Test Driven Development (TDD)

TDD

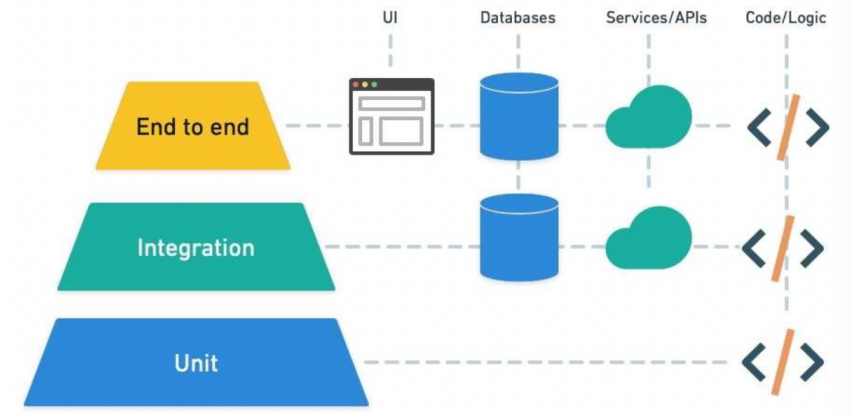
- TDD sits nicely in the XP way of doing things
  - TDD can be used without practicing XP
- Giving lectures is one such way of achieving that goal
  - Practising them is better
- Reduce the amount of re-testing that is required
  - Especially when dealing with legacy applications
  - Avoid introducing new bugs after refactoring existing code
- TDD had "broader goals" that just insuring quality
  - Improve developers lives (coping, confidence)
  - Support design flexibility and change
  - Allow iterative development with working code early

What is TDD?

- Before you write code, think about what it will do. Write a test that will use the methods you haven't even written yet
- A test is not something you "do" it is something you "write" and run once, twice, three times, etc
  - It is a piece of code
    - Testing is therefore automated
      - lo hace solo
      - le das a un botón y lo hace
      - Se puede ejecutar varias veces
  - Repeatedly executed, even after small changes
  - As much about design as about testing
    - Encourages design from a user's point of view
    - Encourages testing classes in isolation
    - Produces loosely-coupled, highly-cohesive systems
      - highly cohesive
        - module understandable as a meaningful unit (clarity)
        - Components of a module are closely related to one another
      - modules should exhibit low coupling
        - modules have low interactions with others
        - Understandable separately
- Que tiene que hacer ese cacho de código

- TDD is the process of thinking about a block of code from the perspective of the user of that code
- TDD is a technique whereby you write your test cases before you write any implementation code
- An iterative technique for developing software
    - Tests drive or dictate the code that is developed
        - Write a test that show what you want the code to do
        - Letting you codebase grow organically as you create examples of what the code should do
    - Software is written from the outside in: Test provide a specification of "what" a piece of code actually does
        - Some say that "tests are part of the documentation"
        - Other say that "tests are part of the design"

The test pyramid



Programmers dislike testing

- They will test reasonably thoroughly the first time
- The second time however, testing is usually less thorough
- The third time, well

Testing is considered a boring task

- Testing might be the job of another department / person

TDD encourages programmers to maintain an exhaustive set of repeatable tests

- With tool support, test can be run selectively
- The tests can be run after every single change
- Must be learned and practiced: if it feels natural at first, you're probably doing it wrong

More productive than debug-later programming

- Developers work in a predictable way of developing code
- It's an addiction rather than a discipline

The act of writing a unit test is more an act of design than of verification

- DEVELOPER = PROGRAMMER

We are talking about unit testing

- Testing the internals of a class
  - There is some debate about what constitutes a unit
  - Here are some common definition of a unit
    - The smallest chunk that can be compiled by itself
    - A stand - alone procedure of a function
    - Something so small that it would be developed by a singles person
  - Black box testing for objects
    - Classes are testing in isolation
    - Remember: the goal is to produce loosely coupled, highly cohesive architectures

