An introduction to software testing

Andrew Rice

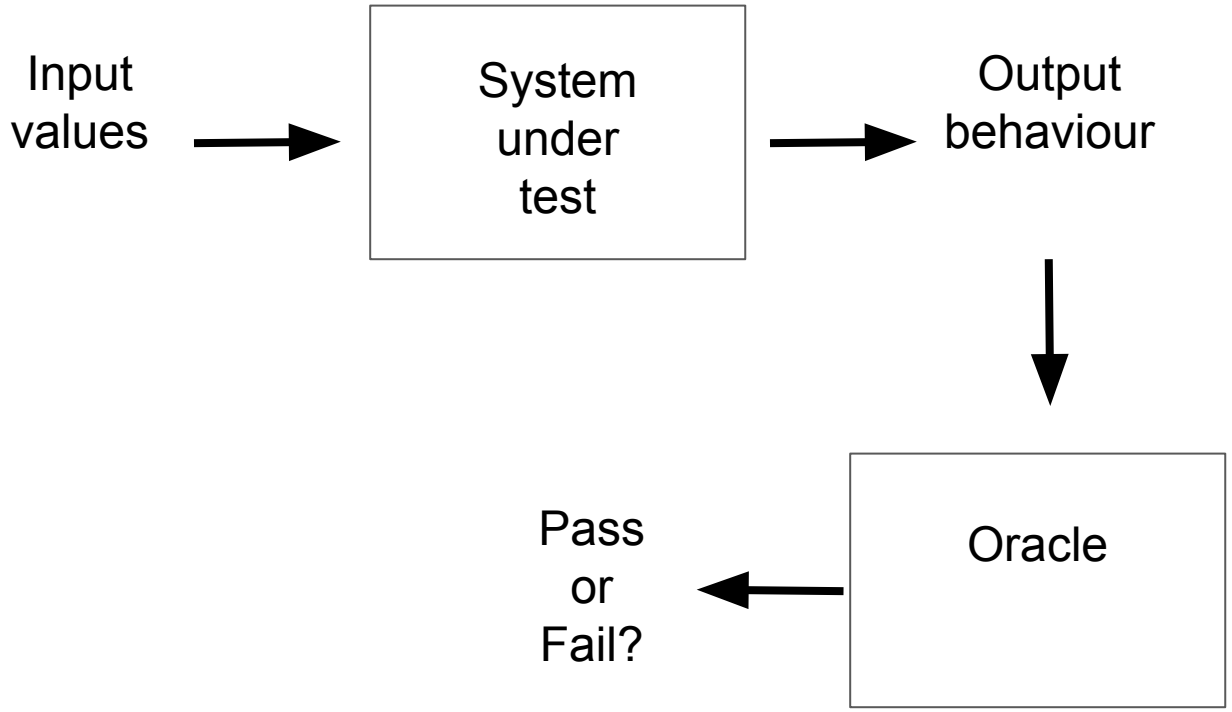
Some problems can be detected statically

```
1       fun nth 0 (x::_) = x
2         |  nth n (x::xs) = nth (n-1) x;
```

Many problems cannot

```
1       fun nth 0 (x::_) = x
2         |  nth n (x::xs) = nth (n-1) xs;
3
4       var l = nth 10 [1,2,3];
```

Testing checks how software performs at run-time



Objectives

1. Identify different types of test
2. Be able to write a 'good' unit test
3. Know about some techniques for measuring test quality
4. Understand how testing fits into the software development process

