Testing and Test-Driven Development

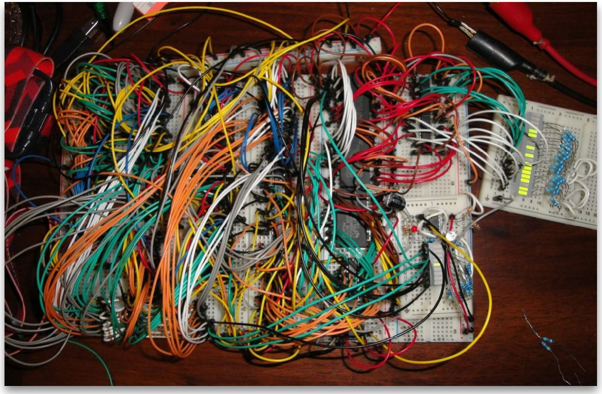
Why do we test?

What do we test?

What IS a test?

Software Quality

Internal Quality



- Is the code well structured?
- Is the code understandable?
- How well documented?

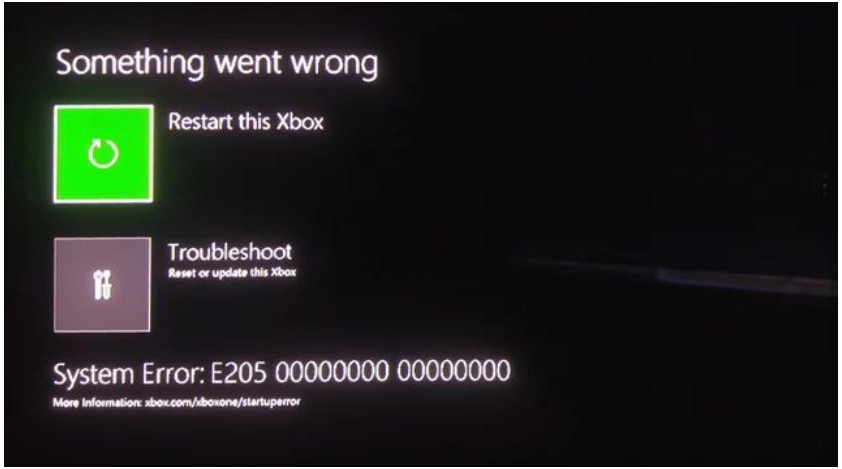
External Quality

- Does the software crash?
- Does it meet the requirements?
- Is the UI well designed?

Testing

Assuring external quality

A problem has been detected and windows has been shut do
to your computer.
The problem seems to be caused by the following file: at
If this is the first time you've seen this Stop error sc
restart your computer. If this screen appears again, fol
these steps:
The device driver got stuck in an infinite loop. This us
problem with the device itself or with the device driver
hardware incorrectly.
Please check with your hardware device vendor for any dr
Technical information:
*** STOP: 0x000000EA (0x890F2020,0x89360460,0xBA517CBC,0
ati2dvag
Beginning dump of physical memory
MPEROR 英皇鐘錶珠寶
TCH AND JEWELLERY
ZENITH
PIAGET



Something went wrong

System Error: E205 00000000 00000000

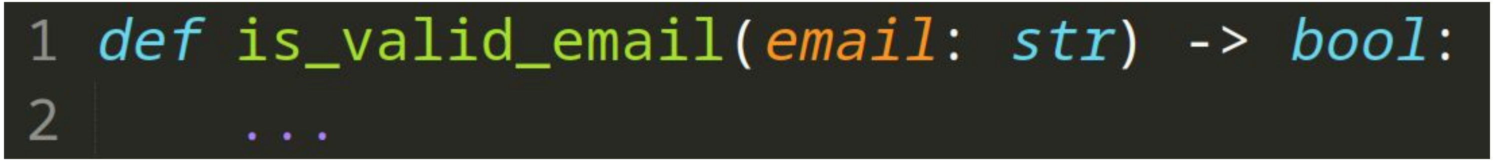
More Information: xbox.com/xboxone/startuperror

