Software testing

Software systems are only useful if they do what they are supposed to do. This is increasingly important given the vital safety-critical roles modern software systems play. Unfortunately, proving that a system is correct is exceedingly difficult and expensive for large software systems. This leads to the fundamental tradeoff at the heart of most commonly applied testing approaches:

> Given a finite amount of time and resources, how can we validate that the system has an acceptable risk of costly or dangerous defects. [Bob Binder]

Given the constraints above, the most prevalent quality validation approach in use today is automated testing (followed by code reviews). Unfortunately, as Dijkstra noted in 1969:

> Program testing can be used to show the presence of bugs, but never to show their absence! [Edsger Dijkstra]

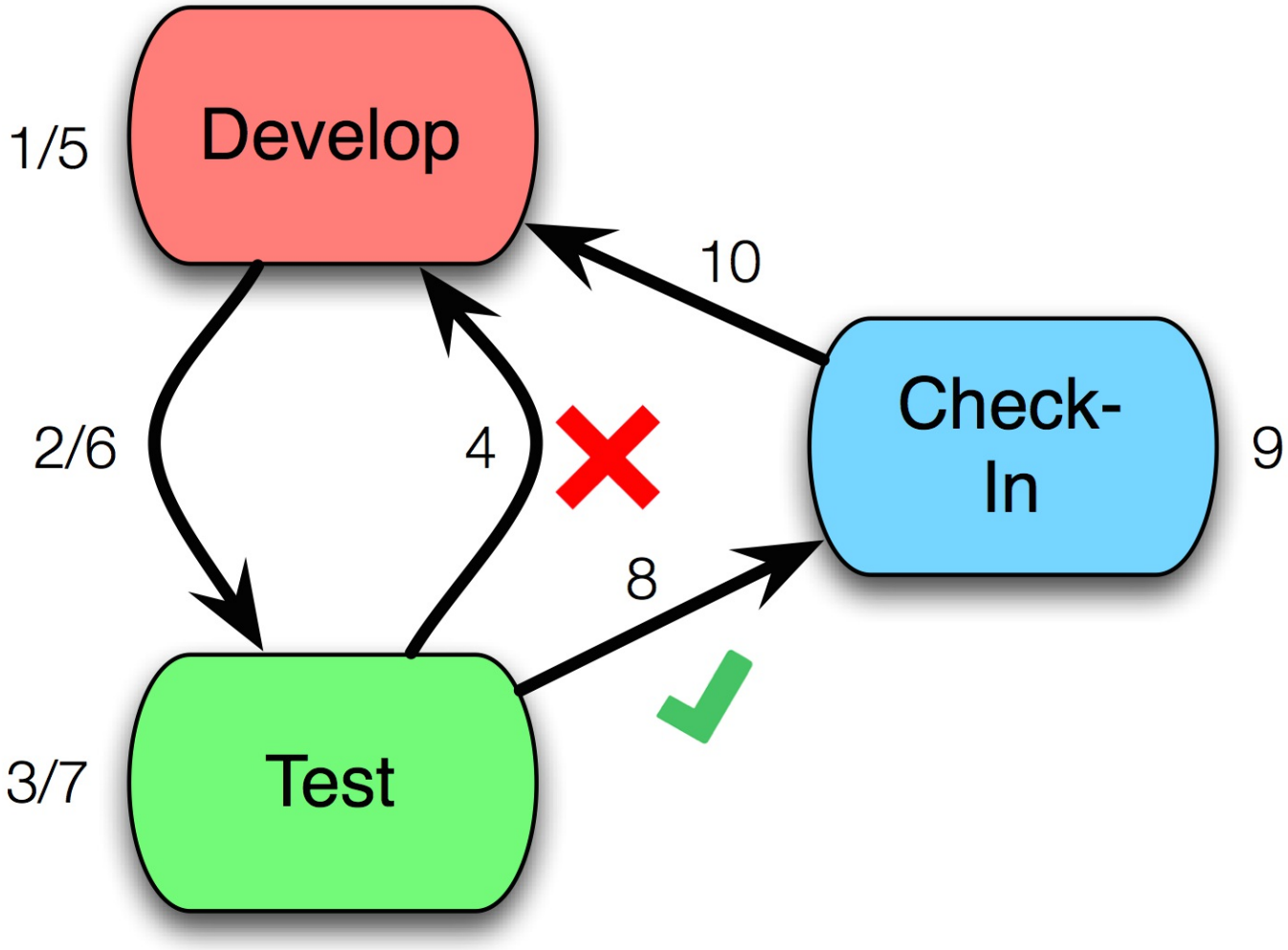
This is because what we are evaluating is whether the probability that there are no bugs in a system given that the tests pass is not actually 0. Since tests themselves are programs (and can have bugs) and specifications are often incomplete or imprecise we therefore must admit that the chance of a defect slipping through a set of tests is > 0 (online discussion).

