TDD with JUnit

Test-Driven Development (TDD)

- TDD Life Cycle
- TDD vs DLP (Debug Later Programming)
- Why TDD is matter
- Unit Testing with F.I.R.S.T
- Code and Test Coverage
- Structure of good unit testing (GUT)

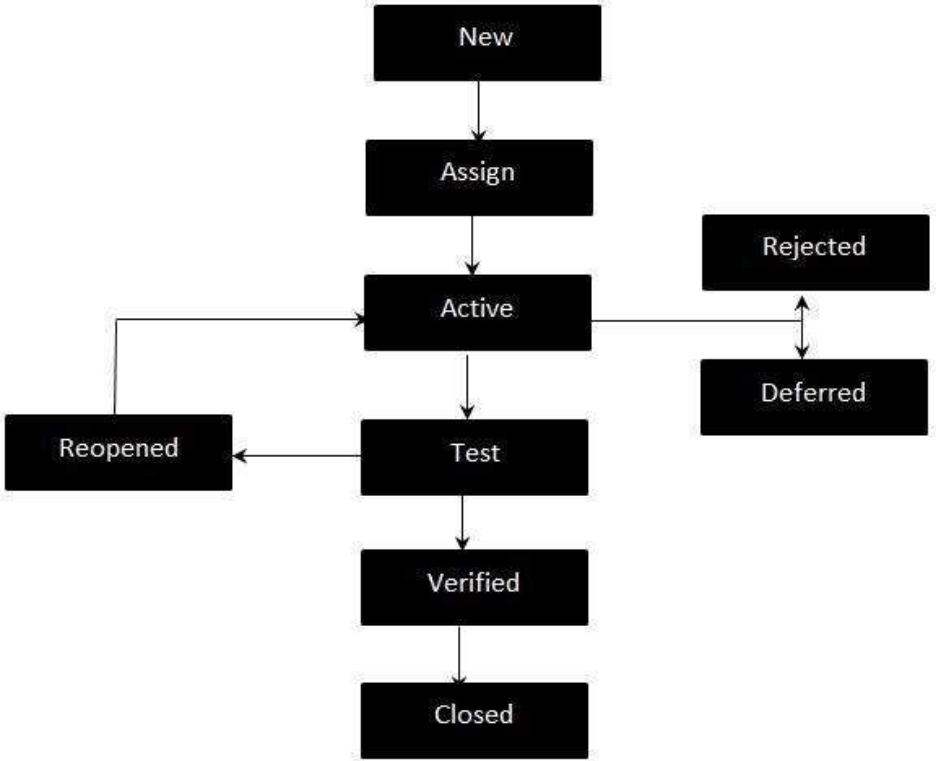
Unit Testing with JUnit

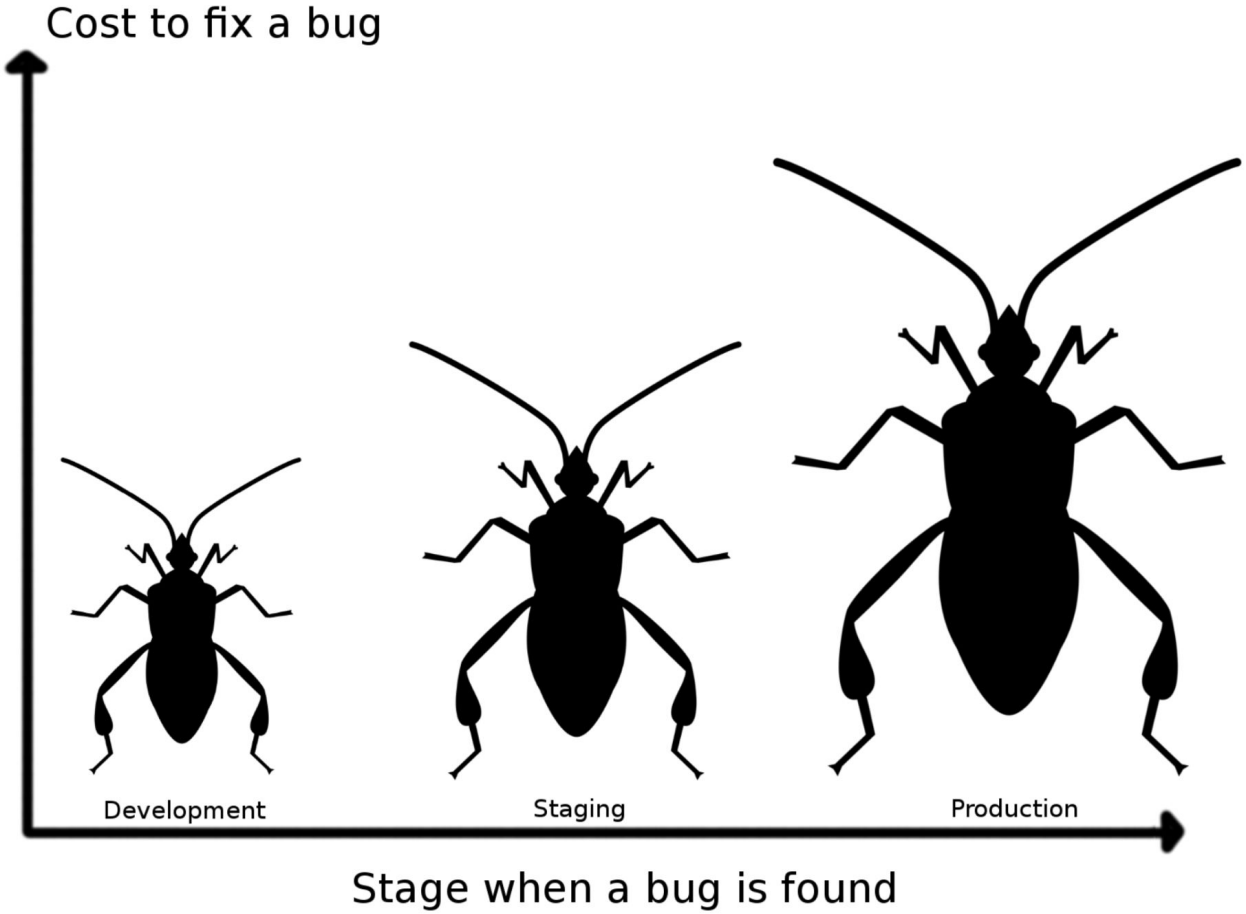
- JUnit lifeCycle
- Assertion
- Data-Driven Test with JUnit
- JUnit features
- Timeout
- Conditional
- Category
- Suite
- Running testing

