Software Testing, Quality Assurance & Maintenance—Lecture 2

Patrick Lam
University of Waterloo

January 9, 2026

Part I

Why Tests?

Why Tests?

Again: you have to move fast &
push a change to `main` by end of day.

Are you going break things? How do you know?

State of industry:
run the test suite!

Reference

Kat Busch. “A beginner’s guide to automated testing.”
https://hackernoon.com/
treat-yourself-e55a7c522f71

Experience report

To pass code review: needed tests.

“Lo and behold, I soon needed to fix a small bug.”

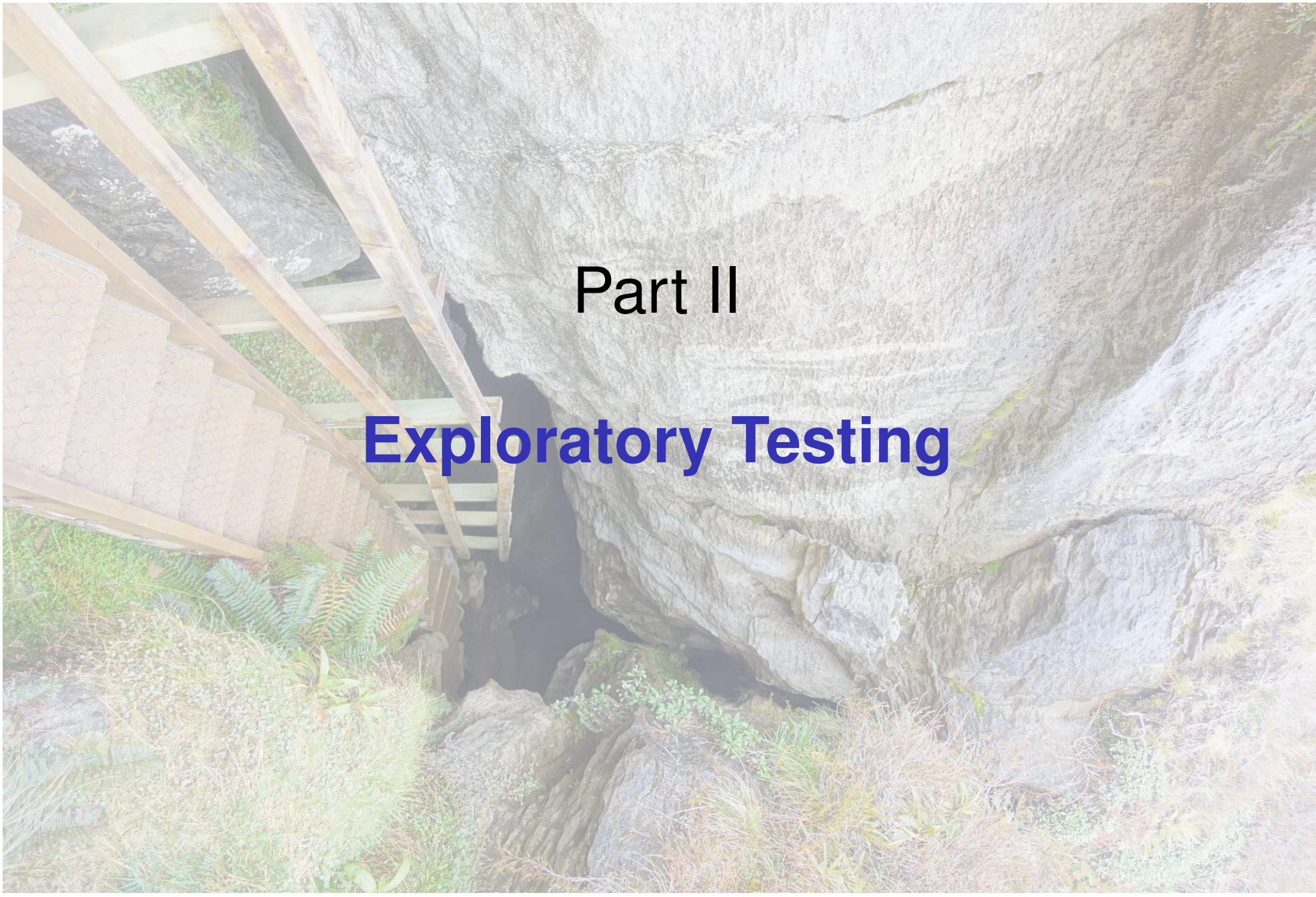
I ran the tests. Within a few seconds, I knew that everything still worked! Not just a single code path (as in a manual test), but all code paths for which I’d written tests! It was magical. It was so much faster than my manual testing. And I knew I didn’t forget to test any edge cases, since they were all still covered in the automated tests.