Principles of Testing #1: Avoid the *absence of defects* fallacy

- Testing shows the presence of defects
- Testing does not show the absence of defects!
- “no test team can achieve 100% defect detection effectiveness”

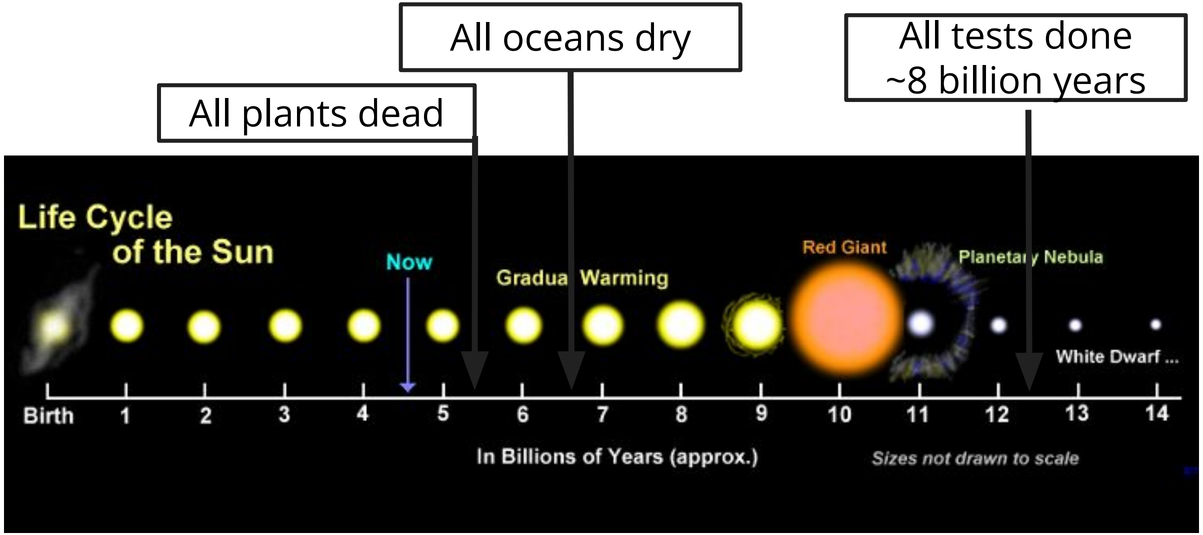
Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #2:
Exhaustive testing is impossible



```python
def is_valid_email(email: str) -> bool:
    ...
```

- A simple function, 1 input, string, max. 26 lowercase characters + symbols (@, ., _, -)
- Assume we can use 1 zettaFLOPS: 10^{21} tests per second



All plants dead

All oceans dry

All tests done ~8 billion years

Effective Software Testing: A developer's guide. Maurizio Aniche

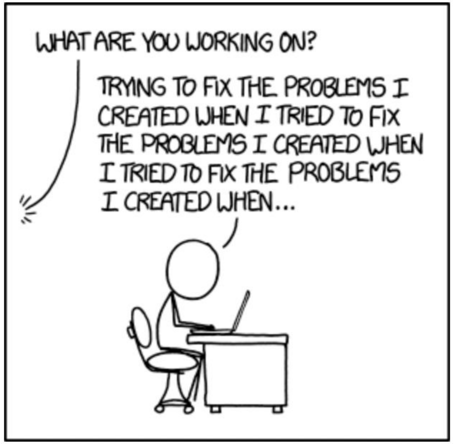
Principles of Testing #3:
Start testing early

- To let tests guide design
- To get feedback as early as possible
- To find bugs when they are cheapest to fix
- To find bugs when have caused least damage

Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #4: Defects are usually clustered

- “Hot” components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, …
- Use as heuristic to focus test effort