- Test Double
- Dummy
- Stub
- Spy
- Mock
- Fake

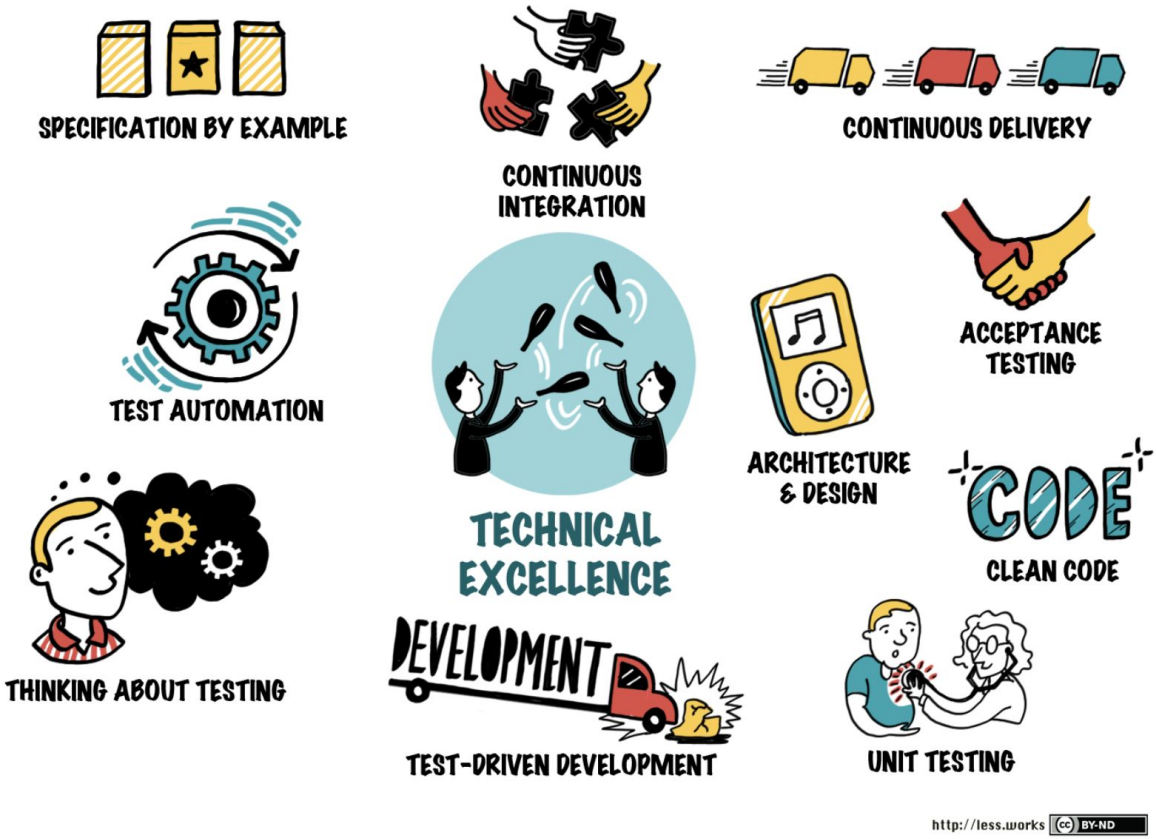
Defect

A **defect is a bug or an error** created in the application. A programmer, while designing and building the software, can make mistakes or errors. These errors mean that there are flaws in the software. These are called defects.