The modern test cycle is summarized below. The cycle between steps 1-7 can occur many times before the code is ready to commit.

1. Develop some new code.
2. Deploy to testing environment.

3. Run your tests.
4. Test fails.
5. Try to fix the failure through more development.
6. Deploy to testing environment.
7. Run the tests again.
8. Test passes.
9. Commit/push change.
10. Pull new changes.

Steps 2/7 may not happen on small teams or when testing happens solely on a single development machine. The code that is developed in steps 1/5 could involve either writing test or product code. In Test-Driven Development (TDD) the first develop step would be to write tests for features that haven't been written yet to guide future development.



Terminology

A number of different terms are commonly used in the testing space:

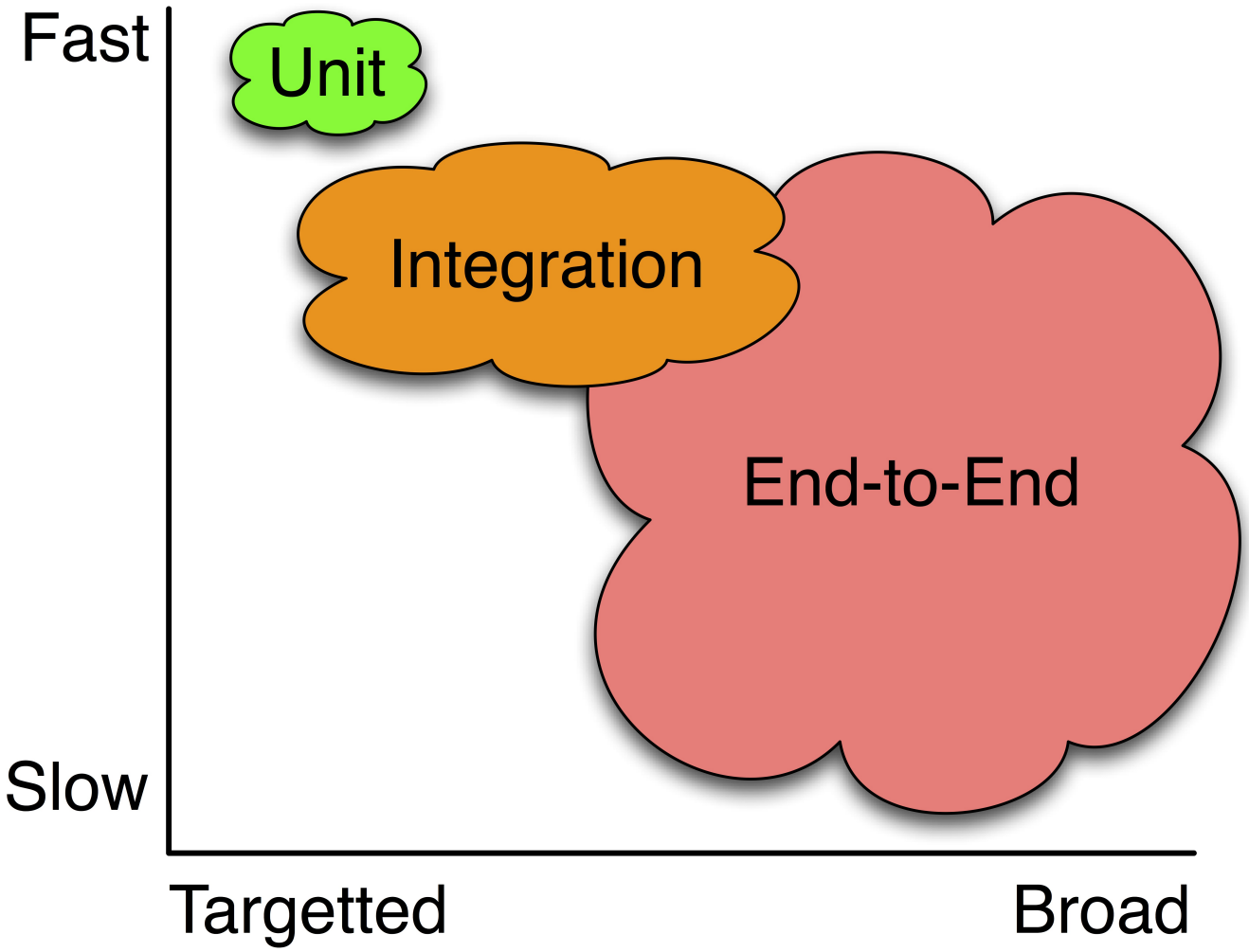
- `SUT/CUT`: System / code under test. This is the thing that you are actually trying to validate.
- `White box testing`: When testing in a white box manner one typically carefully examines the program source code in order to identify potentially problematic sets of inputs or control flow paths.
- `Black box testing`: Black box testing validates programs without any knowledge of how the system is implemented. This form of testing relies heavily on predicting problematic inputs by examining public API signatures and any available documentation for the CUT.
- `Effectiveness`: The simplest way to reason about the effectiveness of a test or test suite is to measure the probability the test will find a real fault (per unit of effort, which can be something like developer creation / maintenance time or number of test executions).
- `Higher/lower testability`: Some systems are significantly easier to test than others due to the way they are constructed. A highly testable system will enable more effective tests for the same cost than a system whose tests are largely ineffective (or require an outsized amount of creation and maintenance effort).
- `Repeatability`: The likelihood that running the same test twice under the same conditions will yield the same result.