Different types of test

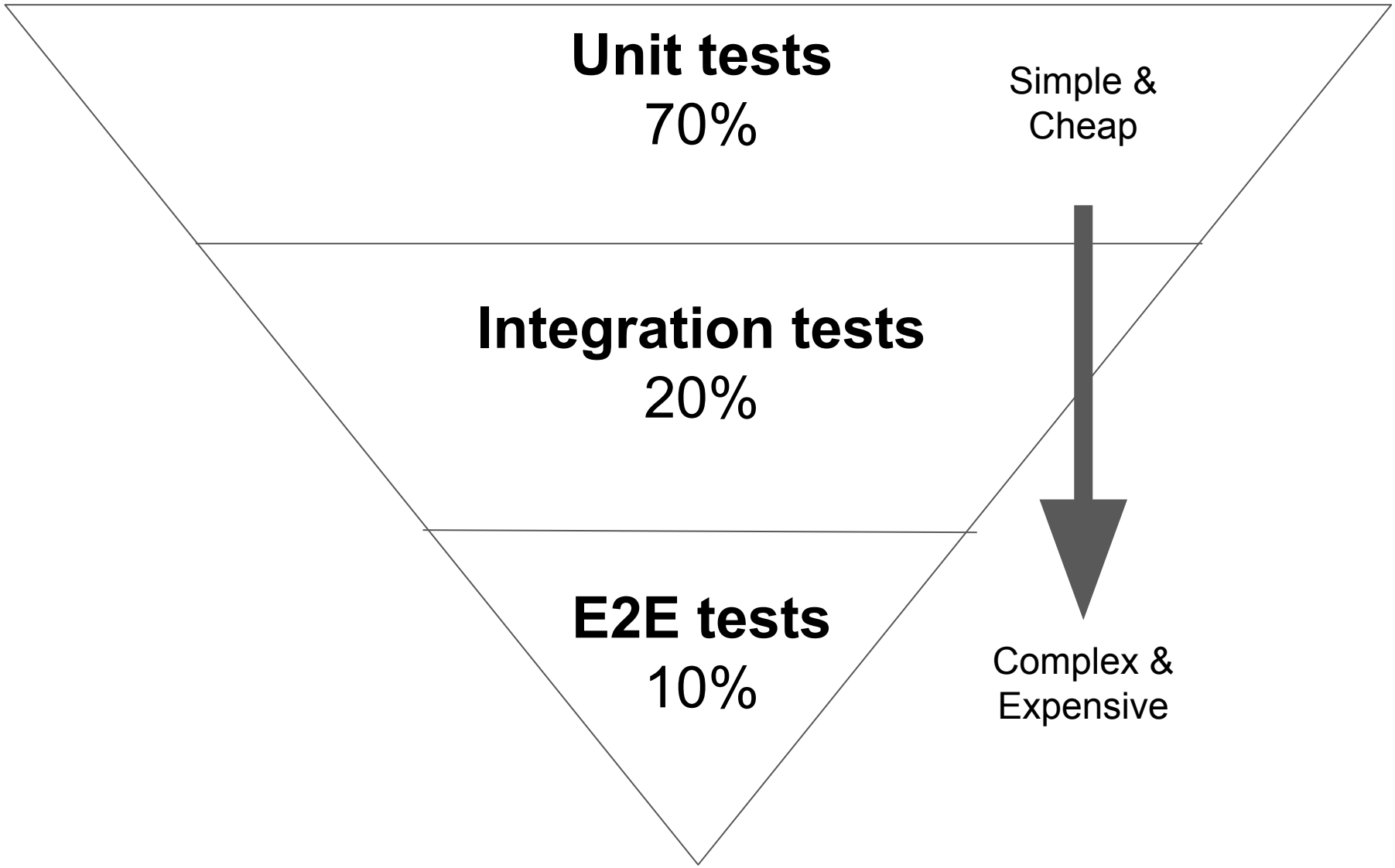
We will consider three kinds of testing

Unit tests
check isolated pieces of functionality

Integration tests
check that the parts of a system work together

