Strategies for Writing Automated Tests

Paulo Cheque

Summer 2009

About

- Refactoring
- TAD
- TFD/POUT
- TDD
- BDD
- Patterns and Anti-patterns

(Refactoring)

- A modification made step-by-step on the SUT that doesn't change the behavior, but improves its internal structure

(Refactoring - Catalog)

- Rename
- Extract Method
- Pull Up Method
- Encapsulate Field
- Replace Conditional with Polymorphism
- ...

(Refactoring - Motivation)

- Unreadable code
- Code replication
- Extensive code/Poorly organized
- A lot of parameters
- Useless comments
- Inappropriate intimacy
- ...

(Refactoring - Consequences)

- Improve design after the code has been written
- Agile development
- Minimizes the chances of introducing errors
- …

(Tests execution)

- Must be agile
- As soon as a new piece of code has been created
- On the same night, on a dedicated machine
- As soon as changes are detected in repository
- It's better to write and run tests that are incomplete than not to run tests that are complete

Test-After Development (TAD)

- To know the architecture
- Develop
- Test automatically

- Comparing with manual tests:
  - Avoid regression errors
  - Useful and updated documentation of what has been tested

TAD - Disadvantages

- SUT with little testability
- Change SUT to test, this makes sense?
- Hard to test
  - Anti-patterns

When should we use TAD

- Teams are learning automated tests
- Legacy systems
- Before refactorings
- Correction of errors: Simulate an error with a test

Test-First Development (TFD)

- Plain Old Unit Test (POUT)

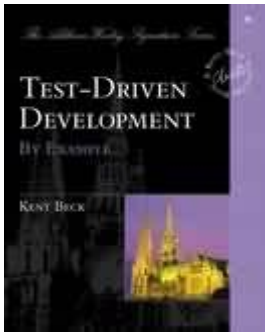
- To know the architecture
- Test
- Develop

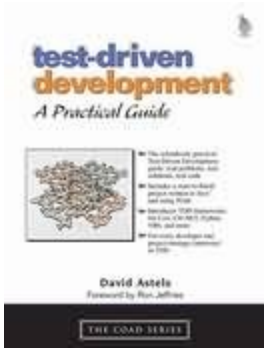
TFD - Characteristics

- Advantages:
  - More tests
  - Less pressure
  - More testability
- Disadvantages:
  - Additional study of design without proving it
  - Generation of code and useless tests

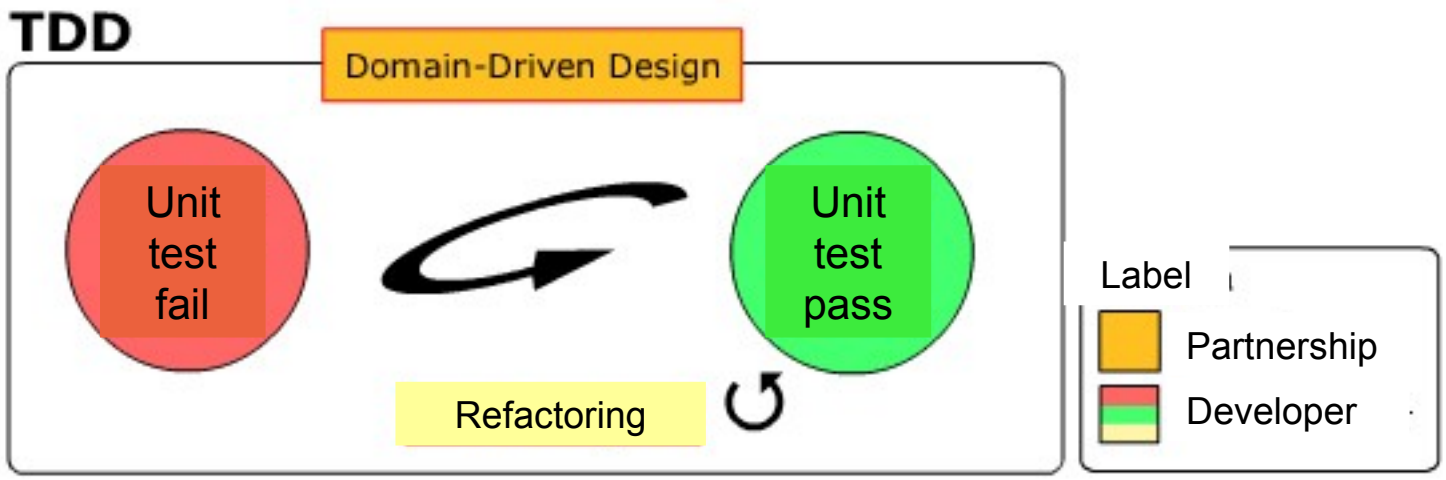
Test-Driven Development (TDD)

