University of Zanjan

Testing In Software Engineering

Testing Software - TDD

Kian Sahafi - Spring 2023

"Professional developers test their code. But testing is not simply a matter of writing a few unit tests or a few acceptance tests. Writing these tests is a good thing, but it is far from sufficient. What every professional development team needs is a good *testing strategy*. "

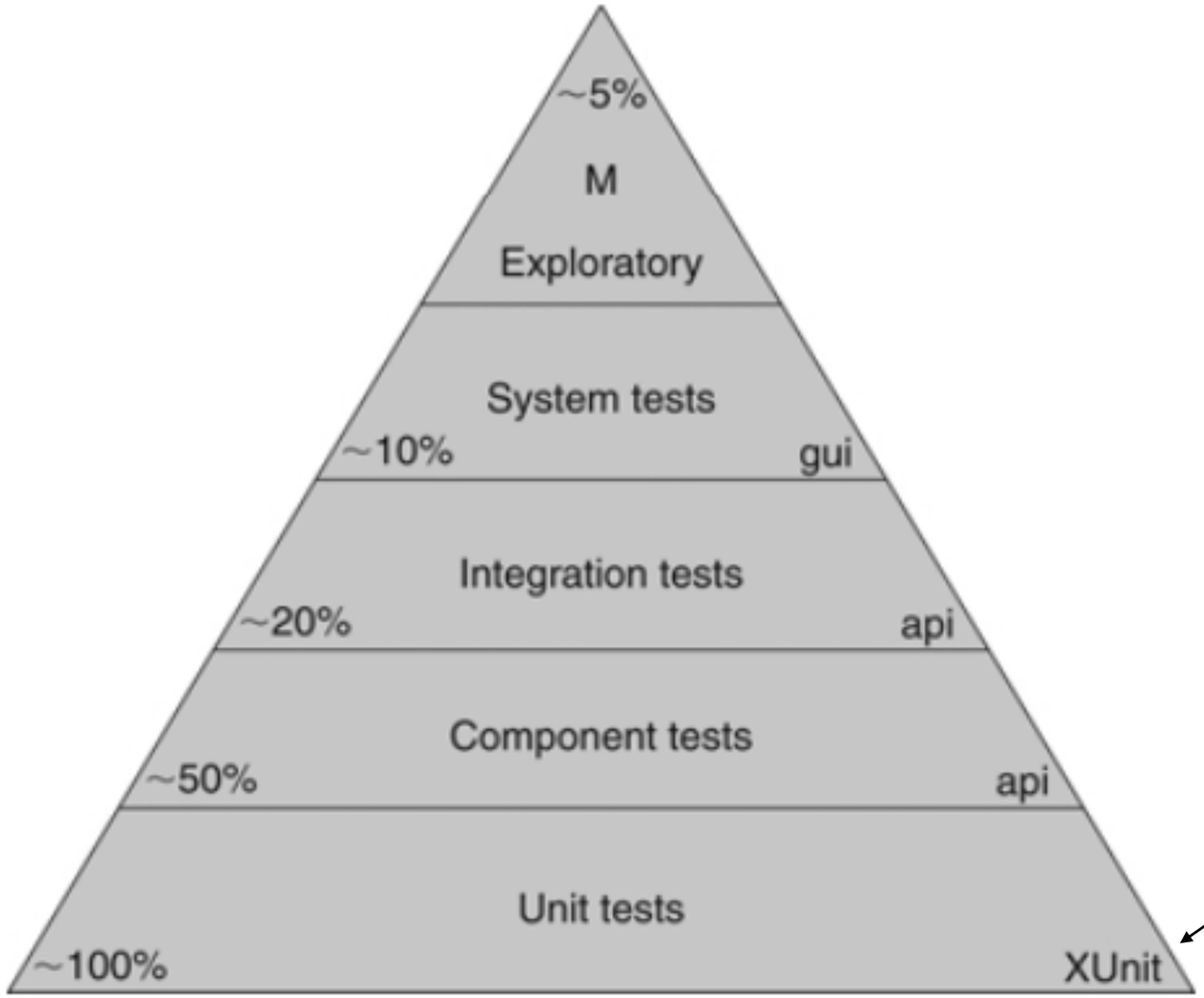
-Robert C. Martin

Testing Strategies

The Test Automation Pyramid

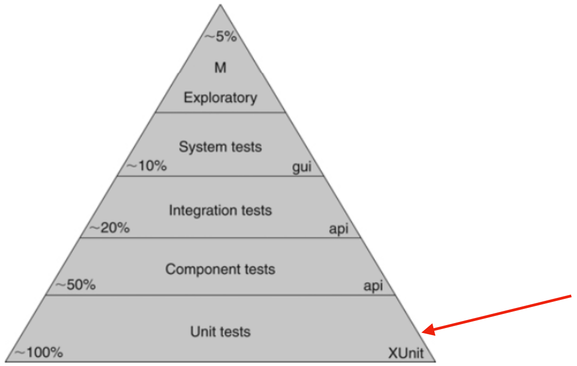
Professional developers employ the discipline of Test Driven Development to create unit tests. Professional development teams use acceptance tests to specify their system, and continuous integration to prevent regression.

The Test Automation Pyramid



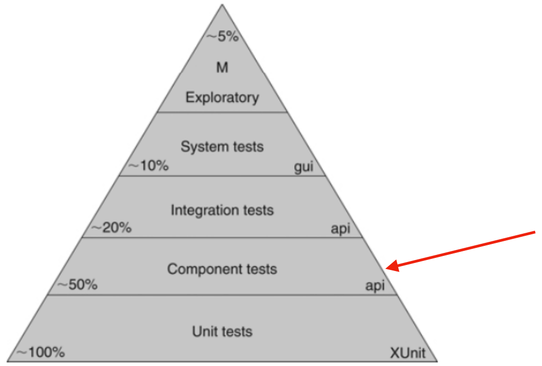
Or Junit (For Java)

