UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Effective Software Testing

Liam Keegan, SSC

Course Outline

- Testing
  - Benefits of a good test suite, difficulties of creating one
- Types of tests
  - Different types of tests, their use cases
- Best practices
  - How to write good tests, with examples
- Testing strategies
  - Strategies for testing new and legacy code
- Hands on with pytest
  - Github sample Python repo with test suite & continuous integration

Testing

What is testing anyway?

There are many ways in which all software is tested:

- It is “tested” every time it is used and produces some output
- It was probably tested with some sample inputs when written
- There is probably some manual testing when changes are made

However this kind of testing can quickly become both insufficient and inefficient as a software project grows in complexity.

Changing the code risks breaking things that previously worked without notice, and manually testing all the functionality quickly becomes an impossible task

Automated tests

Many software projects also have automated tests

- A test is a piece of code that tests some behaviour of the software
- A test suite is a collection of such tests
- The test suite typically runs automatically whenever a change is made
- The more complex the project, the more value a good test suite provides
- But a test suite is not only for large projects!

This is the kind of testing we will learn about in this course.

A good test suite provides many benefits

- Ensure correctness of your code when you write it
- Maintain correctness of your code as things change around it
- Make changes or refactor code without fear
- Find bugs earlier and more easily
- Easier for new contributors to make positive changes
- Complements the documentation as examples of use
- Gives others confidence in the correctness of your code
- Encourages well-designed modular code

So why doesn’t every project have one?

- Requires upfront investment to create
- Changing code (often) also requires changing tests
  - Mitigated by writing good tests
- Slows (initial) speed of development
  - But speeds up later development and improves quality
- Hard to retrofit to legacy code
- Bad tests can be worse than no tests
  - Flaky tests can waste time or result in test failures being ignored
  - False positive tests can give false sense of security
  - Brittle tests can fail due to unrelated code changes

Same scenario, different tests

- Imagine you have an existing, working software project
  - You want to add a small feature
  - You inadvertently break some other functionality with your changes
- With only manual tests
  - You don’t find this with your manual testing of the new functionality: Bad!
- With a brittle test suite
  - The tests for the broken functionality fail: Good!
  - So do a bunch of unrelated tests for unrelated reasons: Bad!
  - Either you waste a lot of time fixing all these unrelated test failures
  - Or you ignore them and miss the actual bug in all the noise
- With an effective test suite
  - The tests for the broken functionality fail: Good!

Types of Tests

Types of tests

- Unit tests
- Integration tests
- System tests
- Regression tests
- Approval tests
- Smoke tests

Unit tests

- Small, self-contained test of a piece of functionality
- Narrow scope: typically a class or method
- Fast to run
- Doesn’t depend on any other components
- Dependencies often replaced with mocks or doubles
- Typically most (e.g. 80%) of your tests should be unit tests
- Primary way of testing correctness
- Should be written alongside coding the functionality being tested
- Failing unit test directly tells you what has gone wrong

Integration / System tests

- Also known as “end-to-end” or “functional” tests
- Tests involving multiple interacting components
- Should be a smaller fraction (e.g. 20%) of your test suite
- If most tests end up as integration tests, consider making code more modular

Regression tests

- Tests written when a bug is found
- Can be unit or integration tests
- They initially demonstrate the existence of the bug
- Once the bug is fixed, they ensure it doesn’t come back
- Note:
  - The phrase “regression testing” is also used to describe running the tests after changing the code to check for any regressions

Approval tests

- Useful when dealing with legacy code that lacks tests
- Run a function, record the output, test that it gives this output
- This is not how you would write a test for new code!
- These are low-quality tests
- But quick to create and can be done without deep understanding of the code
- Then you can start to refactor or make changes
- At least you get a test failure when the outputs change
- Note:
    - “approval testing” is also used to describe the tests a customer might make of a piece of software they commissioned

Smoke tests

- Basic sanity check: does it run without crashing?
- Electronics analogy: is there smoke when you switch it on?
- Plumbing analogy: fill pipes with smoke, does it escape?

Best Practices

Goal: unchanging tests

- When we change the code, do we need to update our tests?
- Types of changes to the software:
  - Refactoring: internal implementation changes
  - New feature: add some new behaviour
  - Bug fix: fix a bug that was found
  - Change behaviour: change the existing behaviour
- Ideal case: only changes in behaviour should require tests to be updated
  - Refactoring: no change to existing tests, no new tests
  - New feature: no change to existing tests, add new tests for new behaviour
  - Bug fix: no change to existing tests, add new tests for bug
  - Change behaviour: update existing tests

Don’t test unrelated things

- Don’t assert things unrelated to the thing you are testing
- Avoid assumptions about the internal structure of the code

Why?

- Avoid the test becoming brittle / noisy
  - Unrelated changes should not cause the test to fail
- Make the test more maintainable
  - Unrelated changes should not require the test to be updated
- Make the meaning of the test clear
  - Test failure should tell you what broke and what needs to be fixed

Test using public APIs

- Tests should use the same interfaces as user code
- Tests should not use or rely on private implementation details
- Also known as “black box” vs “white box” testing

Why?

- Avoids needing to update tests when internal implementation changes
- More realistic / representative use of code
- Tests can serve as examples of use / documentation

Test state, not interactions

- A test can check state: state of the system after some actions (i.e. what)
- Or it can check interactions: which actions did the system do (i.e. how)
- It is better to test state than interactions

Why?

- Similar to using the public API, less brittle to check *what happened* than *how*
- Avoids needing to update tests when internal implementation changes
- More realistic / representative use of code
- Tests can serve as examples of use / documentation

Test behaviours, not methods

- Having a test for a method often involves testing multiple behaviours
- Better to have a separate test for each behaviour
- E.g. *given* state X, *when* action Y, *then* state Z

Why?

- Keep meaning of a test clear
- Avoid complexity of test for a method growing over time
- See also: “behaviour driven development”

Keep test code simple

- Test code should be “obvious upon inspection”
- Should be complete: contain enough information to understand the test
- Should be concise: don’t include irrelevant information
- Avoid “clever” code, complex control flow, magic numbers, etc
- Some code repetition between tests is ok if it makes test code simpler

Why?

- There are no tests for your tests!
- When a test fails, reading the test code should tell you what is wrong

Testing Strategies

Summary

Summary

-

More resources

- “Software Engineering at Google” book
  - Free html version online
  - Three chapters on testing
  - https://abseil.io/resources/swe-book/html/toc.html