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

Effective Software Testing: A developer's guide. Maurizio Aniche

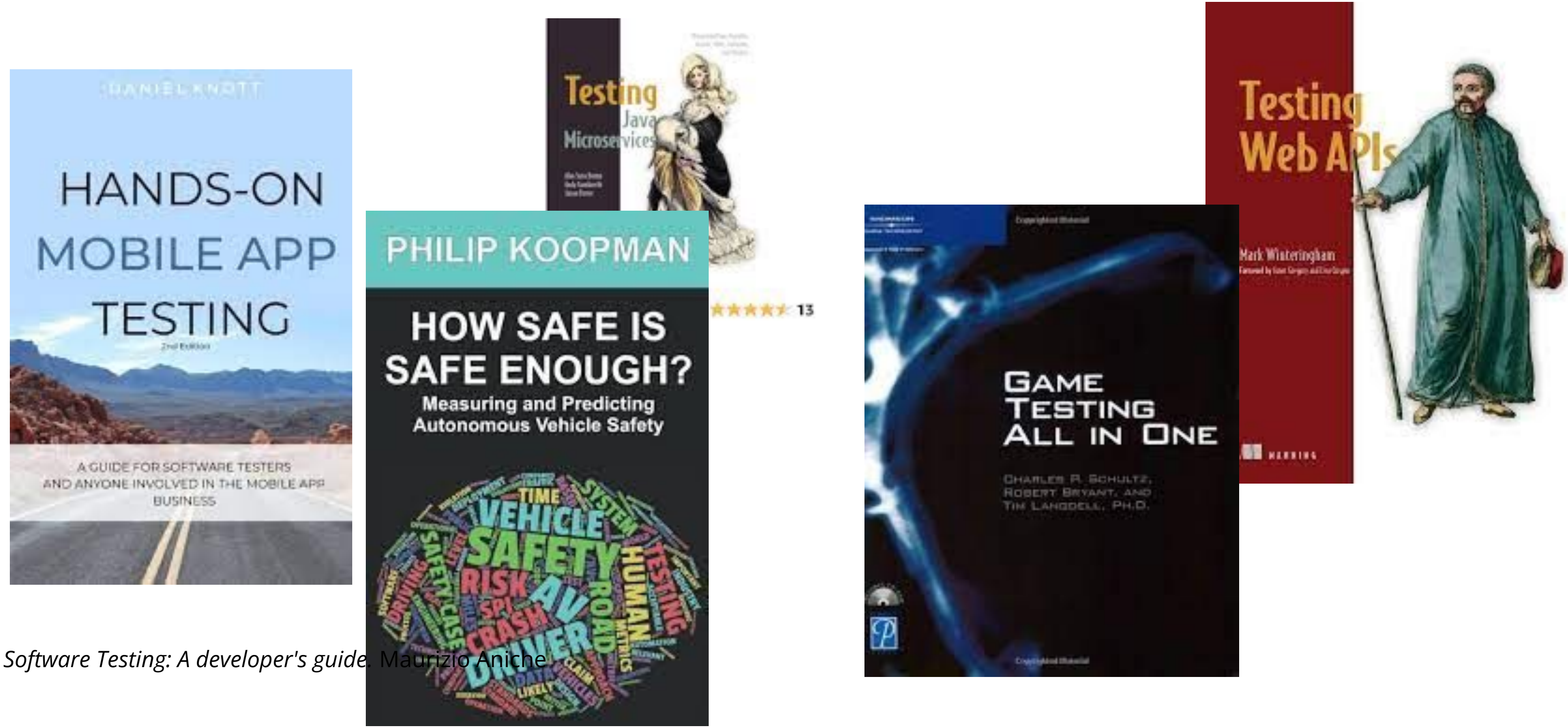
Principles of Testing #5:
The pesticide paradox

“Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.”