There are a number of different *levels* of test; these range in size, complexity, execution duration, repeatability, along with how easy they are to write, maintain, and debug.

- `Unit`: Unit tests exercise individual components, usually methods or functions, in isolation. This kind of testing is usually quick to write and the tests incur low maintenance effort since they touch such

small parts of the system. They typically ensure that the unit fulfills its contract making test failures more straightforward to understand.

- `Integration`: Integration tests exercise groups of components to ensure that their contained units interact correctly together. Integration tests touch much larger pieces of the system and are more prone to spurious failure. Since these tests validate many different units in concert, identifying the root-cause of a specific failure can be difficult.
- `End-to-End`: Also referred to as *acceptance* testing, these tests typically validate entire customer flows. They are especially useful for validating quality attributes (such as performance and usability) that cannot be captured in isolation. These tests are great for ensuring that the system behaves correctly but provide little guidance for understanding why a failure arose or how it could be fixed.
- `A|B`: A special subset of testing that is typically employed in production environments. In A|B testing, two different versions are compared at runtime to validate which one performs 'best'. A|B testing is often used to validate business decisions, rather than evaluate functional correctness.
- `Smoke/Canary`: This is a subset of a test suite that typically executes quickly, is highly reliable, and has high effectiveness. Smoke tests try to expose a fault as quickly as possible, making it possible to defer running large swaths of unnecessary tests for a system that is known to be broken (for example, if you have an error in your data model that touches your whole system, there is no need to run slow integration and end-to-end tests).



