Test-Driven Development (TDD)

Unit tests, mock data & mock objects

Tales From The Trenches: Complicated Code

ceOrganizer.aspx.vb

How Can You Program Safely?

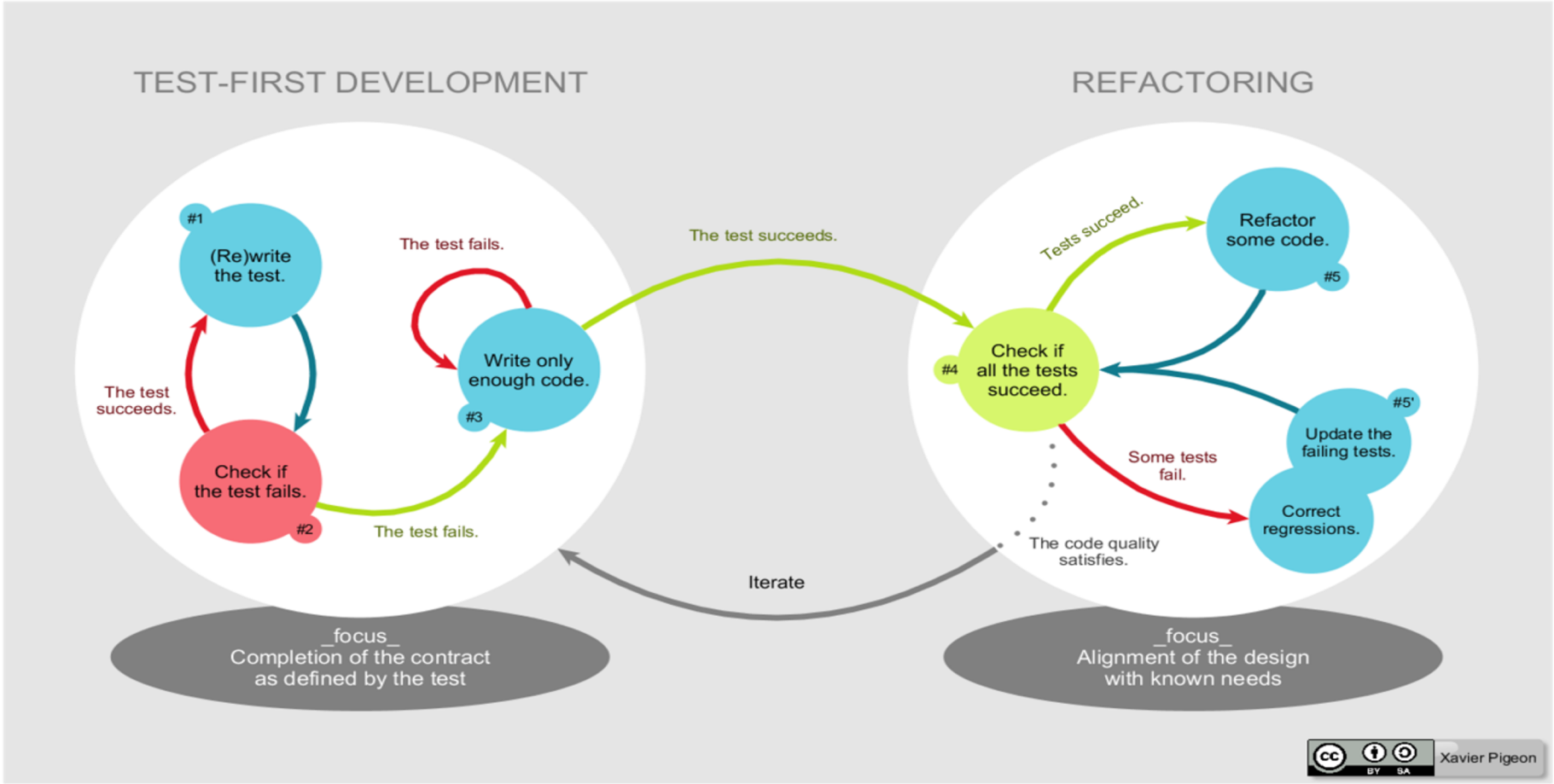
- Refactoring existing code:
  - *“Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code. The tests are essential because even though I follow refactorings structured to avoid most of the opportunities for introducing bugs, I’m still human and still make mistakes. Thus I need solid tests.”* Refactoring: Improving The Design Of Existing Code, Martin Fowler
  - You cannot **confidently** change massive codebases without unit tests, without them you are gambling
- New code:
  - TDD helps drive the **good** design of your code
  - Put faith in the long term benefits, you’re making an investment
  - Remember you are not only helping your future self, but building **value for your company**