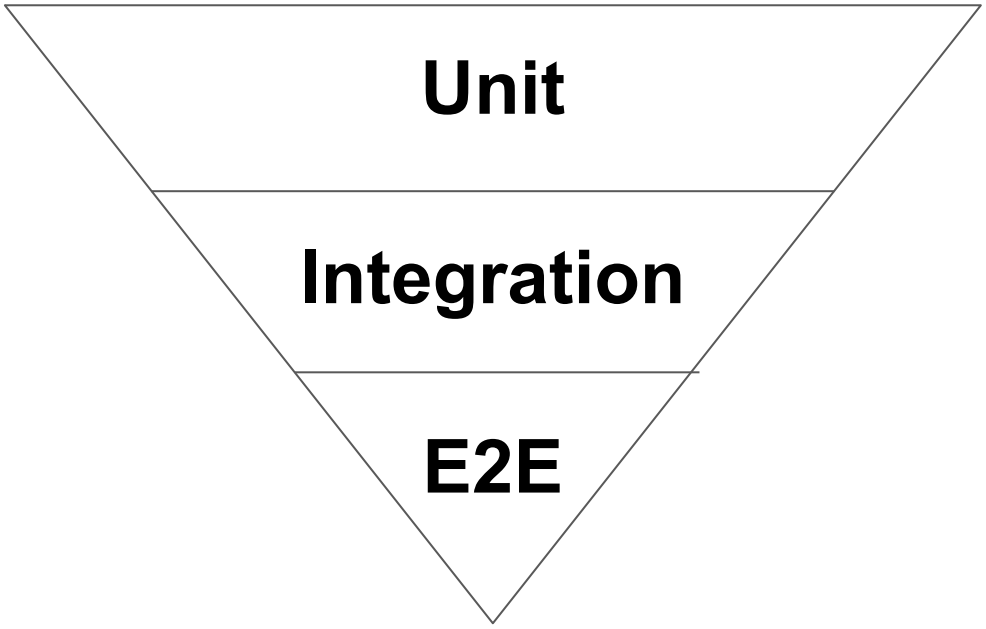
E2E (end-to-end) tests
simulate real-user scenarios

These form the 'testing pyramid'



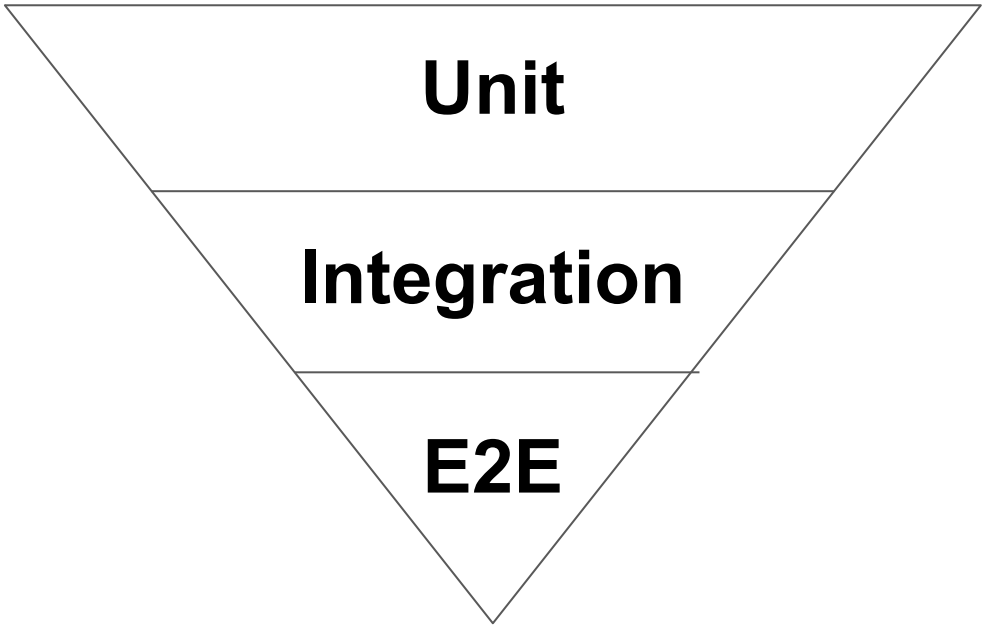
(1) What kind of test is this?