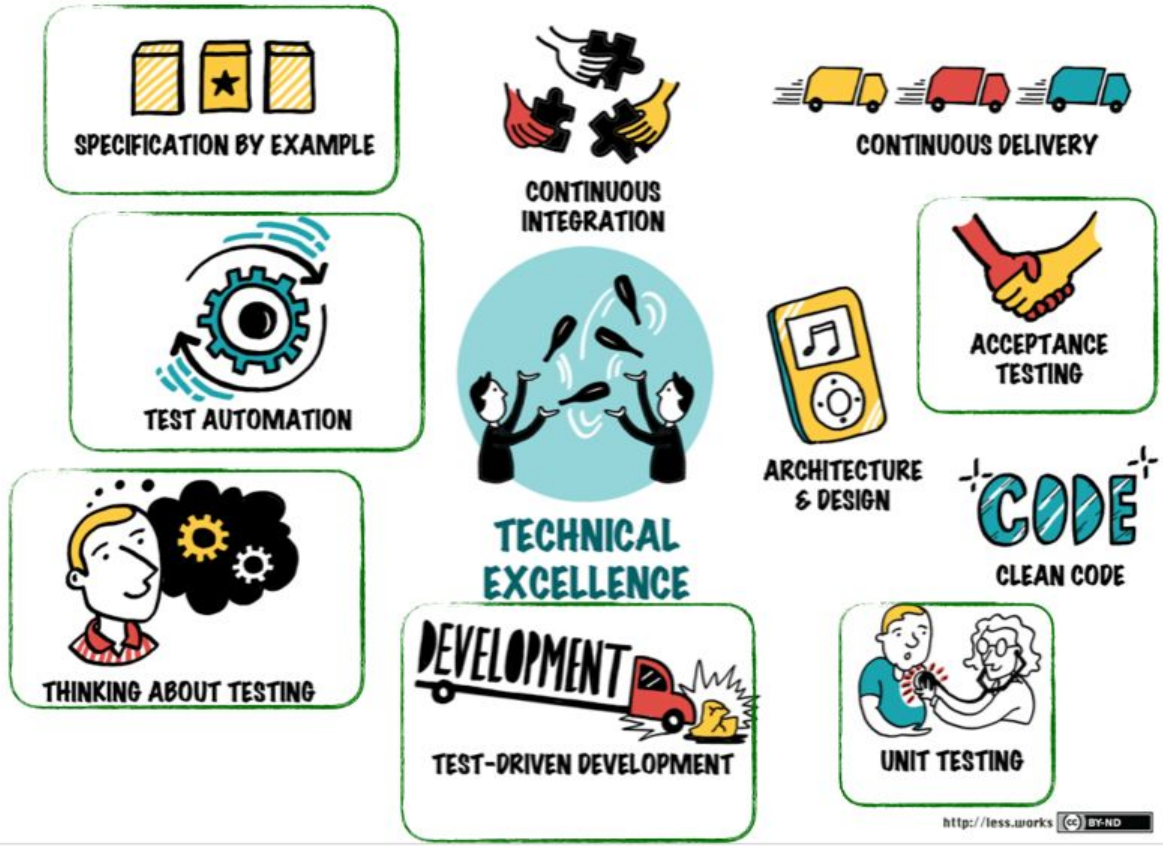



Cost to fix a bug
Development
Staging
Production
Stage when a bug is found

Technical Excellence



Technical Excellence

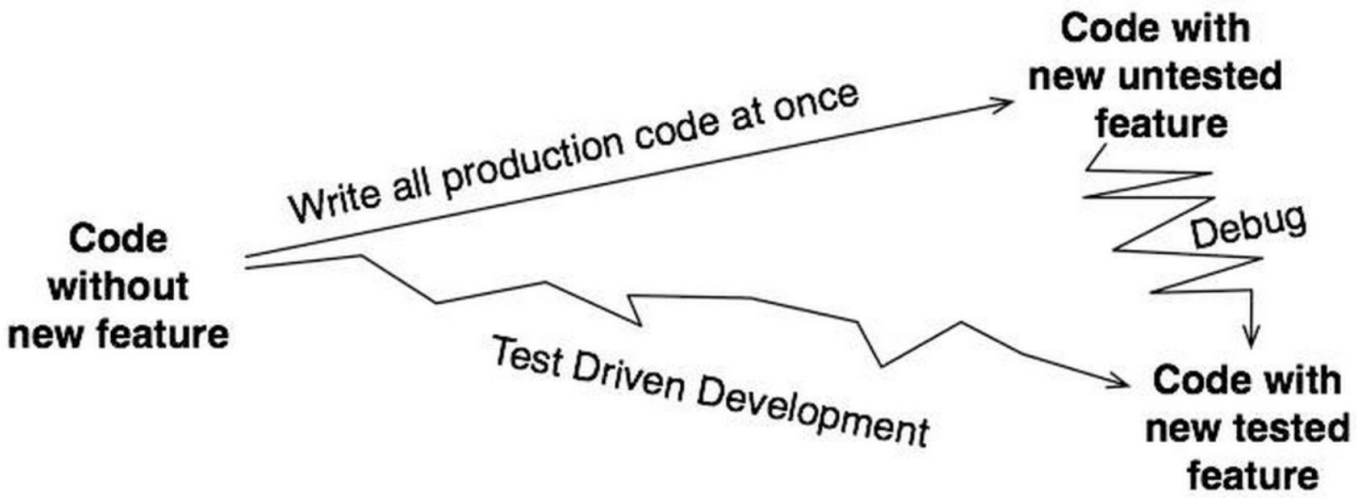


https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/

TDD

Test-Driven-Development
Test-Driven Design

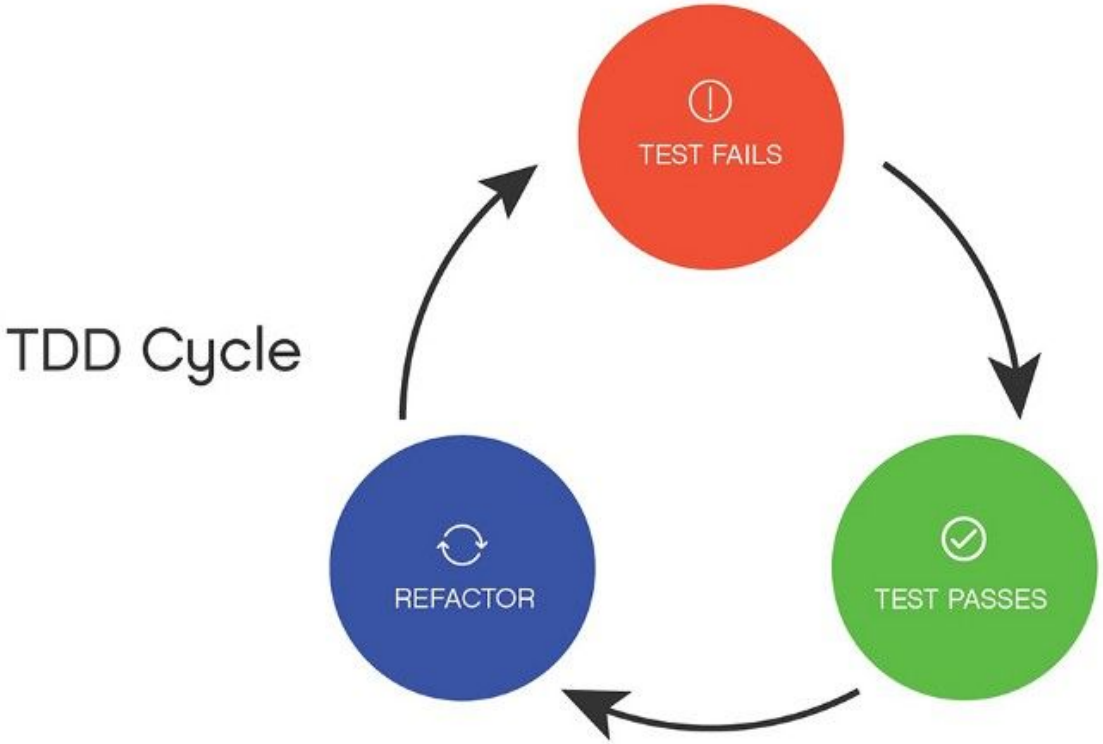
TDD vs DLP (Debug Later)



What Is Test-Driven Development

Test-driven development is a development style that drives the design by tests developed in short cycles of:

1. Write one test.
2. Implement just enough code to make it pass.
3. Refactor the code so it is clean.



Red, Green, Refactor

Red Green Refactor

A TDD cycle Should Be …

Short

The turnaround time for passing each test is short. It could take 5 mins per cycle.

Rhythmic

You'll feel the rhythm distinctly - "red, green, refactor... red, green refactor..."

Incremental

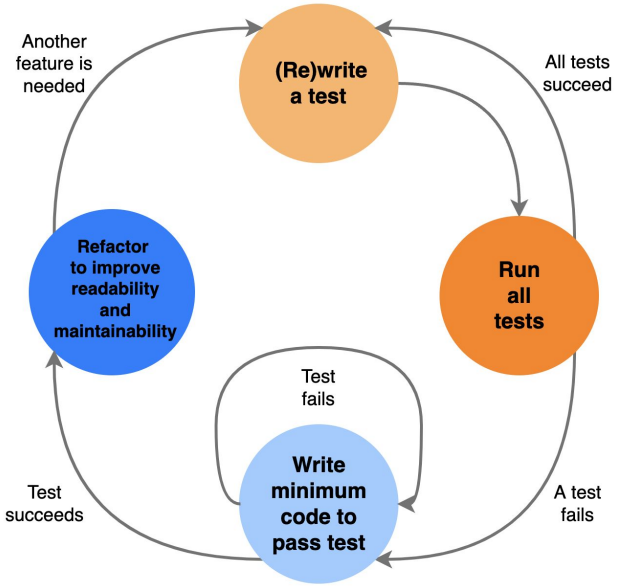
You'll know that as you write and pass more tests, working functionalities are being build up incrementally.

Design-focused

With good knowledge of software design principles, you'll discover TDD is not a testing technique but a method of designing software.

Disciplined

TDD is a different way of developing software. To break the old habit of "code and fix" and to adopt a new habit will require discipline and persistence.



The cycle of test-driven development

Code Smell

Code Smells

– What? How can code "smell"??

– Well it doesn't have a nose... but it definitely can stink!

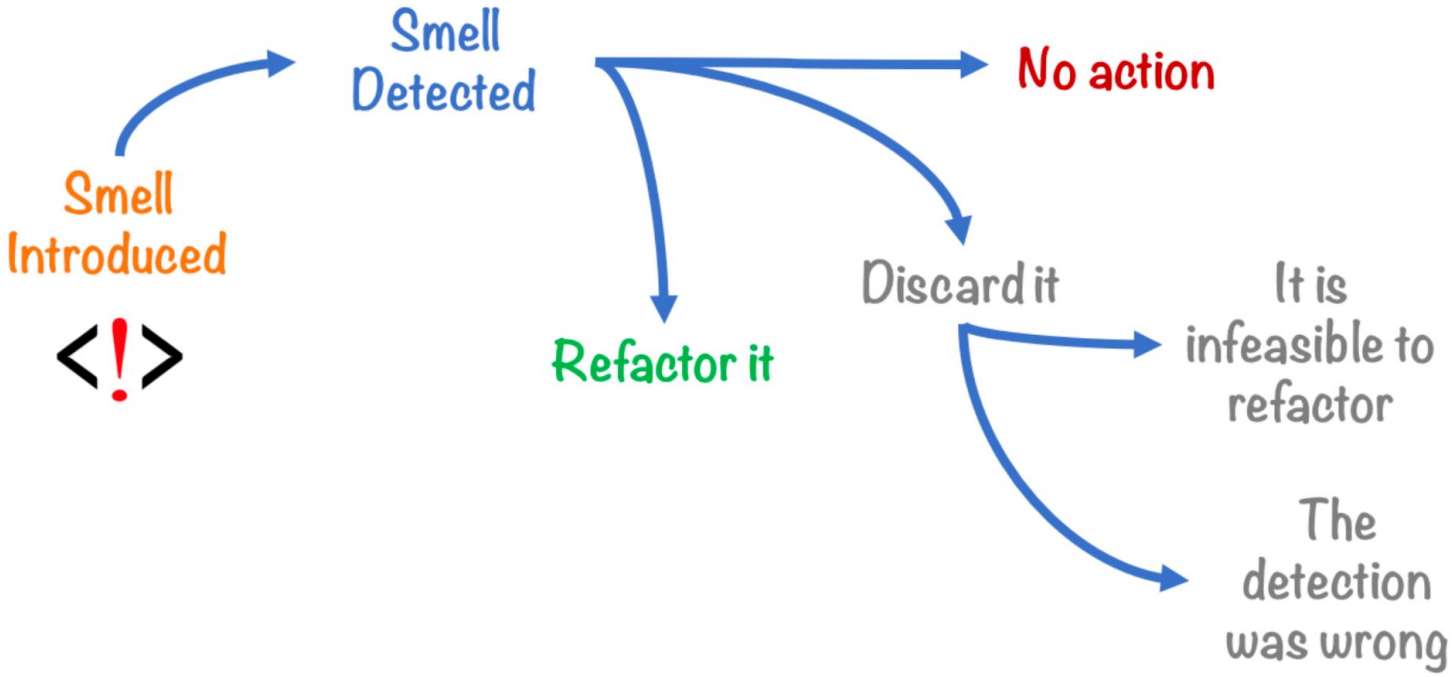
Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