More quotes

If your code is still in the codebase a year (or five) after you've committed it and there are no tests for it, bugs will creep in and nobody will notice for a long time.

If it matters that the code works you should write a test for it. *There is no other way you can guarantee it will work.*

(We'll look at other ways in this course, but tests are the state of the industry.)



Part II

Exploratory Testing

Exploratory Testing

Different from other testing and verification activities in this course.

- usually done by dedicated testers, not developers;
- but, you may do “hallway usability testing”.

References in long-form notes.

Exploratory Testing Scenarios

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process

- Start with a charter for your testing activity,
  “Explore and analyze the product elements.”
  These charters should be somewhat ambiguous.
- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Output from Exploratory Testing

- a set of bug reports;
- test notes: overall impressions & summary of test strategy / thought process;
- artifacts like test data / test materials (also serve as exploratory testing inputs)

In-class exercise: Exploratory testing of WaterlooWorks [5min]

Charter: “Explore the overall functionality of WaterlooWorks”.

- Summarize what the purpose of WaterlooWorks is.
- Identify the tasks that WaterlooWorks should be able to do; primary or contributing?
- Identify areas of potential instability.
- Test each function and record results (bugs).

PS: don’t do things with actual real-world effects;
normally you’d test on dev not prod.

Part III

Regression Testing

Why Regression Tests?

People hate regressions.
So, aim to detect regressions:

- of fixed bugs;
- in related and unrelated other features.

Usually this is integration-level testing.

Properties of Regression Tests

- automated!
  (usually low-yield)
- appropriately sized
  (should be part of continuous integration)
- up-to-date

Automating Regression Tests

Pretty easy when it's e.g. a compiler.
Otherwise, not so easy.

Input:

- from a file?
- webform submissions?
- interacting with a webpage?

Approaches for non-file inputs:

- special mocks to take file-based inputs;
- capture and replay events (e.g. Selenium).

Automating Regression Tests: Output

How to verify output? Can be hard!
resolution, whitespace, window placement...

Case study: Gecko (used in Firefox, Thunderbird).