This Figure shows the Test Automation Pyramid,[2] a graphical depiction of the kinds of tests that a professional development organization needs.



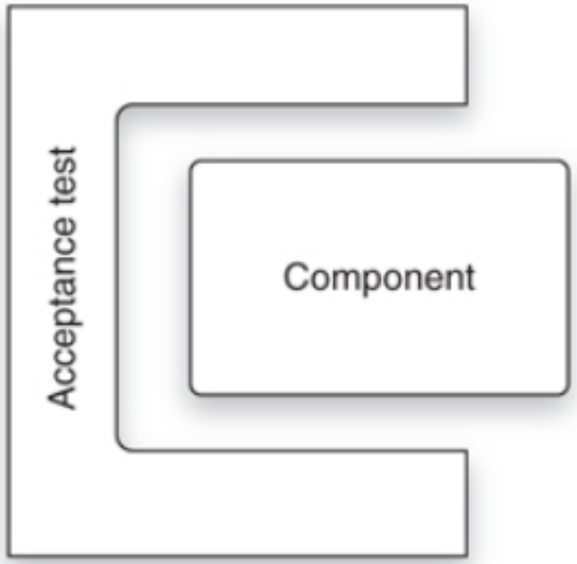
Unit Test

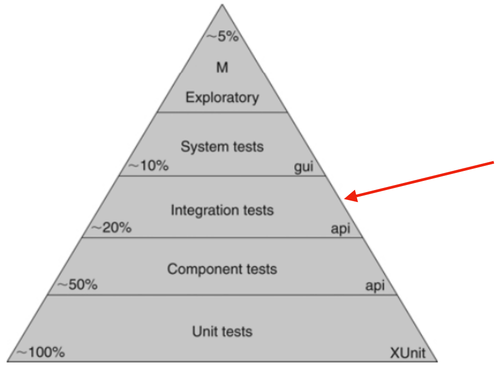
- At the bottom of the pyramid are the unit tests. These tests are written by programmers, for programmers, in the programming language of the system. The intent of these tests is to ***specify the system at the lowest level***. Developers write these tests **before** writing production code as a way to specify what they are about to write. They are executed as part of Continuous Integration to ensure that the intent of the programmers' is upheld.
- Unit tests provide as close to 100% coverage as is practical. Generally this number should be somewhere in the 90s. And it should be *true* coverage as opposed to false tests that execute code without asserting its behavior.