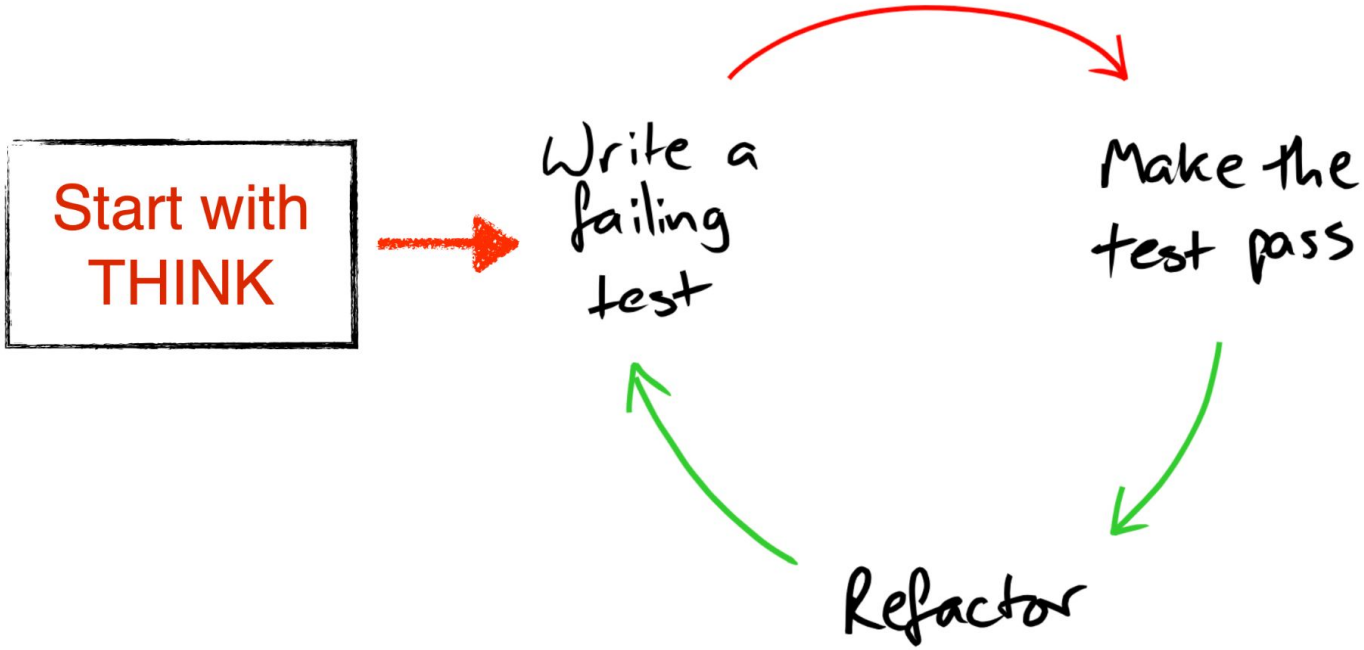
- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

https://sourcemaking.com/refactoring/smells

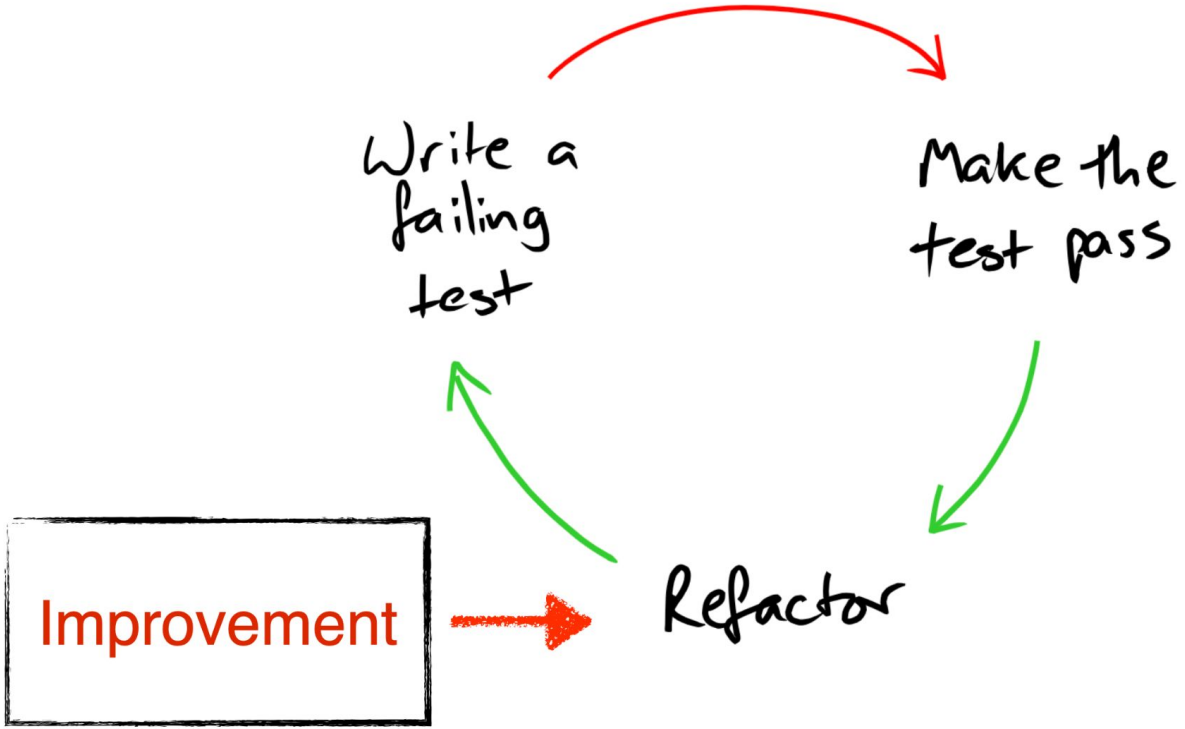
Code Smell


Smell Introduced
<!>
Smell Detected
No action
Refactor it
Discard it
It is infeasible to refactor
The detection was wrong

Improve TDD Cycle



Improve TDD Cycle



The Golden rule of TDD

Discipline #1

You are not allowed to write production code until you have first written a failing unit test.

Discipline #2

You are not allowed to write more of a unit test than is sufficient to fail, and not compiling is failing.

Discipline #3

You are not allowed to write more production code than is sufficient to pass the currently failing test.

The Golden rule of TDD

"Never write a line of code without a failing test"

- Kent Beck -

Make the test pass !!

Fake or hard code
Triangulation
Professional code

Good Unit Tests (F.I.R.S.T)

Fast
Independent/Isolate
Repeat
Self-validate
Thorough/Timely

Good Unit Test Patterns

A good pattern to follow in a unit test is “**AAA**”: **Arrange**, **Act** and **Assert**.

The AAA protocol is a recommended approach for structuring unit tests. As a unit testing best practice, it improves your test’s readability by giving it a logical flow. AAA is sometimes referred to as the “Given/When/Then” protocol.

You can use the AAA protocol to structure your unit tests with the following steps:

- **Arrange**: Arrange the setup and initialization for the test
- **Act**: Act on the unit for the given test
- **Assert**: Assert or verify the outcome

Arrange, Act, and Assert (AAA)

```
def test_abs_for_a_negative_number():

  # Arrange
  negative = -2

  # Act
  answer = abs(negative)

  # Assert
  assert answer == 2
```

What Is Unit Test?