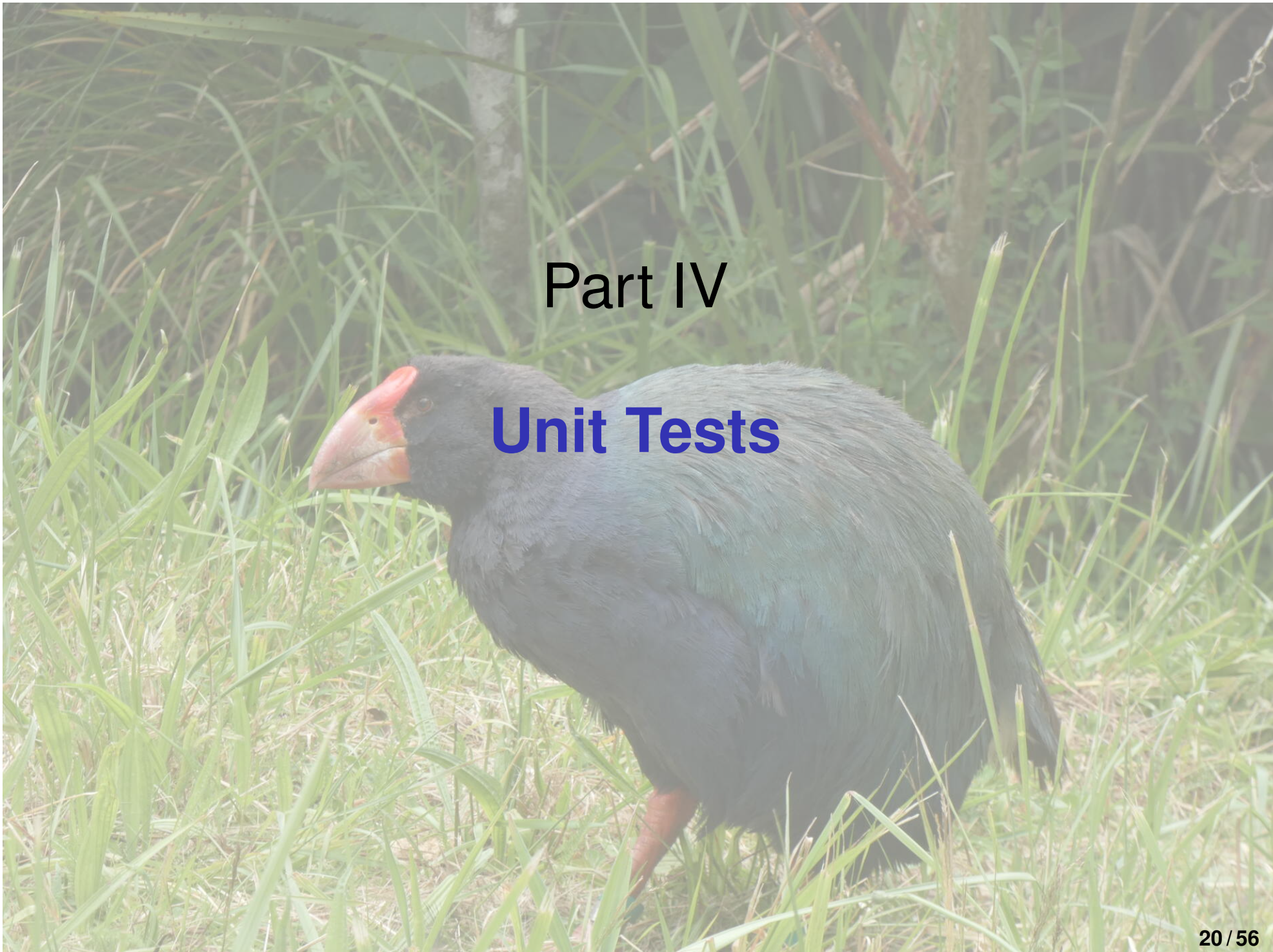
1. manual testers
2. capture screenshots, compare (often failed)
3. enable logging, compare logs.

Some Industrial Best Practices

What have you seen?

Some Industrial Best Practices

- unit tests
- code reviews
- continuous builds
- one-button deploy
- undo/back button



Part IV

Unit Tests

About Unit Tests

- focus on one particular class, module or function;
- should execute quickly;
- may need fake inputs or mocks;
- should generally not use an entire real input for a unit test.