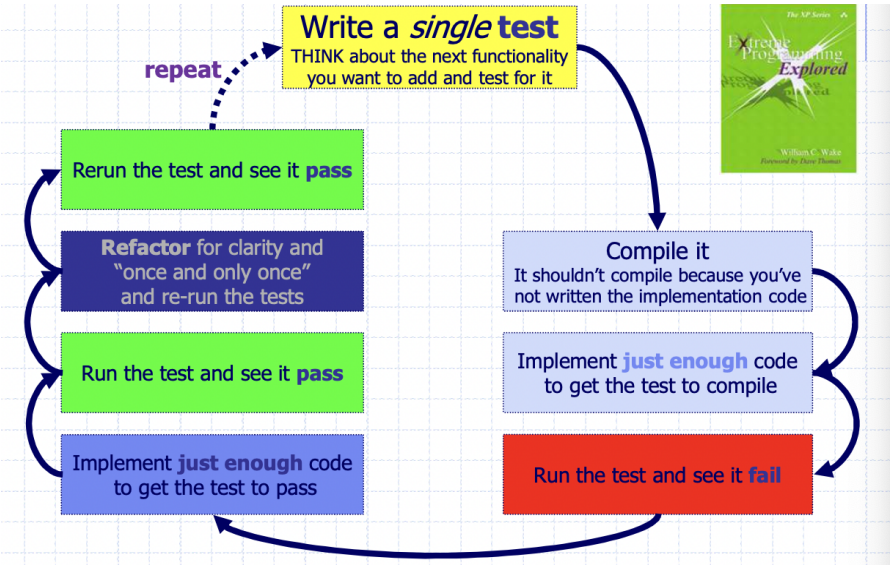
Who should write the tests?

- The programmers should write the test
  - They can't wait for somebody else to write tests
- TDD promotes "small steps" and lots of them
  - Small steps
    - the real shortest distances between two points
  - Use TDD to get from A to B in very small. verifiable steps
    - You often en up in a better place
- BDUF
  - Big design Up-front
    - Software development method where a "big" design is created before coding and testing takes place
- Spike
  - A quick (minutes,hours) exploration by coding of an area in which the development team lacks confidence

Unit testing framework

- Framework
  - Andamio
  - forma de trabajar
- Test must be automated
  - if not, they won't be run
- xUnit is a colloquial umbrella term
  - Most languages have a testing framework
    - JUnit, PyUnit
      - Simple tool (platform-specific implementations)
      - Collects, organizes and automatically calls your test code
- Graphical test runner
  - Green bar makes you feel good

TDD Stages: red/green/refactor



A Test Life Cycle

- Write a test (red)
  - The test fails
  - The test must not work; must now even compile at first
  - Think about how you would like the feature to appear in code
    - Invent the API you wish you had
  - Include all the elements in the story that you imagine will be necessary to calculate the right answers
- Make it run (green)
  - Make the test work quickly, doing whatever sins be necessary
  - Quickly getting the test to pass dominates everything else

        - If a clean simple solution is obvious, type it in
        - If the clean, simple solution is obvious but will take a minute, make a note of it and get back to the main problem - making the test pass
- Make it right (refactor)
    - Change the design
    - Now that the system is ok, put the sins of the recent pass out
        - Step back onto the straight and narrow path of software righteousness
    - Eliminate the duplication created in just getting the test to work

TDD Process

- Features and benefits
    - Once a test passes, it is re-run with every change
    - Broken test are not tolerated
    - Side-effects defects are detected immediately
    - Assumptions are continually checked
    - How should running of tests affect one another
        - they shouldn't be affected they are isolated test
    - Automated test provide a safety net that gives you the courage to the refactor
    - What do you test? everything that could possibly break
    - Don't test anything that could not possibly break
        - Always a judgment call
        - Example
            - Simple accessors and mutators