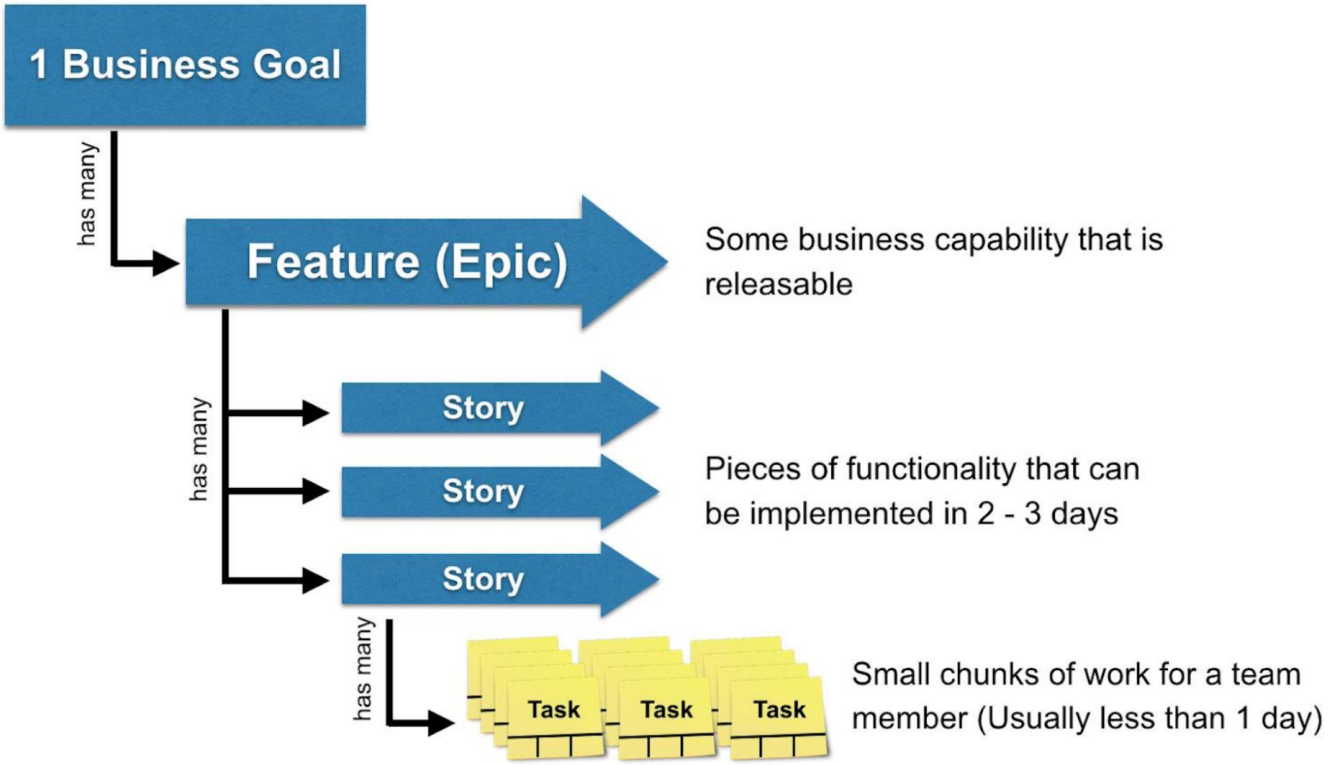
Unit Tests are software programs written to exercise other software programs (called **Code Under Test**, or **Production Code**) with **specific** preconditions and verify the **expected behaviour**s of the CUT.

Unit tests are usually written in the same programming language as their code under test.

Each **unit test** should be small and test only limited piece of code functionality. Test cases are often grouped into **Test Groups** or **Test Suites**. There are many open source **unit test framework**s. The popular ones usually follow an **xUnit** pattern invented by Kent Beck, for example, JUnit for Java and CppUTest for C/C++.

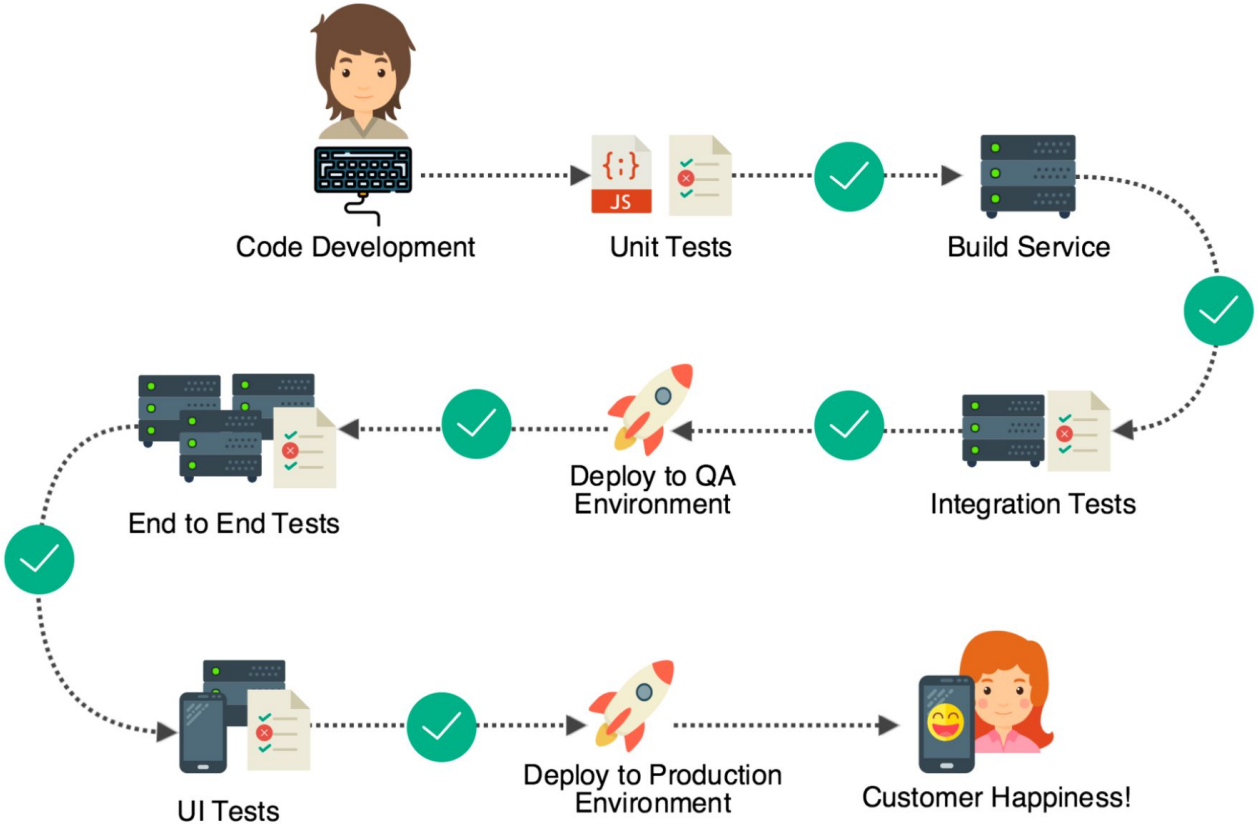
Unit tests should also run very fast. Usually, we expect to **run hundreds of unit test cases within a few seconds**.

Start with Business Goal to Tasks



Source: The Whole Team Approach to Agile Testing by Janet Gregory and Lisa Crispin