- TDD != TFD
- *"Clean code that works"* - Ron Jeffries
- Developing in small steps:
  - Write a test that fails
  - Develop enough code to make that test pass
  - Refactor it
- SUT do what tests are testing



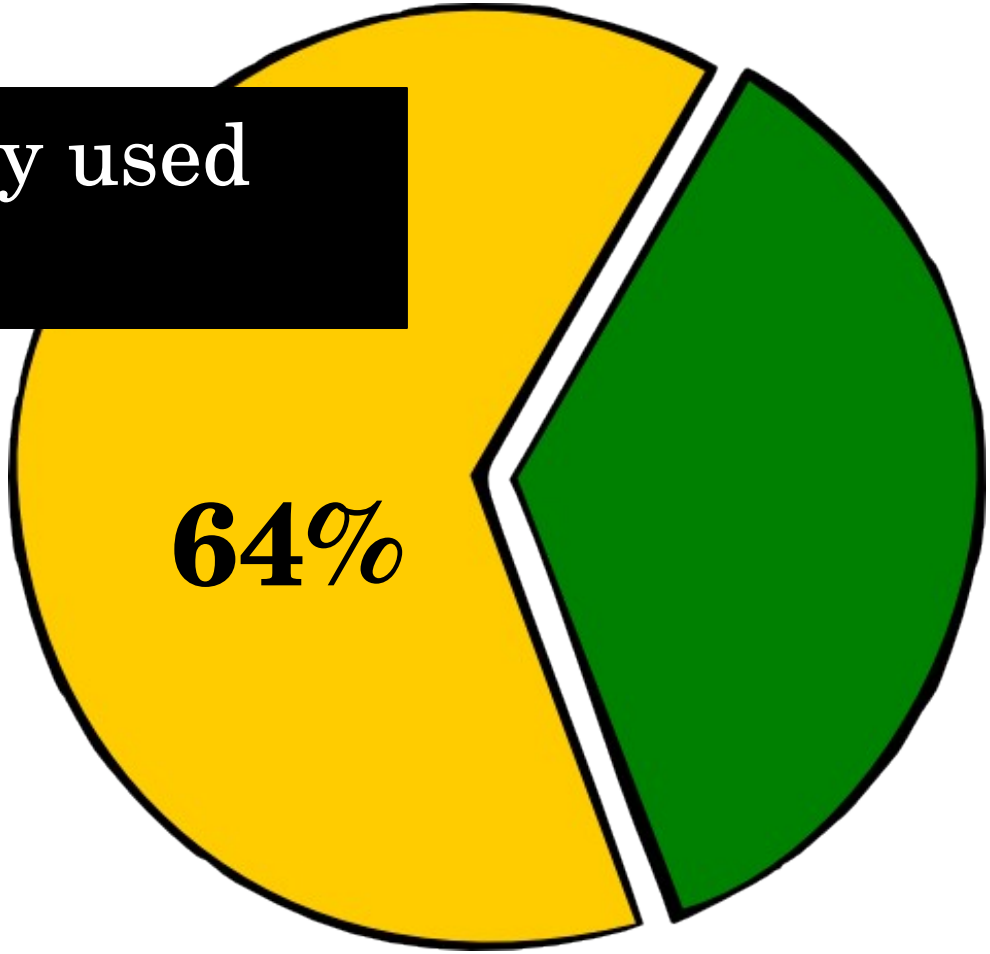


TDD


TDD
Domain-Driven Design
Unit test fail
Unit test pass
Refactoring
Label
Partnership
Developer