For additional reference, take a look at this in-depth talk about how Google tests their systems.

Testability

In order to be successful, a test needs to be able to execute the code you wish to test, in a way that can trigger a defect that will propagate an incorrect result to a program point where it can be checked against the expected behaviour. From this we can derive four high-level properties required for effective test writing and execution. These are controllability, observability, isolateablilty, and automatability.

It is often necessary to restructure software that has not been written in a testable way to make it possible to validate a system effectively; this commonly happens when units within the code take on more than one responsibility or when features become scattered across a codebase.

One of the biggest advantages of Test-Driven Development (TDD) is that by writing your tests first you are able to ensure that your system is structured in a testable way.

Controllability

Tests dynamically execute the system. If the CUT cannot be programmatically controlled, an automated test cannot be written for it. There is often a controllability tradeoff between what code it is *possible* to write a test for and what code it is *efficient* to write a test for. For example, if you build a large system that embeds all of its logic tightly with its interface it can be impractical to validate how the system works without writing UI-tests, which are error prone and expensive.

Observability

Sometimes the CUT can be invoked by a test but its outcome cannot be observed. For example, if a defective method mutates some external inaccessible object but does not return a value. Often these can be resolved by returning data to the callee that might otherwise just been passed further down a call chain. One surprising challenge when considering observability is determining what values are correct and what values are erroneous. While this sounds simple, this is the root cause of the old saying "That's not a bug, it's a feature!"; at their core these are often caused by imprecise, incomplete, or contradictory specifications.

Isolateability

Being able to isolate the CUT under test is crucial to be able to quickly determine what has caused a failure so it can be resolved. This is challenging in large modern systems due to the number of (often third party) dependencies software systems have. For example, if a data

access routine fails is it the logic in the routine or is it a failure in the underlying database? The most common approach to solving these is through simulation. In simulation-based environments code dependencies are *mocked* or *stubbed* whereby they are replaced with developer-created fake components that take known inputs and return known values (e.g., a `MockLoginRejectController`) would always return false for a `login(user, pass)` without needing to check a user store, database, or external system. In this way the developer can test their code that uses `login(..)` and ensure it handles the false case correctly without fear that a bug in the real login controller may return an incorrect or inconsistent value. In addition to isolation, mocking also greatly increases performance and makes components less prone to non-determinism as the result being returned is usually fixed and not dependent on some external complex computation. Mocking can also make it possible to test program states that would otherwise be hard to trigger in practice (for instance if you want to test a situation where a remote service is down you can have a `MockTimeoutService` that just does not respond to requests).

Automatability

The above properties all aim to support making it possible to automate the execution of the test suite. Suites that can run without human intervention are much more likely to be executed than those that must be manually performed. They are also more likely to be run after a system has been released in the form of a regression suite to ensure that a system continues to execute as expected. All systems exist within organizations, even if the software does not change, external dependencies, libraries, frameworks, and computing infrastructure will which will require continuous validation of even deployed systems. The economic benefits or automation are also clear: even if it takes 5 hours to configure an automated test case that can be performed in 30

minutes, the suite will break-even (in terms of time) after only 10 test iterations. Automated suites can be run online as changes are made to a system, after every commit, nightly, or weekly, depending on the needs of the system. These suites also provide global visibility of the state of the system; if the suite has been 'green' for a week a team might feel more comfortable about deploying it than for a system whose suite has been failing for the same period of time.

More details about testability can be found in Bob Binder's GTAC talk. Miško Hevery has a cool walkthrough video of live coding a refactoring to enable a simple system to be more testable. A nice collection of testability links have been compiled on StackOverflow.

Why not test?

