepository =
                                        mock(AccountRepository.class);

@BeforeEach
void setUp() {
    underTest = new AccountOpeningService(
                        backgroundCheckService,
                        referenceIdsManager,
                        accountRepository);
}
```

A Unit Can Consist of More Than One Class

```
private AccountOpeningService underTest;
private BackgroundCheckService backgroundCheckService =
                                                mock(BackgroundCheckService.class);
private ReferenceIdsManager referenceIdsManager =
                                                mock(ReferenceIdsManager.class);
private AccountRepository accountRepository =
                                                mock(AccountRepository.class);

@BeforeEach
void setUp() {
    underTest = new AccountOpeningService(
                        backgroundCheckService,
                        referenceIdsManager,
                        accountRepository);
}
```

A Unit Can Consist of More Than One Class

```
private AccountOpeningService underTest;
private BackgroundCheckService backgroundCheckService =
                                                mock(BackgroundCheckService.class);
private ReferenceIdsManager referenceIdsManager =
                                                new ExternalNationalReferenceIdsManager(…);
private AccountRepository accountRepository =
                                                mock(AccountRepository.class);

@BeforeEach
void setUp() {
    underTest = new AccountOpeningService(
                        backgroundCheckService,
                        referenceIdsManager,
                        accountRepository);
}
```

A Unit Can Consist of More Than One Class

2-class unit

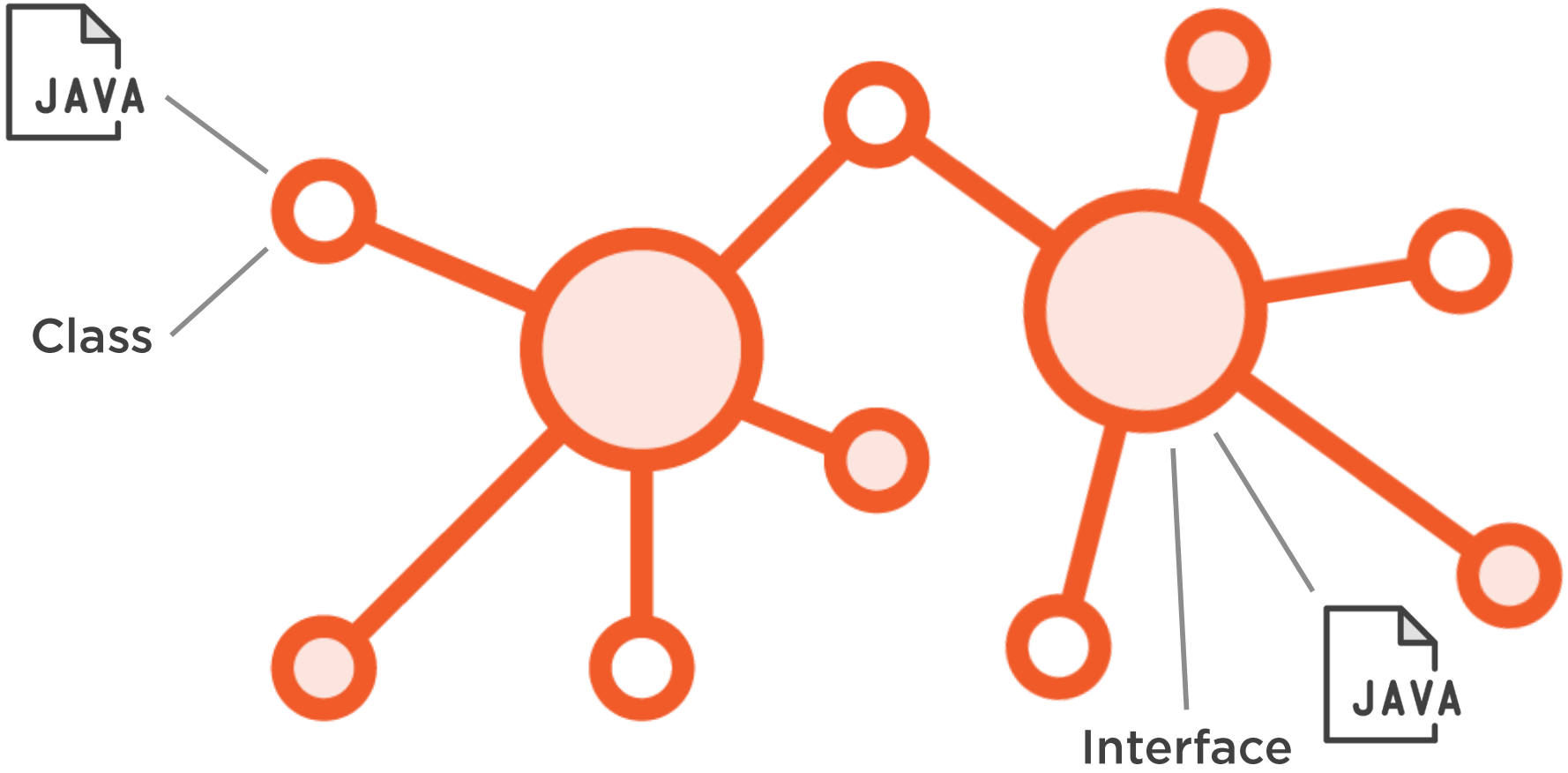
```
private AccountOpeningService underTest;
private BackgroundCheckService backgroundCheckService =
                                        mock(BackgroundCheckService.class);
private ReferenceIdsManager referenceIdsManager =
                                        new ExternalNationalReferenceIdsManager(…);
private AccountRepository accountRepository =
                                        mock(AccountRepository.class);