Waste

Never or rarely used features



TDD

- Design evolves with knowledge
  - *"Code for tomorrow, design for today"* - K. Beck
- Develop in small steps
- Express the intention of a programmer through the tests
- Tests are documentation

Design

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Dependency Injection

- Avoid object instantiation inside wrong objects
  - Constructor (Collaborator)
  - setCollaborator(Collaborator)
  - method(Collaborator)
- OO: Don't ask, tell!

Dependency Lookup

- Permits normal coupling between objects to be broken in test automation
- Implementation: Find dependencies through a register of dependencies service
- Example:
- getCurrentTimeAsHTML()
  - return “” + new TimeProvider().getTime() + “”

Humble Object

- Dependencies too coupled to frameworks
- Humble Object: Adapter connected to framework that redirects calls to a testable component

Test Hook

- Avoid
- Forbidden!
- Test logic in production
- if(TESTING) { do something }
- Else { do anything else }

TDD

- The source code:
  - Test names define expected behavior
  - Factored (without replication)
  - High coverage
  - Avoid useless code
  - High quality
    - Safe refactoring

TDD - Obstacles

- Hard to begin
  - Much different from traditional way of programming
- Hard to apply in complex systems
- Hard to apply in legacy systems

Red Bar Patterns

- One step: Start by what you know
- First test: Start testing simple situations
- Explanation test: Talk about SUT using test terms
- Test to learning: Create tests to know external applications
- Regression test: Write a test that simulates an error before trying to fix it

Green Bar Patterns

- Falsification: Until the real implementation
- Triangularization: Test two cases to find an abstraction
- Obvious Implementation: When it is possible
- One to Many: Before testing collections, test a unique element

Anti-Patterns

- Tests focused on implementation, not in feature
- Fragile tests
- Lack of tests of important cases

Classicist TDD vs Mockist TDD

- Classic TDD vs Mocked TDD
  http://martinfowler.com/articles/mocksArentStubs.html
- Classic:
  - Use real objects instead of mocks if it has a low cost
  - Mini-integration tests
- Mocked:
  - Isolates all dependencies, even if they are easily used

Classicist TDD

- 1. View + Controller Stubs
- 2. Model
- or
- 1. Model (DDD)
- 2. View and Controller without stubs or doubles

- Bugs: Affect tests of incorrect feature and higher layers

Mockist TDD

- Structured:

  View + mock the rest

  Controller + mock the rest

  Model + mock the rest (DDD)