TDD Process

- Start small or not at all
    - Select one small piece of functionality that you know is needed and you understand
- Ask "what set of test, when passed, will demonstrate the presence of code we are confident fulfills the functionality correctly"
- Make a to-do list, keep it next to the computer
    - List test that need to be written
    - Reminds you of what need to be done
    - keeps you focused
    - When you finish an item, cross it off
    - When you think of another test to write, add it to the list

Automated Unit Tests: Right

- Unit tests show the programmer that the code does what is expected to do
    - Specifies what the code must do (Specification by example)
    - Provides examples of how to use the code (documentation)
    - All test are run every few minutes, with every change
- Right: Are the result right?
    - Validate results
        - use the acronym BICEP
            - B-Boundary
                - Garbage input values
                - Badly formatted data
                - Empty or missing values(0,null,etc)
                - Values out of reasonable range
                - Duplicates (if they are not allowed)
                - Unexpected ordering
            - I- Inverse relationships
                - Check inverse relationships
                - if your method does something that has an inverse the apply the inverse
                    - Square and square - root
                    - Insertion then deletion
                - Beware errors that are common to both your operations
                    - Seek alternative means of applying inverse if possible
            - C - Cross-check using other means
                - Can you do something more than one way?
                    - your way, and then the other way, match?
                - Are there overall consistency factors you can check?
                    - Overall agreement
            - E - Force Error conditions
                - Some are easy, invalid parameters, out of range values
                - Others not so easy, exceptions
                - Failures outside your code
                    - Out of memory, disk full, network down
                    - Can simulate such failures
                        - Example
                            - Use mock objects
            - P - Performance

            - Perhaps absolute performance, or
            - Perhaps how performance changes as input grows
            - Perhaps separate test suite in JUnit
        - Does the expected result match what the method does
    - If you don't know what right would be, then how can you test? How do you know if your code works?
        - Perhaps requirements not known or stable
        - Make a decision
            - Your test document what you decided
            - Reassess if it changes later

Mock Objects

- Fake objects that replace real ones
- Why use them?
    - Avoid external dependencies
    - Reduce coupling
    - Keep tests fasts
    - Test object interactions
    - Promote interface based design
    - Ensure tests are durable

Use the compiler

- Let it tell you about errors and omissions
- Read carefully its warning and error messages

One assertion(check/assert) per test

- Subject of furious debate on Yahoo's TDD group

Use this four techniques

- Design: Do it the simplest thing
- Fake it("Till you make it")
- Triangulate
- Obvious implementation

Design: Do the simplest thing

- KISS Principle - Keep it simple and stupid
    - Consider the simplest thing that could possibly work
    - Simplicity is the ultimate sophistication

- When coding YAGNI - You ain't gonna need it
    - Build the simplest possible code that will pass the tests
    - DRY Princpile
        - Don't Repeat Yourself
            - Refactor the code to have the sumplest design possible

Fake it("Till you make it")

- What is your first implementation once you have a broken test?
    - Return a constant
    - Once you have a test running, gradually transform the constant into an expression using variables

Triangulate

- How do you most conservatively drive abstraction with tests?
- Abstract only when you have two or more examples

Obvious implementation

- How do you implement simple operations?
    - Just implement them
- Fake it and triangulation are for taking tiny steps
- If you know what to type, and you can so it quickly, then do it
- Keep track of how often you are surprised by red bars using obvious implementation

After the first cycle you have

- Made a list of tests we knew we needed to have working
- Told a story with a snippet of code about how we wanted to view one operation
- Made the test compile with stubs
- Made the test run by committing horrible sins
- Gradually generalized the working code, replacing constant with variables
- Added items to our to - do list rather than addressing then all at once

Refactoring

- Refactoring cannot change the semantics of the program under any circumstance
- This is called Observation equivalence
    - All test that pass before refactoring must pass after it

    - Places burden on you to have enough tests to detect unintended behavioral changes due to refactoring

Summary

- TDD does not replace traditional testing
- TDD isn't new
- TDD means less time spent in the debugger
- TDD negates fear
- TDD creates a set of "programming tests"
- TDD allows us to refactor, or change the implementation of class, without the fear of breaking it