Component Tests

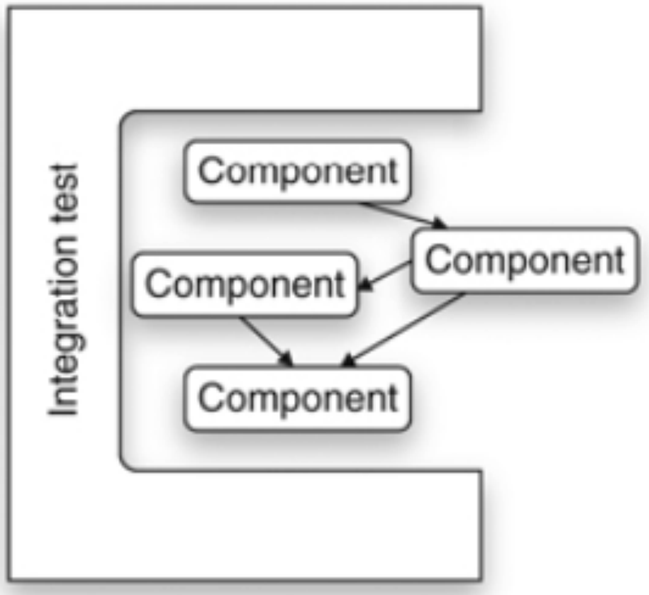
- Generally they are written against individual components of the system. The components of the system encapsulate the business rules, so the tests for those components are the acceptance tests for those business rules.
- As depicted in this Figure a component test wraps a component. It passes input data into the component and gathers output data from it. It tests that the output matches the input. Any other system components are decoupled from the test using appropriate mocking and test-doubling techniques.
- Component tests are written by QA and Business with assistance from development. They are composed in a component-testing environment such as FITNESSE, JBehave, or Cucumber. (GUI components are tested with GUI testing environments such as Selenium or Watir.) The intent is that the *business should be able to read and interpret these tests*, if not author them.
- Component tests cover roughly half the system. They are directed more towards happy-path situations and very obvious corner, boundary, and alternate-path cases. The vast majority of unhappy-path cases are covered by *unit tests* and are meaningless at the level of component tests.

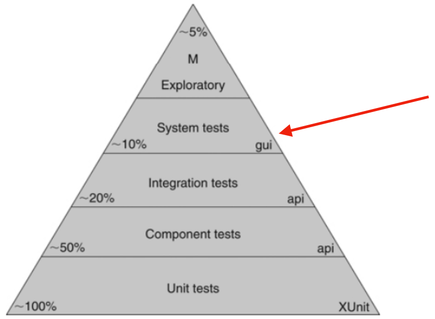




Integration Tests

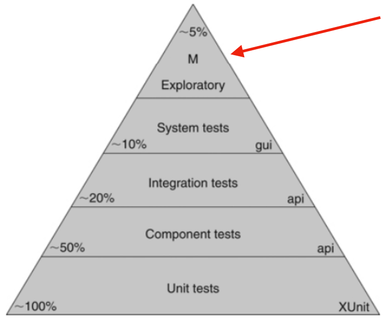
- These tests only have meaning for larger systems that have many components. As shown in the Figure, these tests assemble groups of components and test how well they communicate with each other. The other components of the system are decoupled as usual with appropriate mocks and test-doubles.
- Integration tests are *choreography* tests. They do not test ***business rules***. Rather, they test how well the assembly of components dances together. They are *plumbing* tests that make sure that the components are properly connected and can clearly communicate with each other.
- Integration tests are typically written by the system architects, or lead designers, of the system. The tests ensure that the architectural structure of the system is sound. It is at this level that we might see performance and throughput tests.
- Integration tests are typically written in the same language and environment as component tests. They are typically *not* executed as part of the Continuous Integration suite, because they often have longer runtimes. Instead, these tests are run periodically (nightly, weekly, etc.) as deemed necessary by their authors.