- Bugs: Affect just relevant test cases

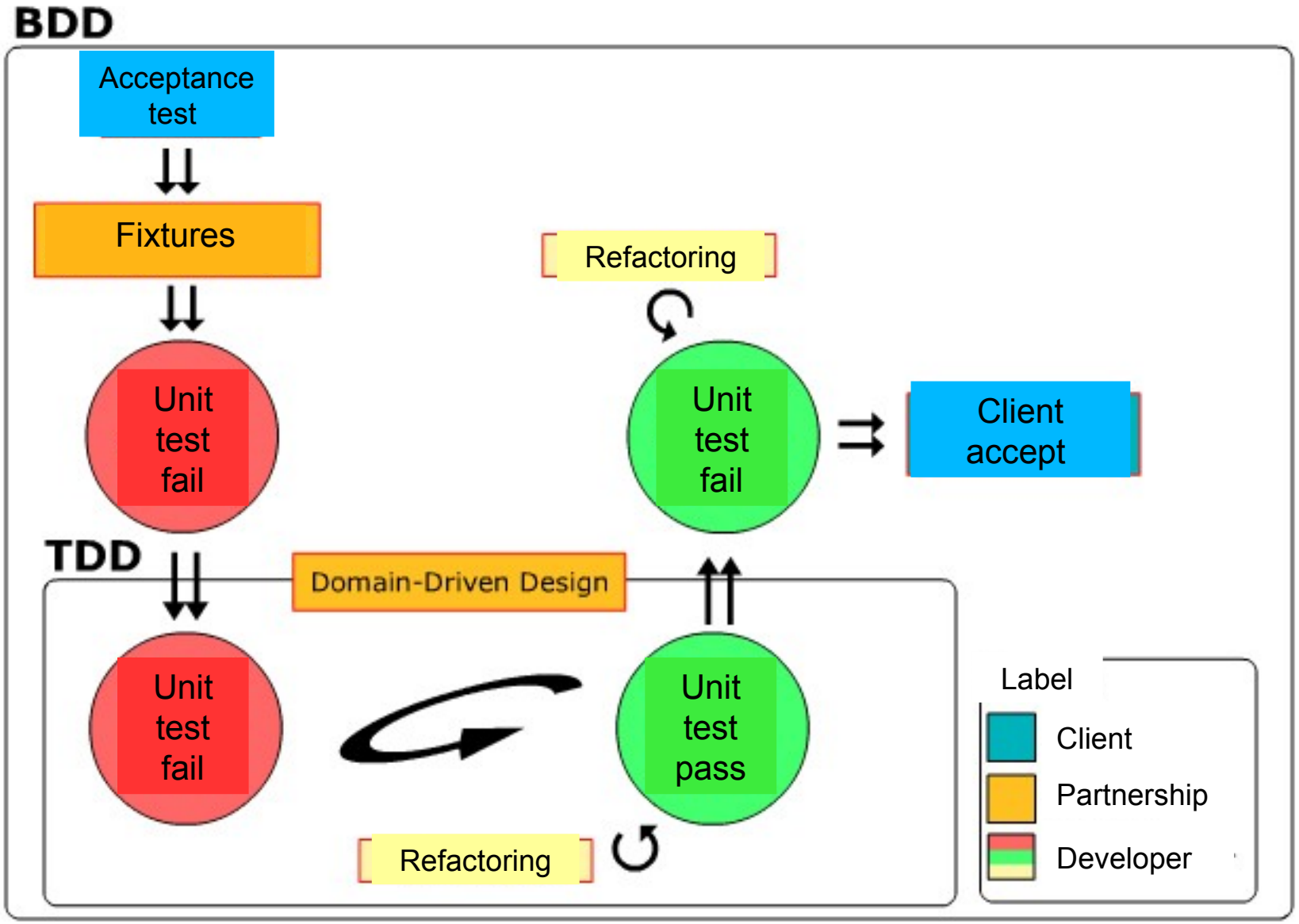
- Closer to implementation

Behavior Driven Development (BDD)

- Additional cycle for acceptance tests
- Pending tests
- Client language / Common language
- As a … / I want to … / So that …
- Given, When, Then
- Executable documentation

- Tools: FIT, rSpec

BDD


BDD
Acceptance test
Fixtures
Unit test fail
TDD
Domain-Driven Design
Unit test fail
Unit test pass
Refactoring
Refactoring
Unit test fail
Client accept
Label
Client
Partnership
Developer

Program Exploration (Pex)

- Microsoft research project
  http://research.microsoft.com/en-us/projects/Pex

- Generate input of data in tests through exploratory tests

- Try to understand source code of an algorithm to generate other pertinent test cases

Contact

http://www.agilcoop.org.br

http://ccsl.ime.usp.br

http://qualipso.org

agilcoop@agilcoop.org.br

paulocheque@agilcoop.org.br