- Re-running the same test suite again and again on a changing program gives a false sense of security
- Variation in testing

Effective Software Testing: A developer's guide. Maurizio Aniche

Principles of Testing #6:
Testing is context-dependent


DANIEL KNOTT
HANDS-ON MOBILE APP TESTING
2nd Edition
A GUIDE FOR SOFTWARE TESTERS AND ANYONE INVOLVED IN THE MOBILE APP BUSINESS
Testing Java Microservices
13
PHILIP KOOPMAN
HOW SAFE IS SAFE ENOUGH?
Measuring and Predicting Autonomous Vehicle Safety
GAME TESTING ALL IN ONE
CHARLES P. SCHULTZ, ROBERT BRYANT, AND TIM LANGDELL, PH.D.
Testing Web APIs
Mark Winteringham
MANNING

Effective Software Testing: A developer's guide. Maurizio Aniche

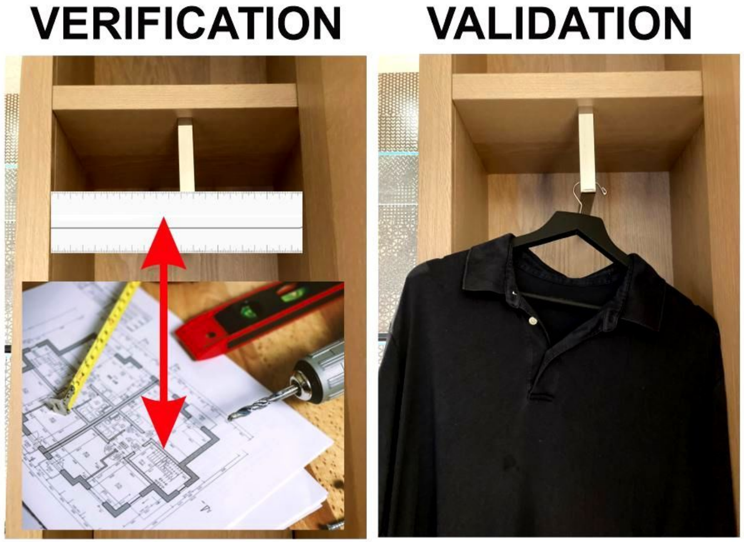
Principles of Testing #7: Verification is not validation

Verification

- Does the software system meet the requirements specifications?
- Are we building the **software right**?

Validation

- Does the software system meet the user's real needs?
- Are we building the **right software**?