An Example Unit Test

This example is JUnit; many other frameworks exist.

```
@Test
public void testFindLast() {
    int[] x = new int[] {2, 3, 5}; // arrange
    int last = FindLast.findLast(x, 2); // act
    assertEquals(0, last); // assert
}
```

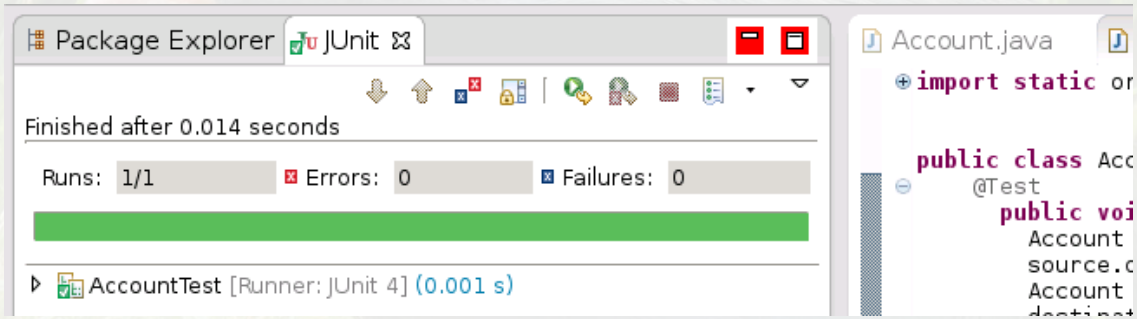
More Pro-Test Propaganda

You'll find that writing tests as you go makes your interfaces better and makes your code more testable. If you find yourself writing something hard to test, you'll notice it early on when there's still time to improve the design.

Goal

Good tests are *self-checking*:

no errors, no failures = successful test.



Why Self-Checking Tests?

Tests automatically report status.

Enables “keep the bar green” coding style.

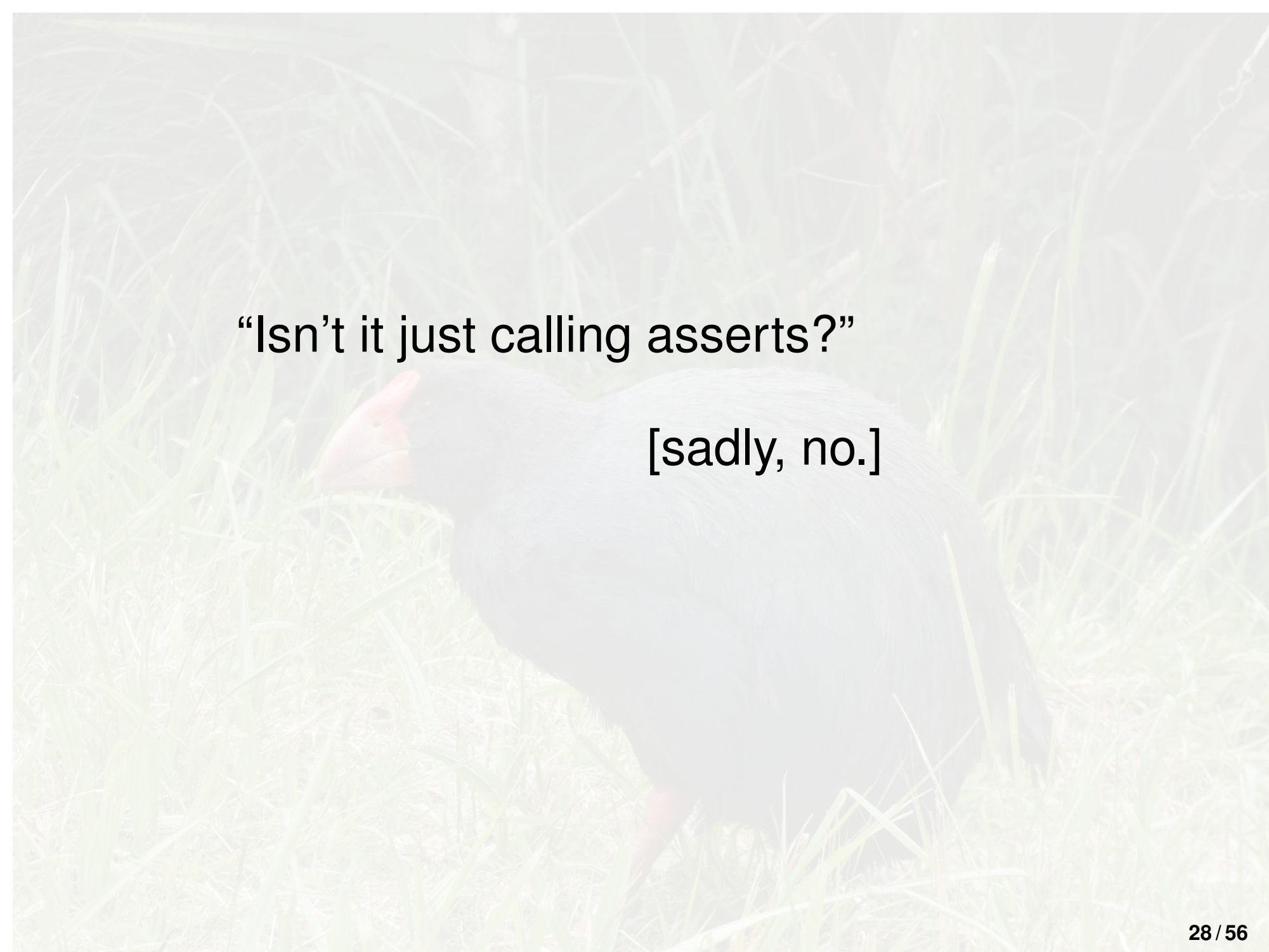
Worry less about introducing bugs.