Delivery process



TDD in Iteration development

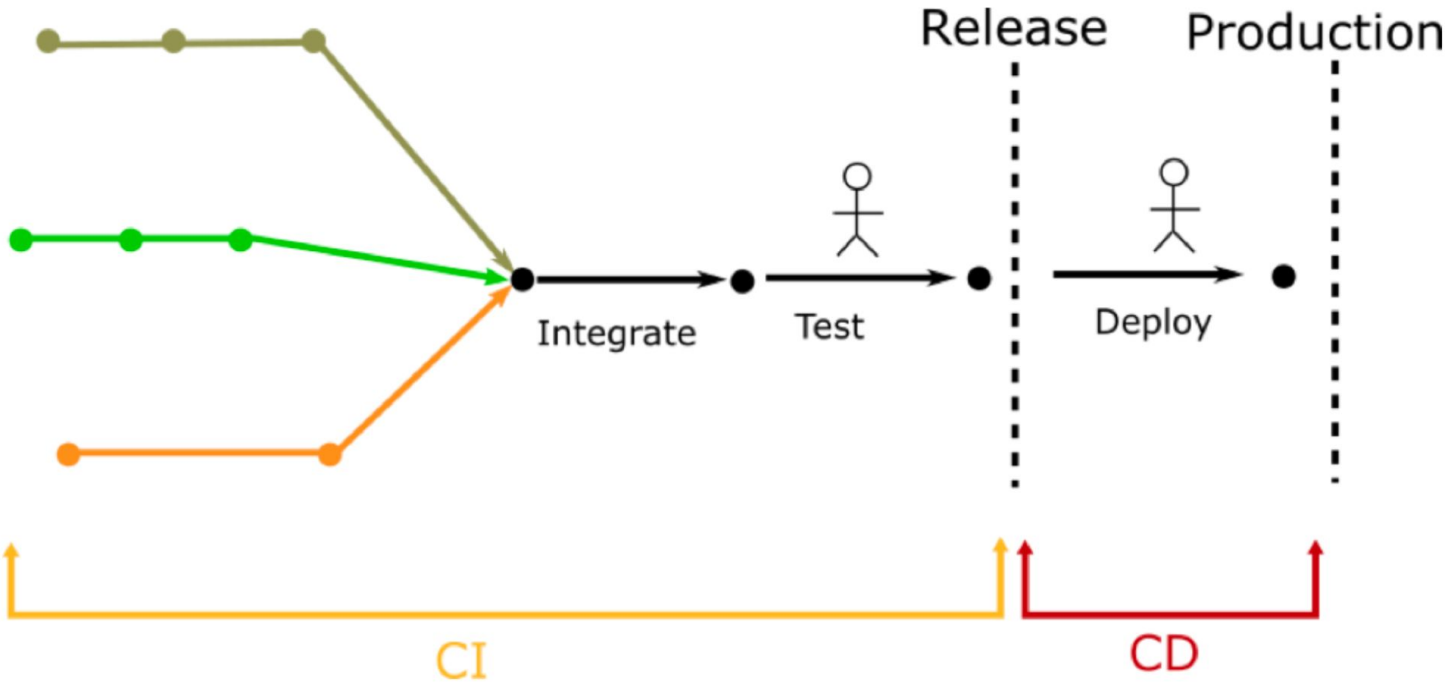
TDD in Iteration development

Iteration cycle 2-4 weeks

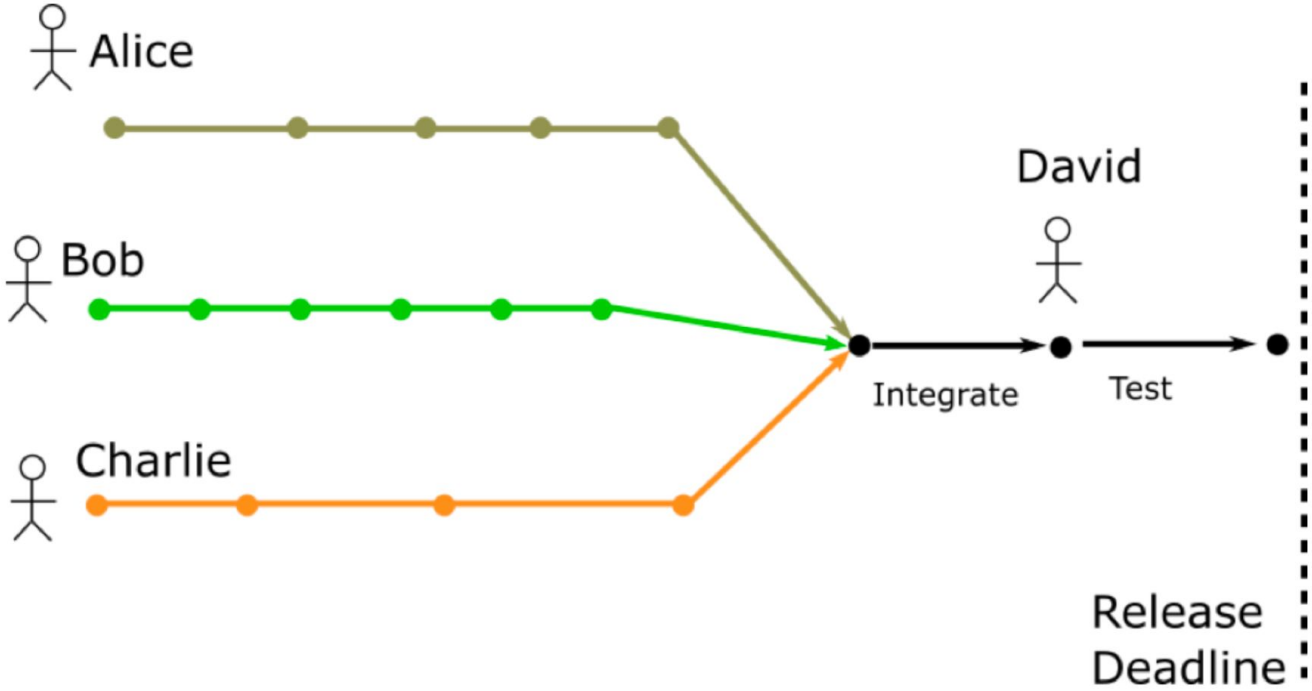
Continuous integration cycle
(multiple times a day)

TDD cycle in a few minutes (3-10 minutes)