There are a number of reasons why software systems are not tested with automated suites. These range from "bad design" to "slow", "boring", "doesn't catch bugs" and "that's QA's job". Ultimately, testing does have a cost: tests are programs too and take time to write, debug, and execute and must also be evolved along with the system.

One core paradox of automated test suites is how they are used. If you think that a test only provides value when it fails by catching a bug that would have otherwise made it to production, then that all passing tests are just a waste of computational cycles. At the same time though, passing tests *do* give us the warm feeling that we have not introduced unexpected regression bugs into our systems. But at the same time, these feelings only work if we believe our test suite is capable of finding faults and *could* fail. Yes: this contradicts the 'we should only run failing tests' implication above.

An area of interesting future research would be to figure out how often a

suite needs to fail to impart trust, while also figuring out how few passing tests you could run to have a sense of confidence in your automated suite.

Common testing assumptions

It has been long held that the cost of fixing a fault rises exponentially with how late in the development process (e.g., requirements, design, implementation, deployment) the fault is detected. This statement arises from several influential studies Barry Bohem performed in the 60s and 70s. A more in-depth description of these costs can be found here or here.

There is a more modern discussion however that openly wonders if these costs are not relevant for more modern-styles of software development and programming languages.

It is also important to accurately consider the costs of automated testing. Writing tests takes time. Executing tests takes time. Diagnosing failing tests take time. Fixing tests that were incorrect takes time (tests are programs too!). But catching real faults can save money, as can increased velocity enabled through better understanding how shippable your code is. While it is easier to account for the costs than the benefits, most large teams have invested heavily in automated test suites, although many large teams are actively investigating test prioritization and minimization schemes to reduce the delay between making a change and having a meaningful test result.

Assertions

A key skill when creating automated unit tests is evaluating the *correctness* of the code under test. While a compiler can validate the

syntax of a program, a test suite is required to ensure its runtime behaviour matches its specification. Assertions are the primary piece of machinery used to evaluate correctness within automated unit test suites. Given that jUnit was one of the early (and still most popular) unit testing frameworks, it is unsurprising that many modern frameworks provide features similar to jUnit's assertion features for evaluating program outputs.

There are several high-level models to consider when structuring tests; two of the most widely used are:

- Four-Phase Tests: This model is supported by default by jUnit and its variants. The four phases correspond to 1) setting up the testing environment; 2) executing the code under test; 3) evaluating the output of the CUT; and 4) tearing down the testing environment.

- Give-When-Then: This model was developed by the behavioural driven programming (BDD) community which strives to create 'executable specifications' through unit tests that use descriptive strategies for nesting and naming. Tests are structured from some *given* state, where the system's configuration is understood. The key action of the test is the *when*; this is often described in the test name using plain language (e.g., `it('should be able to parse a document that has UTF-16 characters')`). The *then* step involves observing the output to ensure its correctness. Expect/Should-style assertions (such as those used by the Chai library) are also often used by the BDD community as they enable extremely descriptive assertion statements.

Chai's Expect/Should Assertions

The Chai expect library provides a rich set of assertions that can be used

to check for correctness. While traditional assertions are supported, e.g.,

```
expect('myName').to.equal('myName');
expect(1).not.to.equal(2);
```

Chai also includes support for more advanced checking:

```
expect(obj).to.deep.equal({ key: 'value' }); // check large object
expect(42).to.be.a('number'); // check type
expect([ 42, 97, 102 ]).to.have.length.above(1); // check array pr
expect([1, 2, 3]).not.to.include.members([1, 4]); // checks set me
```

One thing to note about the syntax of the assertions above is that they are extremely readable. For example, even without comments, what the following Given-When-Then test doing is straightforward to understand:

```
describe('Check math constants', function() {
    it('Math.PI should be close enough to the correct value', func
        expect(3.1415).to.be.closeTo(3.14, 0.025);
    });
});
```

It is often best to have each behaviour tested as independently as possible. This eases debugging because a change that breaks a behaviour will be easy to isolate as it will only have broken one tests and not a handful. It also eases program evolution because changing a behaviour will not require modifications to lots of different tests.