Plus: Tests help document system specs.

Plan

HOWTO make your tests self-checking.


“Isn’t it just calling asserts?”


“Isn’t it just calling asserts?”
[sadly, no.]

Two questions about asserts:

1. Q: what for?
   A: check method call results

2. Q: where?
   A: usually after calling SUT
   (System Under Test)

Counter Example

```
public class Counter {
    int count;

    public int getCount() { return count; }
    public void addToCount(int n) { count += n; }
}
```

Counter Test

```
// java -cp /usr/share/java/junit4.jar:. \
//    org.junit.runner.JUnitCore CounterTest
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
  @org.junit.Test
   public void add10() {
        Counter c = new Counter(); // arrange
        c.addToCount(10); // act
        // after calling SUT, read off results
        int count = c.getCount();
        assertEquals("value", count); // assert
   }
}
```

State or Behaviour?

Was Counter Test verifying state or behaviour?

State vs Behaviour

State: e.g. object field values.
Call accessor methods to verify.

Behaviour: which calls SUT makes.
Insert observation points,
monitor interactions.

Flight example

```
// Meszaros, p. 471
// not self-checking
public void testRemoveFlightLogging_NSC() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 // have not found a way to verify the outcome yet
 //  Log contains record of Flight removal
}
```

Flight example: state verification

```
// Meszaros, p. 471
// extended state specification
public void testRemoveFlightLogging_NSC() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 assertFalse("flight still exists after removed",
             facade.flightExists(expectedFlightDto,
                                 getFlightNo()));
}
```

What Is State Verification?

1. Exercise SUT.
2. Verify state & check return values.

Inspect only outputs;
only call methods from SUT.

Do not instrument SUT.

Do not check interactions.

Implementing State Verification

Two options:

1. procedural (bunch of asserts); or,
2. via expected objects (won't talk about them this year).

Flight Example: discussing state verification

We do check that the flight got removed.
We don’t check that the removal got logged.

Hard to check state and observe logging.

Solution: Spy on SUT behaviour.

Flight example: procedural behaviour verification

```
// Meszaros, p. 472
// procedural behaviour verification
public void testRemoveFlightLogging_PBV() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 AuditLogSpy logSpy = new AuditLogSpy();
 facade.setAuditLog(logSpy);
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 assertEquals("number of calls",
              1, logSpy.getNumberOfCalls());
 // ...
 assertEquals("detail",
              expectedFlightDto.getFlightNumber(),
              logSpy.getDetail());
}
```

Alternative: Expected Behaviour Specification

Use a mock object framework (e.g. JMock) to define expected behaviour.