@BeforeEach
void setUp() {
    underTest = new AccountOpeningService(
                        backgroundCheckService,
                        referenceIdsManager,
                        accountRepository);
}
```

Unit Formation: Paradigm Comparison

Object-Oriented	Functional
A class	A function
A collection of classes	A collection of functions

Do You Mock Concrete Classes or Interfaces?



Do You Mock Concrete Classes or Interfaces?

EasyMock

Can mock a class

Can mock an interface

Other frameworks

Other frameworks may only allow mocking at interface level:

- by design
- due to technical reasons

```
BackgroundCheckService
backgroundCheckService =
      mock(BackgroundCheckService.class);


ReferenceIdsManager
referenceIdsManager =

      mock(ReferenceIdsManager.class);


AccountRepository
accountRepository =

      mock(AccountRepository.class);
```

- Generally speaking, prefer mocking interfaces over concrete classes in order to follow SOLID principles.
- BackgroundCheckService interface
- ReferenceIdsManager interface
- AccountRepository interface

Do You Mock Concrete Classes or Interfaces?

```
private AccountOpeningService underTest;
private BackgroundCheckService backgroundCheckService =
                                                mock(BackgroundCheckService.class);
private ReferenceIdsManager referenceIdsManager =
                                                mock(ReferenceIdsManager.class);
private AccountRepository accountRepository =
                                                mock(AccountRepository.class);

@BeforeEach
void setUp() {
    underTest = new AccountOpeningService(
                        backgroundCheckService,
                        referenceIdsManager,
                        accountRepository);
}
```

The interactions between the AccountOpeningService and the AccountRepository aren't affected by the inner workings of the class implementing the AccountRepository.

```
accountRepository.save(
id,
firstName,
lastName,
taxId,
dob,
backgroundCheckResults);
```

- In production, the AccountRepository could be implemented by a DatabaseAccountRepository, or a MemCacheAccountRepository, and the contract between AccountOpeningService and AccountRepository, shown in this slide, will be the same.

- And this precisely what we're testing: that they interact correctly.

- This is why we use the AccountRepository interface, rather than a concrete implementation, when creating the mock collaborator for AccountOpeningService.

EasyMock Facts

Current version: 4

- Supports Java 11, requires >= Java 8

Open source

The first-ever dynamic mock generator

Relies on the Java Proxy mechanism

https://github.com/easymock/easymock

Module Summary

PensionReady Application Overview

What Is Mocking

Writing Tests With Mocks

Wrote Tests Without Mocks

Why Use Mocking

Advanced Topics and Test Doubles