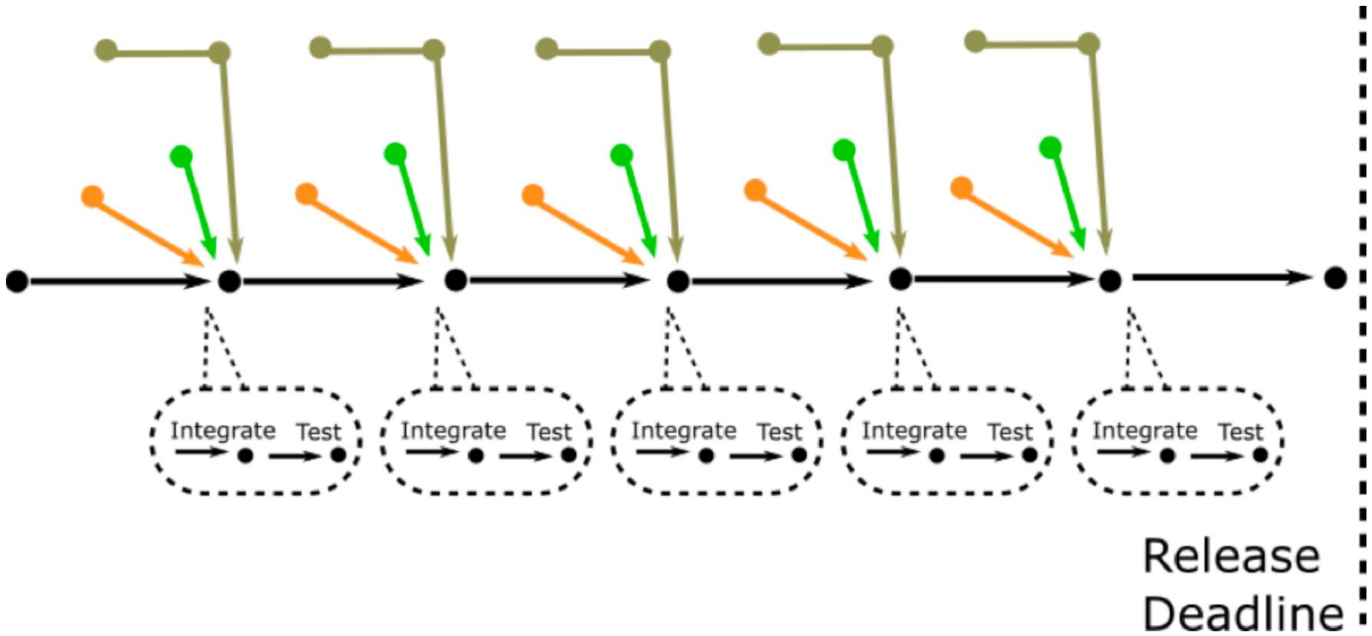
TDD in Iteration development



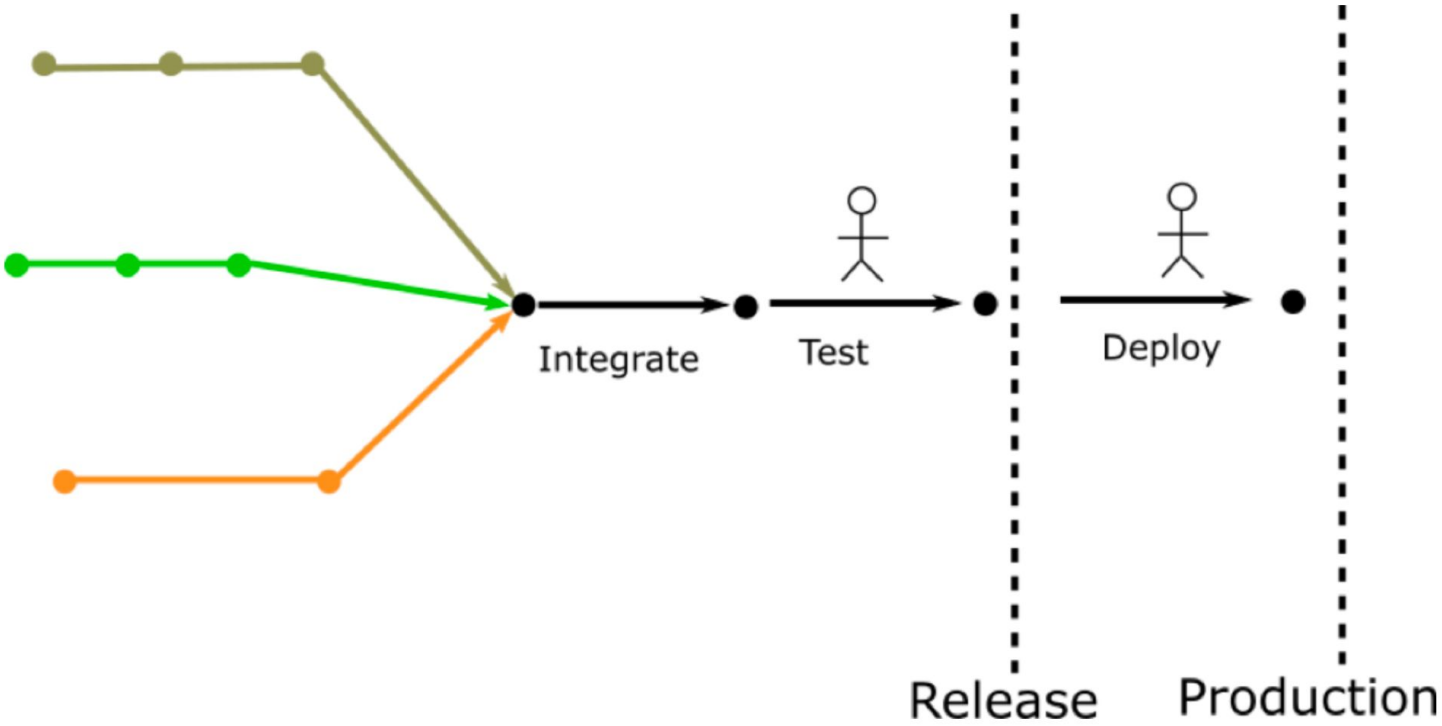
TDD in Iteration development



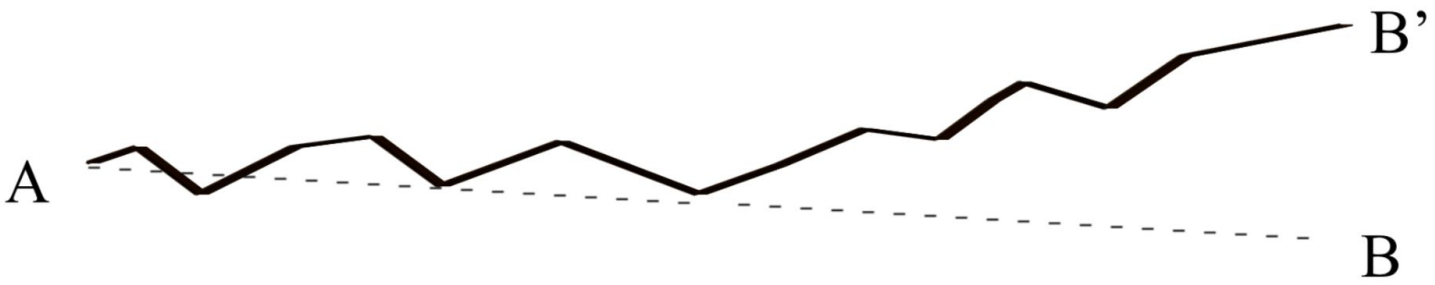
TDD in Iteration development



TDD in Iteration development



Small step



Code coverage

Code coverage

A tool to measure how much of your code is covered by tests that break down into classes, methods and lines.

Code coverage

```
1.  public class MyRange {
2.      public boolean startWithInclude(String input) {
3.          return input.startsWith("[");
4.      }
5.
6.      public boolean endWithInclude(String input) {
7.          if(input == null) {
8.              return false;
9.          }
10.         return input.endsWith("]");
11.     }
12.
13.     public boolean startWithInclude2(String input) {
14.         return input.startsWith("[");
15.     }
16. }
```

Code coverage

Class coverage
Method coverage
Line coverage
Branch coverage

Code coverage

But 100% of code coverage **does not mean** that your code is 100% correct

"Code coverage can show the high risk areas in a program, but never the risk-free."

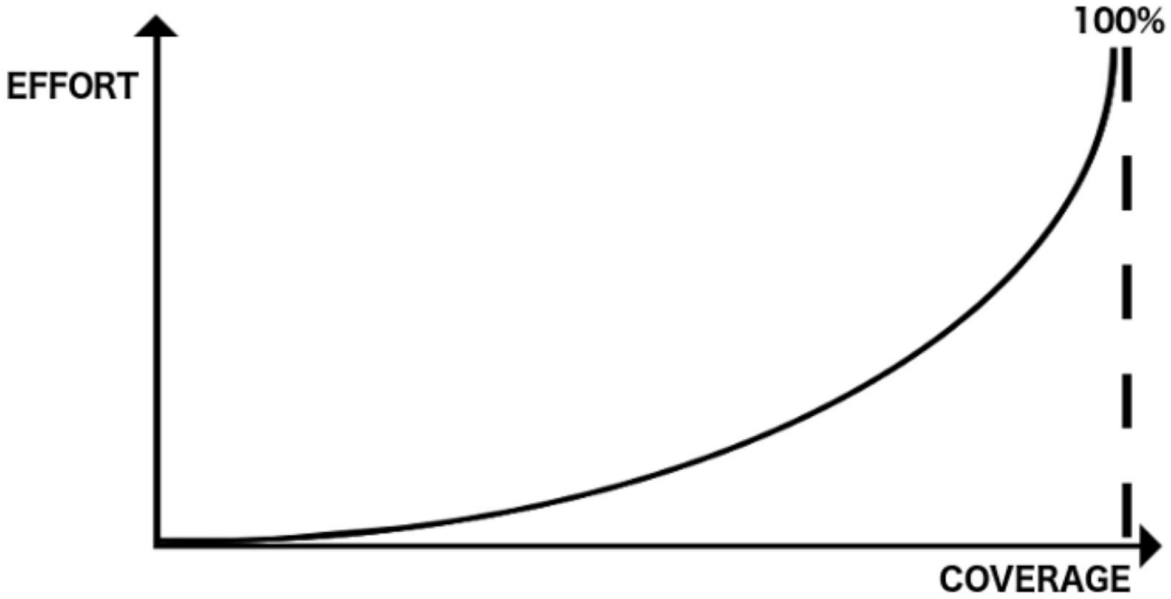
Paul Reilly, 2018, Kotlin TDD with Code Coverage

Code coverage

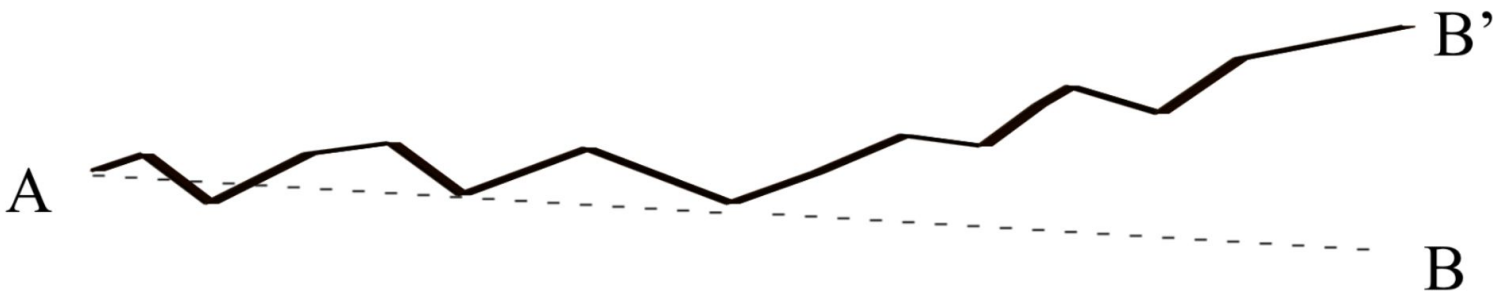
Powerful tool to improve the quality of your code