When creating our tests we can break down the behaviours we are testing three ways:

- Normal conditions: The normal conditions represent the way we expect to be used.
- Unexpected conditions: The unexpected conditions arise when the code is used in unexpected ways.
- Boundary conditions: All programs consume input; by testing boundary values we can ensure our program will successfully operate with the breadth of inputs the program might encounter.

Ultimately each test should compare an expected value against the actual value produced by the code under test. One easy way to keep code easy to read is to make this explicit within the test, for example:

```
describe('Check math constants', function() {
    it('Math.PI should be close enough to the correct value', func
         const expected = 3.1415;
         const actual = Math.PI;
        expect(expected).to.be.closeTo(actual, 0.025);
    });
});
```

Coverage

When evaluating the quality of the system we really want insight into how well the system performs its expected tasks and how robustly it responds to unexpected inputs and situations. Unfortunately, measuring this kind of property meaningfully is impossible, so developers often ask a simpler question:

> Does the test suite test the whole system?

Since the system code is supposed to both perform the expected operations and handle unexpected ones, coverage can make sure we

are testing both of these aspects of the system. At its core, all coverage tools measure the proportion of the system that is executed by the test suite. Coverage has two properties in particular that make it appealing:

1. It is relatively cheap to compute. In contrast to reasoning about expected and unexpected behaviours, a coverage tool simply needs to track which parts of a program have executed when the test suite is executing.

2. It is actionable. Developers can look at the output of a coverage report, find the parts that are not currently covered by the test suite, and decide how to act upon this information in a straightforward way.

There are many flavours of coverage that are used in practice, here we describe three of them:

- **Statement**: This measures the fraction of the statements of the code that are executed. Statement and line coverage are both the simplest forms of coverage. Statement coverage considers every statement within the system.

- **Decision**: Decision (and branch) coverage checks to make sure that all branching options within the system are executed. For example, decision coverage checks whether both true and false branches of `if` statements are executed by the test suite. Decision coverage only measures those program points where execution can take divergent paths within the system.

- **Modified Condition (MCC)**: MCC coverage checks all program entry and exit points along with every combination of decisions within a program.

For example, given the test and code snippet below:

```
1: eval(x: number, c1: boolean, c2: boolean): number {
2:  if (c1)
3:    x++;
4:  if (c2)
5:    x--;
6:  return x;
7: }
```

And the following test:

```
1:    eval(0, false, false);
```

The code has 60% statement coverage (lines 2, 4, 6), 50% decision coverage (only the false branches of the decisions on lines 2 and 4) and 25% MCC coverage. The following suites would provide 100% coverage for each of the different coverage criteria:

```
// 100% statement coverage
eval(0, true, true);

// 100% decision coverage
eval(0, true, true);
eval(0, false, false);

// 100% MCC coverage
eval(0, true, true);
eval(0, false, false);
eval(0, true, false);
eval(0, false, true);
```

The stronger the coverage criteria, the more stringently we can say the

code was covered by the test suite. The greatest shortcoming of coverage however is that while it clearly shows that the suite *executes* the code under test, it does not show that the code under test is *correct*. This can be clearly seen by our tests above, none of them check that the output from `eval` is correct for the inputs given. It is extremely important to keep this shortcoming in mind when thinking about coverage because ultimately we really do care about the correctness of the system over how comprehensively a test suite might execute it.

Other Resources

- Another great talk about how Google stores its source code.
- Steve Easterbrook testing 1.
- Steve Easterbrook testing 2.
- Laurie Williams black box testing notes.
- Laurie Williams white box testing notes.
- Laurie Williams automated testing notes
- MIT testing slides.
- Jim Cordy's quality assurance course.
- Concrete example showing how to increase isolation with mocks and stubs.
- Some best and worst practices for writing unit tests.
- Great article on testability.