Test Driven Development

1. Add a test
2. Run all tests and verify the new test fails
3. Write the code
4. Run the new test, it should now pass
5. Refactor code
6. Run tests, tests should continue to pass
7. Repeat

Good agile stories have acceptance criteria, good candidates for unit tests (acceptance test driven development, ATDD)

Write tests **FIRST** to maximize benefits of TDD



https://commons.wikimedia.org/wiki/File:TDD_Global_Lifecycle.png

Benefits Of Test Driven Development - (Wikipedia, TDD)

- Less debugging
  - Debugging is hit or miss, some developers are good at it, some not
  - Mediocre programmers are VERY bad at debugging (too hard or too much work)
- TDD drives design: you envision usage and adjust course
- Catch problems while they’re small, before they grow to pollute your code
  - Mediocre programmer: Will avoid a problem and code around it rather than fixing it, unit tests police them into good behaviour
- TDD forces modularized, flexible and extensible design and reduces coupling, which are **very** good things

Drawbacks Of Test Driven Development

- Mediocre programmers **hate it** and rebel (because it catches them!)
  - Write crappy tests that aren't comprehensive
  - Don't maintain / add to tests when modifying code
  - This is a leadership / management issue, management must not allow this
- Difficult (not impossible) to test for environmental issues (network, speed, browsers, etc…)
- Tests may share blind spots of code
- Maintenance overhead
- Systems dependent on existence or structure of content and assets
- Difficult to get management / bean counters on board
- Studies showing TDD bad? There's a couple too.
- Slower than high quality programmers combined with high quality QA and automated test suites. But that's like saying if we have too many unicorns we'll run out of grass.

Mocks

- “Mock objects are simulated objects that mimic the behaviour of real objects in controlled ways. A programmer typically creates a mock object to test the behaviour of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behaviour of a human in vehicle impacts.” - Wikipedia, “Mock Object”
- **Unit tests need to be simple**, and for that reason we need mock objects to simplify testing modules, insulating them from their requirements in other modules.
- **Unit tests need to be stable**, they will not be reliable if the rug is pulled out from underneath them (whether module dependencies, data, etc…)

Examples - When To Use Mocks

- Objects that interact with a data layer:
  - Database class:
    - Mock datasets so no need to actually talk to the database
    - If used DB, DB could be changed / restored from another DB and test data lost
  - Class that uses content managed outside dev team:
    - Mock interface to content to prevent outsiders breaking unit tests
- External resources / APIs:
  - E.g. Class uses a webservice, mock the webservice
  - E.g. DLL or module being developed separately? Mock the interface until its complete
- Slow processes
- Difficult to reproduce states (e.g. network errors, race conditions, etc...)

How Do We Use Mocks?

- When writing your code how do you write it so that it supports mock objects?
- **Dependency Injection!**
  - Wikipedia: “a technique whereby one object supplies the dependencies of another object…The client delegates the responsibility of providing its dependencies to external code (the injector).”
  - Rather than hardcoding your code to use data layer classes to load data, write your code to be passed an object that implements an interface
    - In your real code (the real injector) pass an instance of a data layer class that does the loading and implements the interface
    - In your unit tests, pass an instance of a mock object that also implements the interface, returning **fake but known, repeatable and testable data**

Best Practices (Wikipedia - TDD)

- Separate setup and teardown logic into test support services (not inside the test)
- Treat your test code like your production code, it’s just as important
- Discuss unit test coverage / practices with your team:
  - Ensure best practices and design choices are communicated across the team
    - Mocks are useless if nobody knows about them
    - If people don’t know about utility / mock objects, each developer ends up rolling their own, which bloats framework
- Don’t disable tests, fix them / find out why they’re failing and correct it

Anti-patterns (Practices To Avoid) (Wikipedia - TDD)

- Test cases based on system state manipulated by previously executed tests, i.e. start with a clean slate
- Dependencies between test cases (complex and brittle), don’t want to be spending time debugging tests!
- Testing system performance / timing
- Omega tests, “all-knowing oracles”, brittle / difficult to maintain
  - Tests should follow all clean coding best practices, including small functions that do one thing, and one thing only
- Testing inside the black box
- Slow running tests (use mocks to speed them up)

Project TDD Requirements

- All business logic must be 100% covered by unit tests
- Unit tests must thoroughly test not just standard inputs but also boundary conditions, exceptional inputs, bad inputs, etc. If something CAN go wrong, you should have a test to catch it and ensure your code handles the bad eventuality.
- **Unit test classes must SHADOW business logic classes**. I.e. if you have a class User, you need a test class UserTest where the unit tests for User are implemented. If it's hard to find your unit tests they don't count.