Credit: Philip Koopman

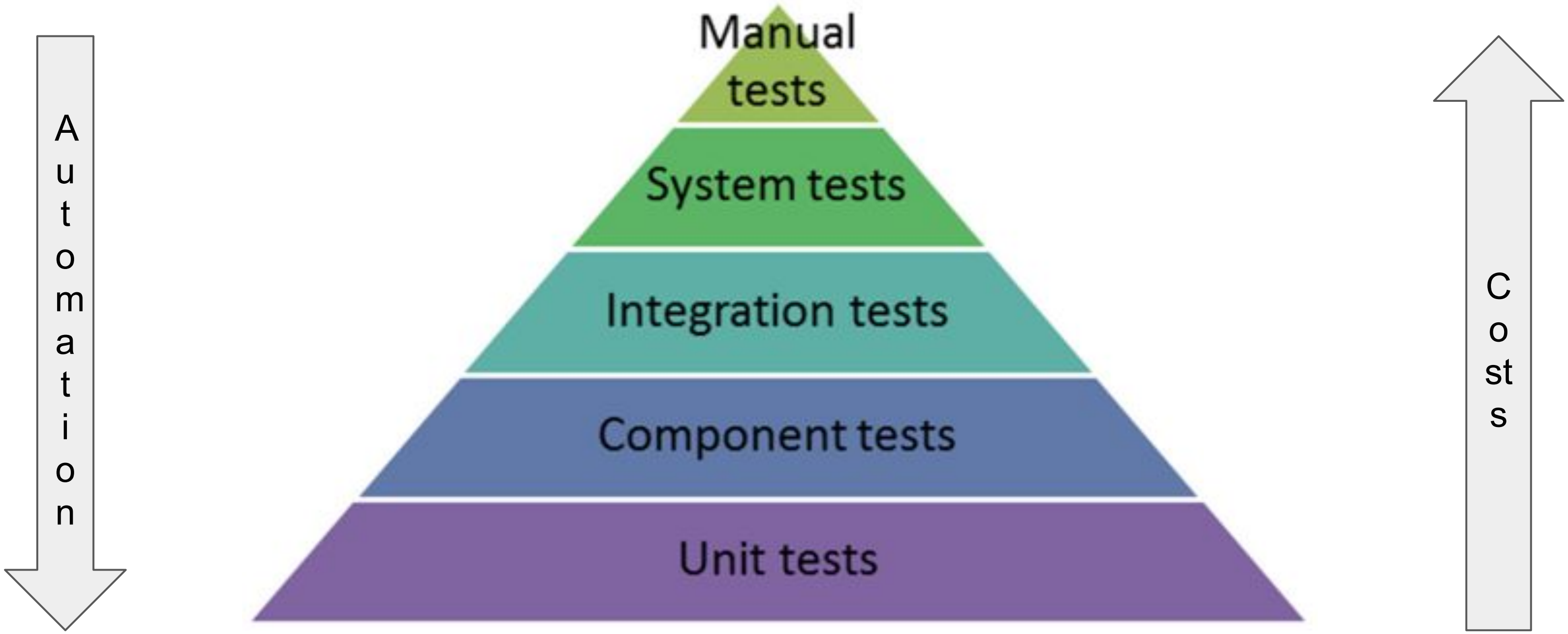
Effective Software Testing: A developer's guide. Maurizio Aniche

How to create tests?

Types of Tests & Testing

- Functional tests
- Unit tests
- User interface tests
- Integration tests
- System tests
- “Black Box” tests (and White Box tests)
- Performance testing
- Security testing
- Fuzz testing
- Database testing

The Testing Pyramid



Automation

Manual tests

System tests

Integration tests

Component tests

Unit tests

Costs

Test design techniques

- **Opportunistic/exploratory testing**: Add some unit tests, without much planning
- **Specification-based testing ("black box"):** Derive test cases from specifications
  - Boundary value analysis
  - Equivalence classes
  - Combinatorial testing
  - Random testing
- **Structural testing ("white box"):** Derive test cases to cover implementation paths
  - Line coverage, branch coverage

What about exhaustive testing?

Idea: Try all values!

- **age: int** (2 - 117) years
- **datetime: DateTime** (hh:mm + M/D/Y)
- **rideTime: int** (in minutes, 1 - 2 Hours)
- **is_public_holiday: bool** (2 values)

116 x 1440 (minutes per day) x 1826 (days in the next 5 years) x 120 (ride time) x 2

~ 72 Billion test cases

What about exhaustive testing?

Exhaustive testing is usually impractical – even for trivially small problem

Key problem: choosing test suite

- **Small enough** to finish in a useful amount of time
- **Large enough** to provide a useful amount of validation