Testing whether clicking the logout button on a website clears the cookie set in the user's browser.



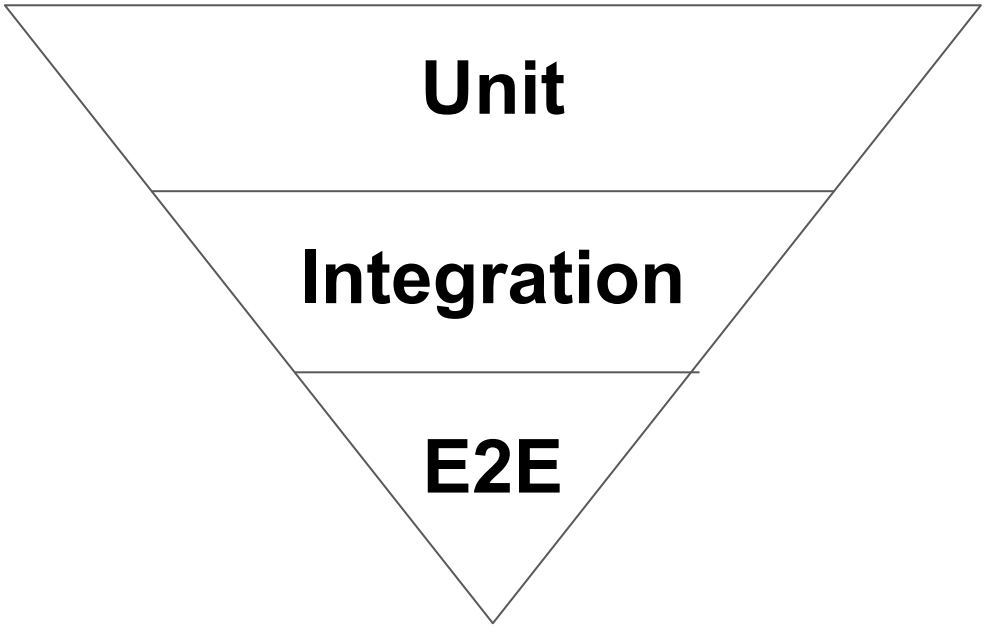
(2) What kind of test is this?

Testing that the `computeShortestPath` function returns a sensible result when there are negative edge-weights in the graph.



(3) What kind of test is this?

Testing whether the room booking system is able to query a user's calendar correctly



Unit testing demo

```
static long calculateAgeInDays(String dateOfBirth) {
    Instant dob = dateFormat.parse(dateOfBirth).toInstant();
    Instant currentTime = new Date().toInstant();
    Duration age = Duration.between(dob, currentTime);
    long ageInDays = age.toDays();
    if (ageInDays < 0) {
        return 0;
    }

    return ageInDays;
  }
}
```

Unit testing takeaway points

Design for test: dependency injection

Test naming

One property per test

Arrange, Act, Assert

Writing assertions

JUnit lifecycle

Using @Before vs constructors

Mocking can be used to simulate a dependency

```
1       import static org.mockito.Mockito.mock;
2       import static org.mockito.Mockito.when;
3       import static org.mockito.Mockito.verify;
4
5       LinkedList mockedList = mock(LinkedList.class);
6
7       // can specify behaviour that you want
8       when(mockedList.get(0)).thenReturn("first");
9
10      mockedList.add("added");
12      // assert that things got called
11      verify(mockedList).add("added");
```

Integration and E2E tests are more complicated

Testing whether clicking the logout button on a website clears the cookie set in the user's browser

1. Start up a test instance of the server
2. Start a webdriver
3. Login to the site and collect the session cookie
4. Simulate a click on the logout button
5. Check the response from the server contains the directive to clear the cookie