Observe calls to the logger,
make sure right calls happen.

Kinds of Assertions

Three built-in choices:

1. assertTrue(aBooleanExpression)
2. assertEquals(expected, actual)
3. assertEquals(expected, actual, tolerance)

note: `assertTrue` can give
hard-to-diagnose error messages
(must try harder when using).

Using Assertions

Assertions are good:

- to check all things that should be true
  (more = better)
- to serve as documentation:
  when system in state S_1,
  and I do X,
  assert that the result should be R, and
  that system should be in S_2.
- to allow failure diagnosis
  (include assertion messages!)

Not Using Assertions

Can also do external result verification:

Write output to files, diff (or custom diff) expected and actual output.

Twist: expected result then not visible when looking at test.

(What's a good workaround?)

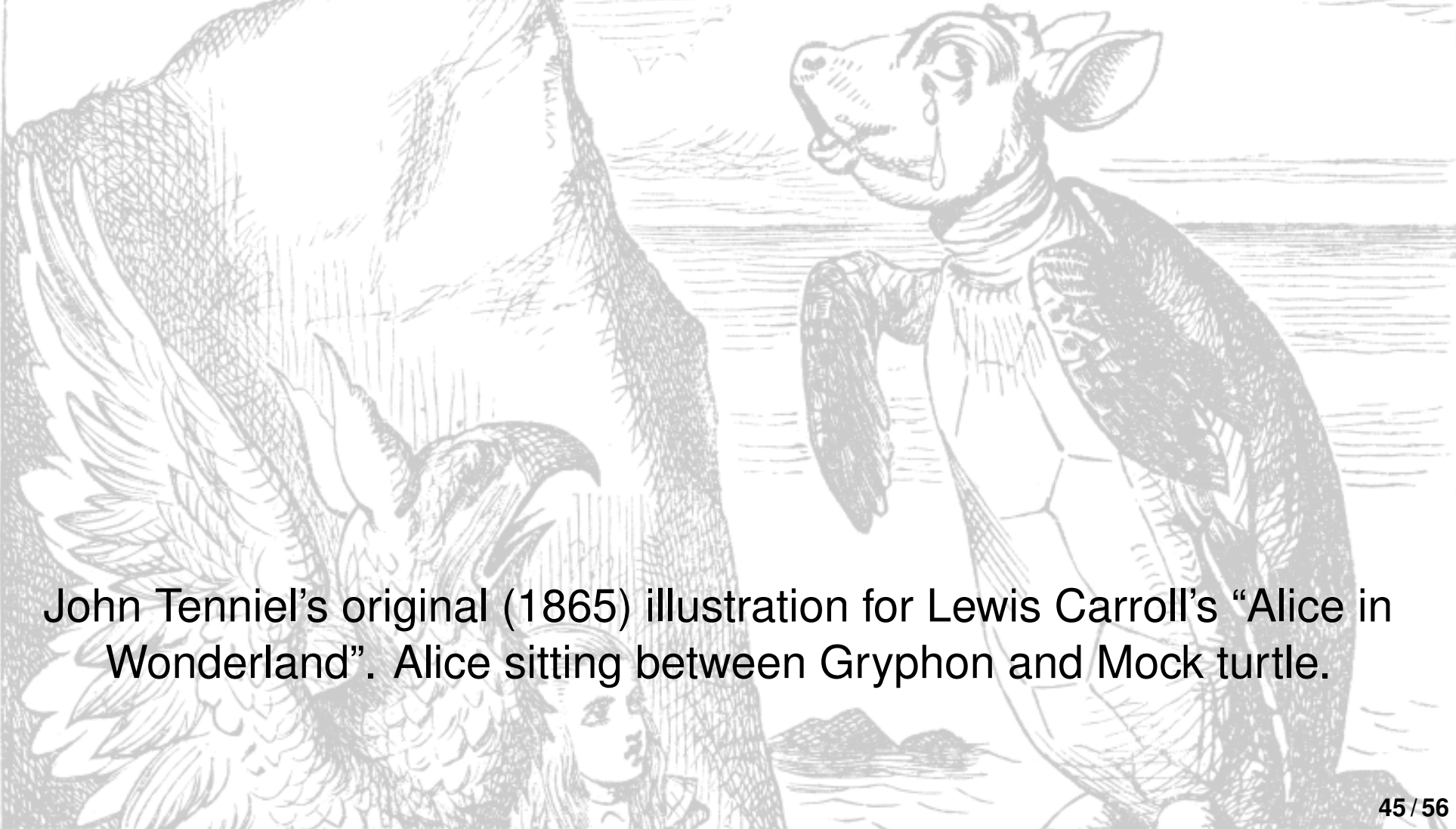
Verifying Behaviour

Observe actions (calls) of the SUT.

- procedural behaviour verification; or, (challenge: recording & verifying behaviour)
- via expected behaviour specification. (also captures outbound calls of SUT)

Part V

Mock Objects



John Tenniel's original (1865) illustration for Lewis Carroll's "Alice in Wonderland". Alice sitting between Gryphon and Mock turtle.

Things like mocks

- dummy objects: don't do anything; are like `null` or `None`,
  but pass nullness checks.
- fake objects: have actual (correct) behaviour,
  but unsuitable for production, e.g. in-memory db.
- stubs: produce canned answers.
- mocks: produce canned answers,
  but also check the interaction protocol.

Mock Objects: Email Sender

```
// state or behaviour verification?
public class MailServiceStub
                    implements MailService {
   private List<Message> messages =
       new ArrayList<Message>();

   public void send (Message msg) {
     messages.add(msg);
   }
   public int numberSent() {
     return messages.size();
   }
}
```