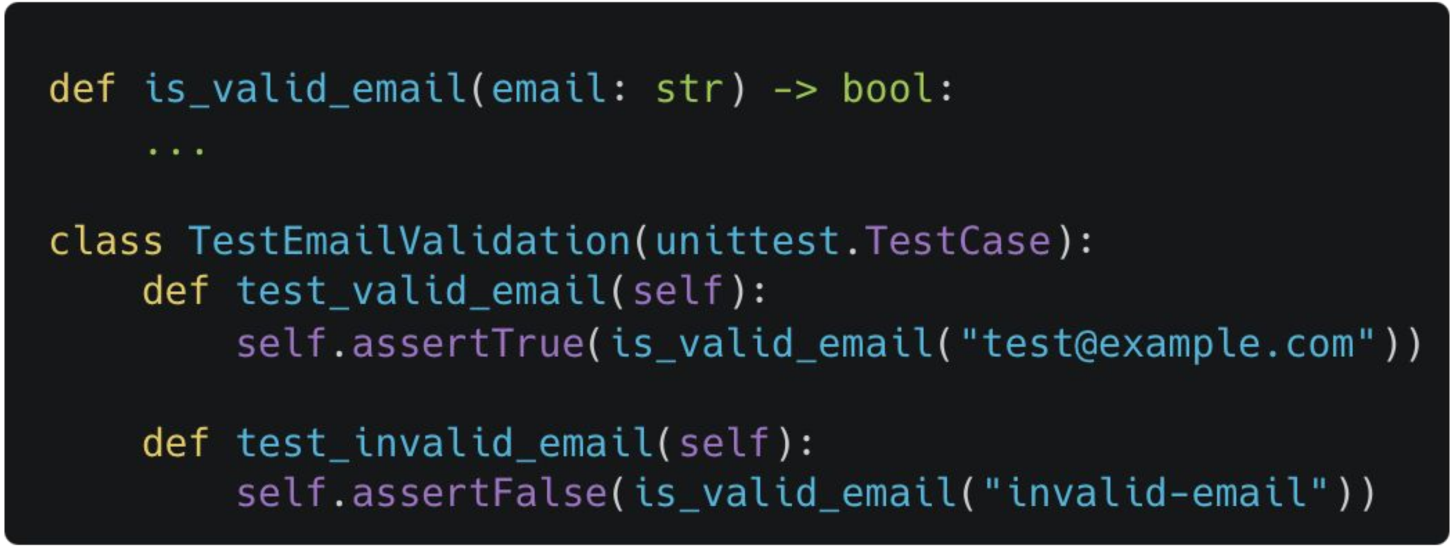
Alternative: **Heuristics**

Boundary-value analysis

Key Insight: Errors often occur at the boundaries of a variable value

- For each variable, select:
  - minimum,
  - min+1,
  - medium,
  - max-1,
  - maximum;
  - possibly also invalid values min-1, max+1