System Tests

- These are automated tests that execute against the entire integrated system. They are the ultimate integration tests. They do not test business rules directly. Rather, they test that the system has been wired together correctly and its parts interoperate according to plan. We would expect to see throughput and performance tests in this suite.
- These tests are written by the system architects and technical leads. Typically they are written in the same language and environment as integration tests for the UI. They are executed relatively infrequently depending on their duration, but the more frequently the better.
- System tests cover perhaps 10% of the system. This is because their intent is not to ensure correct system behavior, but correct system *construction*. The correct behavior of the underlying code and components have already been ascertained by the lower layers of the pyramid.

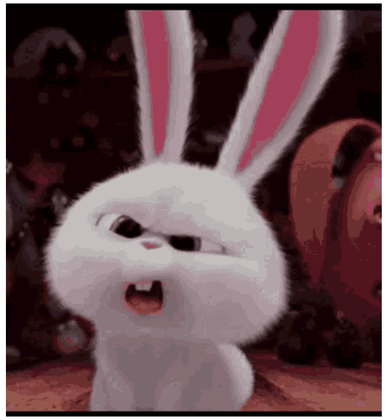


Manual Exploratory Tests

- This is where humans put their hands on the keyboards and their eyes on the screens. These tests are not automated, *nor are they scripted*. The intent of these tests is to explore the system for unexpected behaviors while confirming expected behaviors. Toward that end we need human brains, with human creativity, working to investigate and explore the system. Creating a written test plan for this kind of testing defeats the purpose.
- Some teams will have specialists do this work. Other teams will simply declare a day or two of "bug hunting" in which as many people as possible, including managers, secretaries, programmers, testers, and tech writers, "bang" on the system to see if they can make it break.
- The goal is not coverage. We are not going to prove out every business rule and every execution pathway with these tests. Rather, the goal is to ensure that the system behaves well under human operation and to creatively find as many "peculiarities" as possible.

What the Heck is TDD?

And why am I hearing this much about it?



- It has been over ten years since Test Driven Development (TDD) made its debut in the industry. It came in as part of the Extreme Programming (XP) wave, but has since been adopted by Scrum, and virtually all of the other Agile methods. Even non-Agile teams practice TDD.

The Three Laws of TDD

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

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

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

The Litany Of Benefits

Is TDD any good?

How certain? Certain enough to ship!

FITNESSE has 64,000 lines of code,
With 2,200 individual unit tests
With 90% coverage!

Certainty

teams have experienced defect reductions
of 2X, 5X, and even 10X

mission-critical application?

Defect Injection Rate

what if you could be sure that your cleaning
did not break anything?

Why don't you fix bad code when you see it?

When programmers lose the fear of cleaning, they clean!

Courage

Each of the unit tests you write is an example,
written in code, describing how the system should be used

third-party framework

Documentation

The problem with testing code
is that you have to isolate that code

Design

What if I write my tests later?

forces you to think about good design

But the tests you write after the fact are defense.
The tests you write first are offense.

What TDD is Not!

- For all its good points, TDD is not a religion or a magic formula. Following the three laws does not guarantee any of these benefits. You can still write bad code even if you write your tests first. Indeed, you can write bad tests.
- By the same token, there are times when following the three laws is simply impractical or inappropriate. These situations are rare, but they exist.

Didn’t Get any of it?
No problem. ;-)

Let’s code...

Reference: Clean Coder - Robert C. Martin