Mock Objects: Behaviour Verification

```
// jMock syntax
class OrderInteractionTester... {
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class); // (1)
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send"); // (2)
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

Mock Objects: Document Management

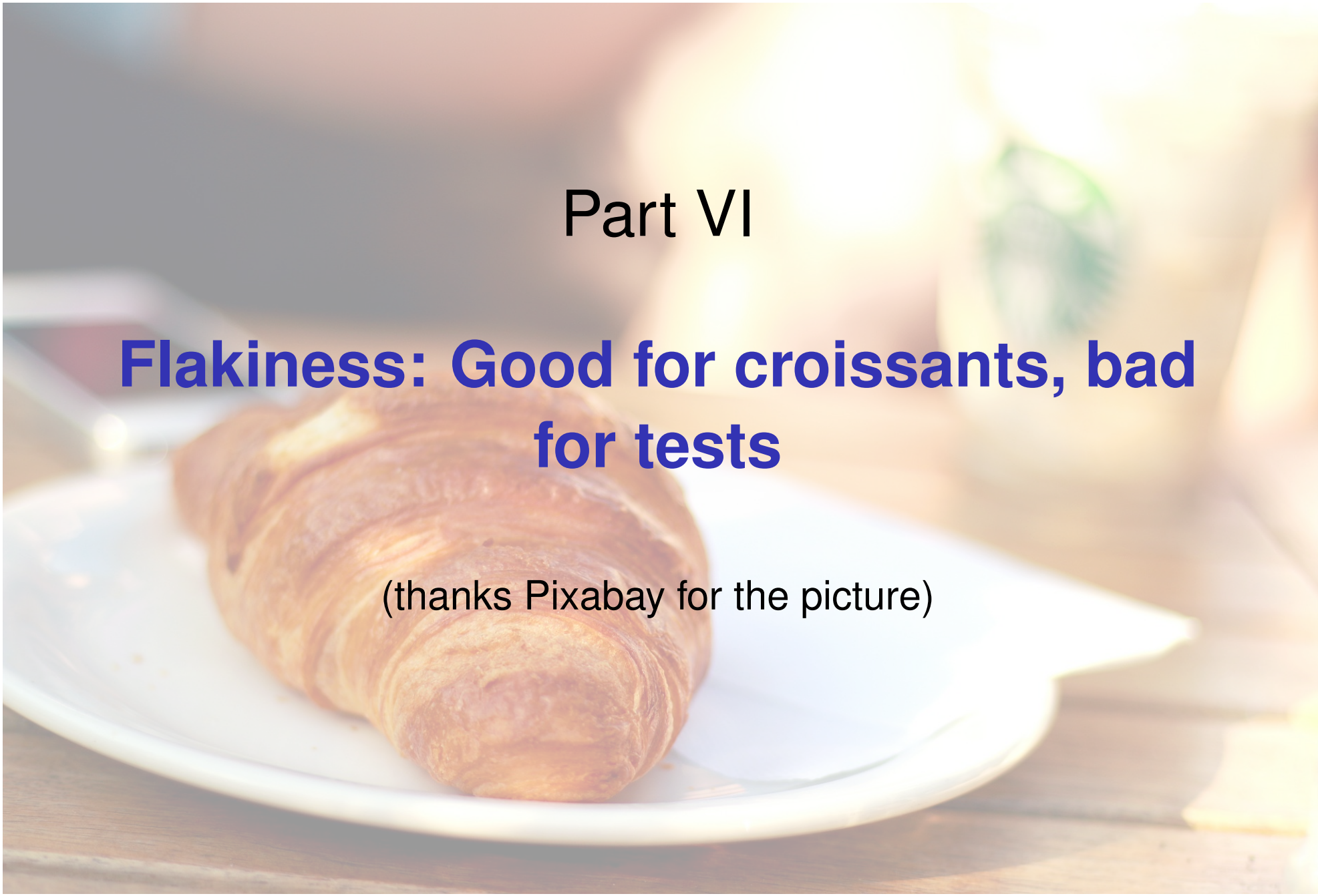
```
// EasyMock syntax
@RunWith(EasyMockRunner.class)
public class ExampleTest {
  @TestSubject
  private ClassUnderTest classUnderTest =
      new ClassUnderTest();

  @Mock // creates a mock object
  private Collaborator mock;

  @Test
  public void testRemoveNonExistingDocument() {
    replay(mock);
    classUnderTest.removeDocument
        ("Does not exist");
  }
}
```

Mock Objects: Using a Mock

```
@Test
public void testAddDocument() {
  // ** recording phase **
  // expect document addition
  mock.documentAdded("Document");
  // expect to be asked to vote for document removal,
  expect(mock.voteForRemoval("Document"))
                .andReturn((byte) 42);
  // expect document removal
  mock.documentRemoved("Document");
  replay(mock);
  // ** replaying phase **
  // we expect the recorded actions to happen
  classUnderTest.addDocument("New Document",
      new byte[0]);
  // check that the behaviour actually happened:
  verify(mock);
```



Part VI

Flakiness: Good for croissants, bad for tests

(thanks Pixabay for the picture)

Dealing with Flakiness

There are mitigations:

- label known-flaky tests and rerun them
- ignore or remove flaky tests

They're not great.

Takes a long time to re-run tests.

Causes of Flakiness

Luo et al studied 201 fixes to flaky tests in open-source projects. Fixable causes:

- improper waits for asynchronous responses;
- concurrency; and
- test order dependency.

Asynchronous waits

Typically: a `sleep()` call which didn't wait long enough.

Best practice:

- use some sort of `wait()` call to wait for the result, instead of hardcoding a sleep time.

Concurrency

The usual (see CS343 for more):

- data races;
- atomicity violations;
- deadlocks.

Test order dependency

- Test A expects test B to have already executed
  (and to have left a side effect, e.g. a file).

Especially common in the transition from Java 6 to Java 7, because the test execution order changed.

Solution: remove the dependency.