A 'flaky' test will pass and fail on the same code

non-hermetic reliance on external systems

more complex tests tend to be more flaky

	% of tests that are flaky
All tests	1.65%
Java webdriver	10.45%
Android emulator	25.46%

https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html

Automated test generation can find unnoticed bugs

Many approaches

One example is random testing

- Generate inputs at random
- Use search to refine these inputs to make them more effective
- Check for 'bad things' like a buffer overflow
- See https://github.com/google/oss-fuzz - found thousands of security vulnerabilities in open source code

How good are my tests?

Code coverage detects how much code you execute

(Demo)

100% coverage does not mean bug-free!

```
public static void xPlusYMinusZ(double x, double y, double z) {
    double t = x + y;
    return t - z;
}

@Test
public void xPlusYMinusZ_correctlyCombines_smallNumbers() {
    double r = xPlusYMinusZ(2.0, 2.0, 2.0)
    // check floating point values with error tolerance...
    assertThat(r).isWithin(0.1).of(2.0);
}
```

This has 100% coverage but the code still has a bug...

Test coverage can use various properties

```
1       if (a == 0) {
2           ...;
3       }
4       else {
5           if (b) {
6               ...;
7           }
8           if (c) {
9               ...;
10          }
11      }
```

Statement coverage: all lines were executed

Branch coverage: all decisions were explored at every branch

Path coverage: all paths through the program were taken

Data flow coverage: is every possible definition tested

Mutation testing can tell us how robust our tests are

Generate small changes to the program under test

- change + to a -
- change constant term
- negate a condition

Verify that this causes a test to fail

Integrating testing into your software engineering process

Defects in software are inevitable

Expect 1-25 errors per 1000 lines for delivered software

80% of errors are in 20% of the project's classes

See Steve McConnell, "Code Complete" 2nd edition, p521, p517

Defects in software are inevitable

Expect 1-25 errors per 1000 lines for delivered software

- when we find a problem we need to know we've fixed it
- once we fix a bug it needs to stay fixed

80% of errors are in 20% of the project's classes

- if we can't test everything then prioritise the error prone parts

See Steve McConnell, "Code Complete" 2nd edition, p521, p517

Continuous integration automatically runs tests

Don't want broken code committed to the repository

Run test suite on every change: can reject changes which break tests or just report

Regression testing preserves existing functionality

1. Write tests that exercise existing functionality
2. Develop new code
3. Run tests to check for regressions

Regression testing helps with bug fixing

1. Write test that reproduces bug
2. Check that it fails
3. Fix bug
4. Check that test passes

We can't run all the tests on every change

Google has 4.2 million tests and 150 million test executions every day

Need to deliver results to developers quickly

Need to manage the execution cost of running tests

See "The State of Continuous Integration Testing @Google"

Test suite minimisation

Choose a subset of tests which achieve coverage on the project

Test set selection

Choose a subset of tests which are appropriate for the change submitted

Test set prioritisation

Choose an ordering such that tests more likely to find a defect are run earlier

Example: test suite minimisation

Select a minimal subset of tests which maximise coverage over the project

NP-complete problem so use heuristics

If some test is the only test to satisfy a test requirement then it is an *essential test.*

1) Choose all the essential tests
2) Choose remaining tests greedily in order of coverage added

Test Driven Development uses tests as specification

1. Write tests which demonstrate the desired behaviour
2. Implement new functionality
3. Check tests now pass
4. Repeat

Pros: guarantees that you write tests and that your code is testable, tests can be written that directly describe the customer's requirements.

Cons: early commitment to how the project will work, changes in approach are hard, some areas are more important to test than others.

Objectives

1. Identify different types of test
2. Be able to write a 'good' unit test
3. Know about some techniques for measuring test quality
4. Understand how testing fits into the software development process

...program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence…

--- E. W. Dijkstra