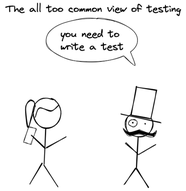
Automated Software Testing

The all too common view of testing



Unfortunately Common Test Style

```
void testMyFunction() {
    String result = myFunction("value");
    assertNotNull(result);
}
```

Code coverage passed

Problems with common testing approaches

- Tests are treated as an afterthought
- Tests are viewed as busywork
- Tests are often written to meet metrics
- Focus on semantics instead of value

Testing Tips

Keep tests as close to the code as possible

Unit and Integration level tests, with a narrow scope are easier to use for edge cases and should be the vast majority of tests

Execute tests before merges

The most valuable tests are those that are executed before code is committed. This prevents bugs from being deployed and avoids the turnaround time of fixing issues that have been merged.

Test methodically

Start with a case that passes all nulls/0s/etc and slowly build up to a test with fully populated data. You'll often find edge cases you didn't expect

Don't use conditional logic

Tests should be explicit about inputs and expected outputs. Having conditional logic makes the intent of the test less clear and introduces the potential bugs. Prefer to create multiple tests instead.

Don't repeat logic that's in the code under test

Provide explicit expected outputs instead of outputs calculated in the same or similar way to code under test. E.g.
DON'T assert add(1, 3) == 1 + 3
DO assert add(1, 3) == 4

Test both success and failure

For almost any method there's at least one success case and one failure case. Usually there are many more than that. Testing even simple failure cases documents the expected behavior for other engineers

Test Driven Development

- Can be a useful way to approach coding
- Helps define testable code more easily
- Very useful for handling bug fixes and changes
- Doesn't work for all coding styles

Testing Stages

Setup

This is where you gather the data you're going to use for a test. It should instantiate any necessary classes or services and setup any mocks.

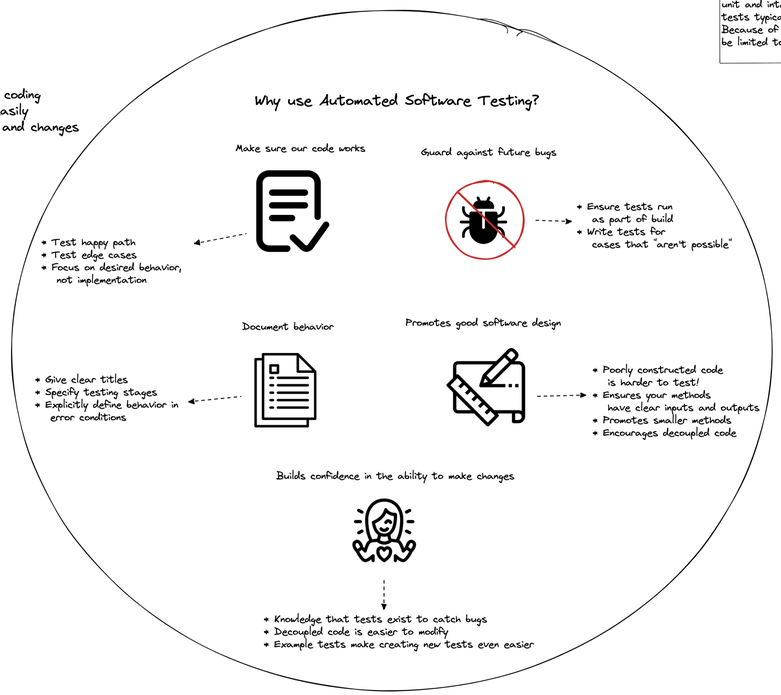
Execution

This should be a very short section of code, explicitly executing the code under test. For most tests this will be a single method or API call. This section should be clearly defined so future users can see exactly what's being tested.

Assertions

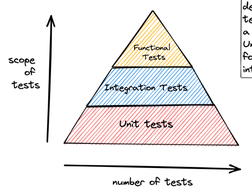
This is where you check the output or behavior of the code you tested. It should assert outputs and mock interactions

Why use Automated Software Testing?



Our arbitrary definitions of test types

Functional tests are tests that are run against a deployed environment. Unlike unit and integration tests, functional tests typically don't use any mocks. Because of this their scope should mostly be limited to how systems interact.



Integration tests have many different definitions. Here, we'll define integration tests as tests that can be run against a local service or ad-hoc environment. Unlike unit tests, they are typically more focused on how classes and methods interact with each other.

Unit tests are the foundation of a good testing strategy for a project. They are focused at the lowest level, typically at a method or class level. They have clear inputs, clear expected outputs and are typically focused on a single method.

Things that Don't Matter

- Which type of test you write
- Your test coverage
- Whether or not you used TDD

Things that do matter

- Scope of tests
- Quality of tests
- Tests are created and run before merge

DAO Tests

As part of a decoupled design an application should have a DAO layer that is responsible for all interactions with the database.

To properly test this layer, there should be tests that hit the DAO object and ensure that the queries interact correctly with the datasource.

This is true regardless of if you use Hibernate/JOOQ/vanilla JDBC/whatever.

- Executes against a single DAO class
- Uses a container for hitting a real datasource
- Runs as part of local test suite
- Runs before building artifacts
- No mocking should be necessary

Endpoint Tests

For backend services, endpoint tests cover a full path of functionality from an entry point to any backend dependencies, for example a call to an external service.

- May or may not mock out DAO layers to allow for easier edge case setup
- They do mock out any services that can't be run as part of a local build to ensure repeatable, contained testing.
- The focus of endpoint testing should be to ensure that a full path within a service works as expected.

Service Connectivity Tests

Code that directly connects to external tools such as message queues or databases should be isolated within a codebase.

This code can be tested by spinning up a running version of that external tool and doing simple tests to ensure the code can communicate with it.

A different view of test types

Local Development

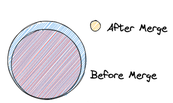
- Can be run during local development
- Run quickly / automatically
- Can be used for TDD

Before Merge

- Can be run on a Continuous Integration systems
- Ensure that tests are run before code is merged to main branch

After Merge

- Can test Functionality that cannot be run in an isolated environment
  - integration with other services
  - tests that run in production / against production data



- functions
- function interactions
- service integrations
- production data

Testing Legacy Code

Define current state

Building tests around current functionality can build an understanding of current state. This can often give insight into current behavior or bugs.

Write them even if they're messy

Untested, legacy code may not be easy to test. In these cases, some of the principals outlined previously may need to be compromised in the interest of creating coverage

Refactor against tests

Once there is some level of coverage on legacy code it's easier to make changes to structure and be confident that it doesn't break things

Create better tests, repeat

After refactors it should be easier to test pieces of functionality and add new conditions. Continue as necessary.

Automated Software Testing

- Repeatable
- Scripted
- Automated