Unit test example

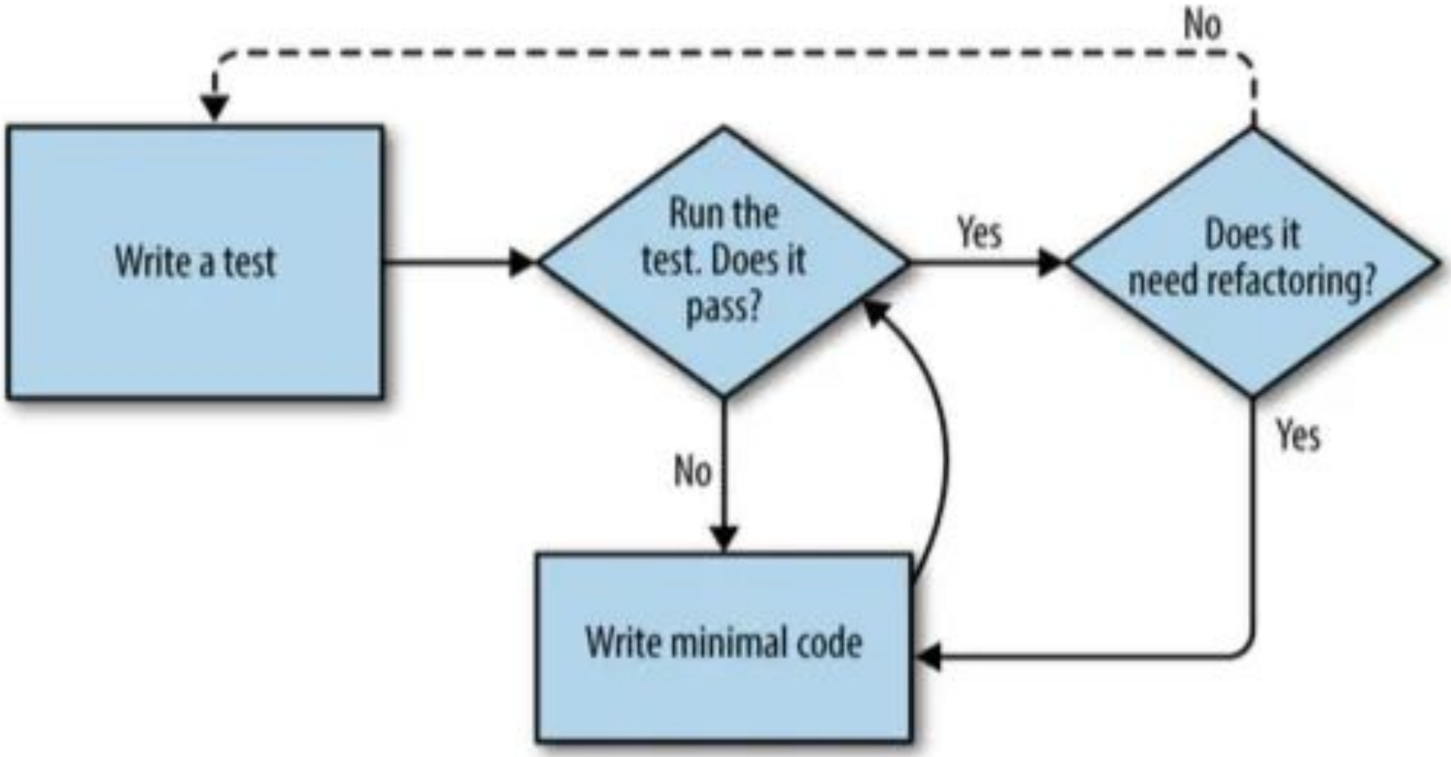


```python
def is_valid_email(email: str) -> bool:
    ...

class TestEmailValidation(unittest.TestCase):
    def test_valid_email(self):
        self.assertTrue(is_valid_email("test@example.com"))

    def test_invalid_email(self):
        self.assertFalse(is_valid_email("invalid-email"))
```

Test-Driven Development


No
Write a test
Run the test. Does it pass?
Yes
Does it need refactoring?
No
Yes
Write minimal code

The Rules of TDD

1. Write a failing automated test **before you write any code.**
2. Write (and refactor) the code to pass the tests.

TDD isn’t something that comes naturally. It’s a discipline.

1. Add a test

You are adding a new feature – a new test ensures the feature spec is met

You are uncovering the spec – adding new use cases and user stories

You are focusing on requirements before writing code

This is much better than adding tests later …

2. Run All The Tests

The new test(s) should FAIL for known reasons – **there is no code yet**!

Validates the test environment is correctly running and runnable

3. Write the Simplest Code to Pass the New Test(s)

Everything is acceptable at this point (including hard coding) because code will be refined and refactored in Step 5.

4. All Tests Should Now Pass

This solves for two problems

- New code is running to solve for requirements
- No original code has been broken

5. Refactor and Refine

Refactor code for readability and maintainability, **running the test suite after each change to ensure functionality has not broken**

Refactoring includes:

- Move code to where it logically belongs (cohesion and loose coupling)
- Remove duplication
- Clean-up naming
- Clean-up methods and functions into (possibly smaller) logical units

Rinse/Repeat

Tests are small & incremental so code changes should be small & incremental

This means you can easily revert a change in version control or quickly review & edit while you still have the context in your head instead of deep debugging time – or having a test cycle long after you wrote the code

Testing Frameworks & Tools

TDD somewhat requires you to create your tests within the context of the program code

- JUnit
- NUnit
- Selenium
- PyTest testing framework
- Go built in test support ($ go test)
- Rust built in language support ($ cargo test)
- Tox and Tempest for OpenStack
- Mocha, Jest, Jasmine in the Node.js world

- All manner of home grown solutions and proprietary products