Code coverage != quality of tests

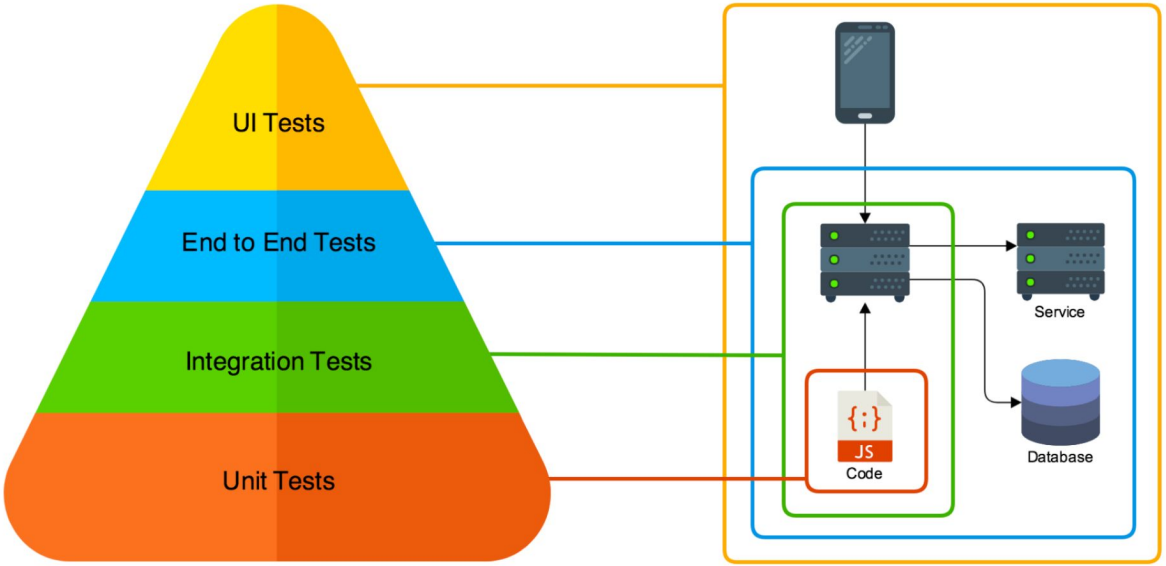
Code coverage 100% ?



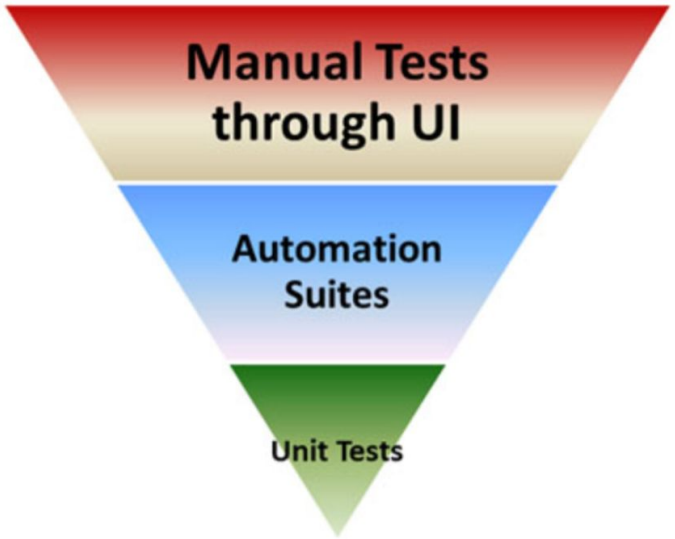
Small steps



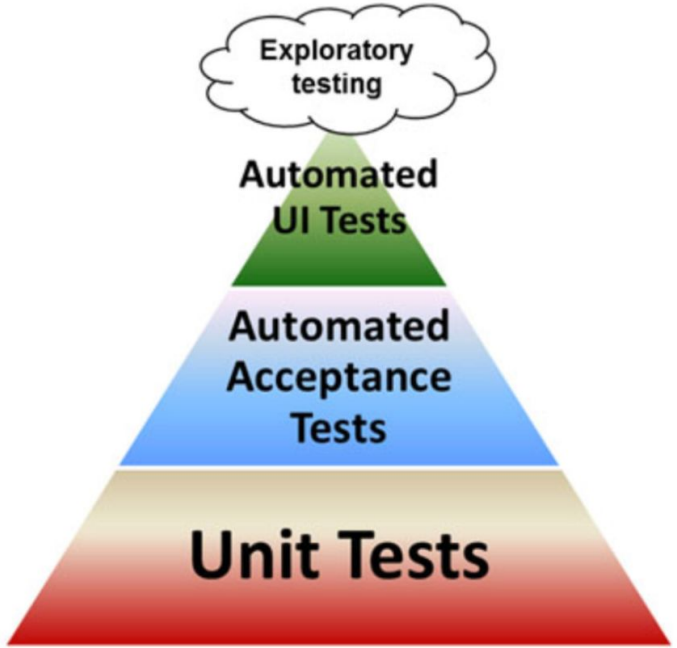
Types of Test


UI Tests
End to End Tests
Integration Tests
Unit Tests
Service
Database
JS
Code

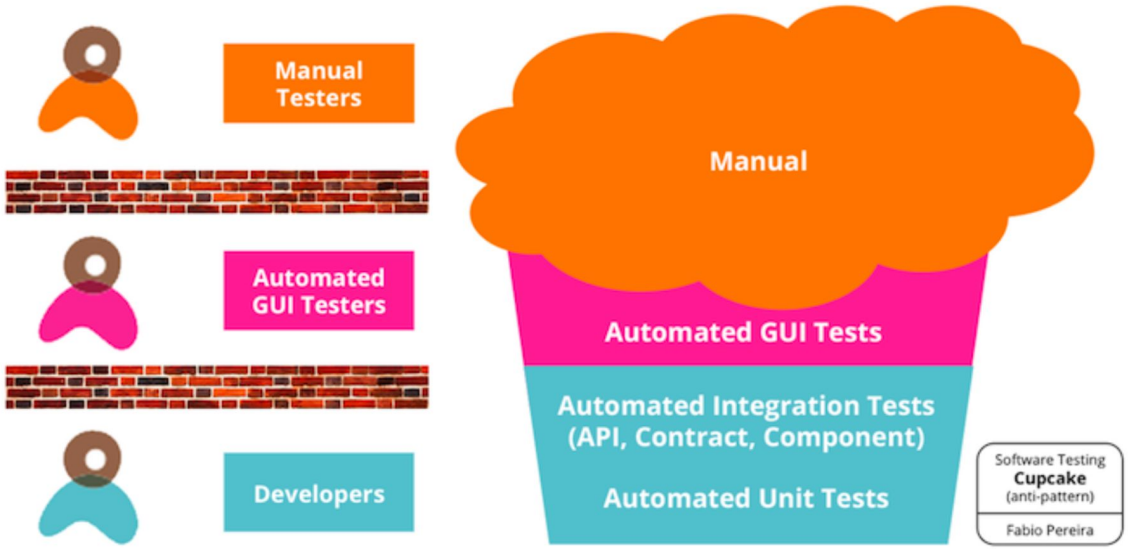

D D T D
T D T T
D T D D
Testing is an ACTIVITY!
I'm part of the board!
T
Todo
BUSY
SHOW ME
DONE



Traditional
(find bugs)



Agile
(prevent bugs)



https://www.thoughtworks.com/insights/blog/introducing-software-testing-cupcake-anti-pattern

Workshop

Workshop#1

Setup Your Workspace

Workshop#2

Your first TDD

Workshop#3

FizzBuzz

Input	Expected Result
1	1
2	2
3	Fizz
4	4
5	Buzz
6	Fizz
7	7
8	8
9	Fizz
10	Buzz
15	FizzBuzz