Test Driven Development

An Introduction

By Robert Stehwien

---

Test Driven Development

- Introduction
- How
- Why
- Tips
- Credits
- Katas

---

What is Test Driven Development?

- An iterative technique to develop software
- As much (or more) about design as testing
  - Encourages design from API user’s point of view
  - Encourages testing classes in isolation
  - Produces loosely-coupled, highly-cohesive systems
- As much (or more) about documentation as testing
- Must be learned and practiced
  - If it feels natural at first, you’re probably doing it wrong
Automated Tests

- **Unit tests**
  - Tests that show the programmer that the code does what is expected
  - Specifies what the code must do
  - Provide examples of how to use the code (documentation)
  - All tests are run every few minutes, with every change

- **Acceptance tests**
  - Tests that show the stakeholders that the code delivers the feature
  - All tests are run at least daily

- All tests are Automated, you run them with every change

Uses of Automated Tests

- **Living documentation of design**
  - The tests fail if the design changes

- **Example of how to use an interface**

- **Watching for bugs**
  - Once a test passes, it is re-run with every change
  - Broken tests are not tolerated
  - Side affect defects are detected immediately
  - Assumptions are continually checked

- **Improving the design of code**

Test Driven Development

- **Introduction**

- **How**

- **Why**

- **Tips**

- **Credits**

- **Katas**
Test Driven Development Cycle

1. Add a test
2. Compile
3. Fix compile errors
4. Run all tests and see the new one fail
5. Write some code
6. Run the tests and see them succeed
7. Refactor code
   - Many small steps toward a goal often leads to a different or better place

Do the Simplest Thing

• Assume simplicity
  – Consider the simplest thing that could possibly work
  – Iterate to the needed solution
  – YAGNI
• When coding:
  – Simple, clear, and maintainable
  – Build the simplest possible code that will pass the tests
  – Refactor the code to
    • Have the simplest design possible
    • As clear and communicative as possible
  – Eliminate duplication

The Rules of Simple Design

IN PRIORITY ORDER!
1. The code passes all tests
2. There is no duplication
3. The code expresses the programmer’s intention
4. Using the smallest number of classes and methods

Higher priority rules must be satisfied first
What is Tested?

- Every class (module) has one or more unit tests
- Test everything that can possibly break
- If it can't break, don't test it
  - Always a judgment call

```cpp
int Student::GetName()
{
    return name;
}
```

Focus on Interface

- The test treats the object being tested like black box
- Encourages design to be done from a client point of view
  - The test is a user of the API
- You confront interface design issues
  - What are the parameters?
  - What is the return type?
  - What is the behavior?
  - Who controls object lifetime?
- Looks similar to Design by Contract

Testing Frameworks

- Tests must be automated
  - Otherwise they won’t be run
- Most OO languages have a testing framework, xUnit
  - JUnit, CppUnit(Lite), PyUnit, NUnit, VBUnit, Ruby’s Test::Unit
  - A simple tool
  - Collects, organizes and automatically calls your test code
- Graphical test runner
  - Green bar makes you feel good
- Could be added to build environment
JUnit Demo

- Wherein the instructor shows you how to make a simple unit test framework for a class

Test Driven Development

- Introduction
- How
- Why
- Tips
- Credits
- Katas

Objections Heard

- “I know how to just write the code, but I don’t know how to test it.”
- “We have to write twice as much code.”
- “I have to debug twice as much code.”
- “We have a testing department.”
- “I can test my code after I write it.”
- “That might work on easy software but our problem is really tough”
- “You need the target hardware”
Design Impacts

• Test-first design promotes testing a class in isolation
  − It must be decoupled from other classes
• Produces loosely coupled, highly cohesive systems
  − The hallmark of a good design
  − Object Oriented Design Principles and Programming Languages help

Productivity and Predictability

• Defects kill predictability:
  − Cost of fixing is not predictable
  − When they materialize is not predictable
• Test-driven is predictable:
  − Working at a steady pace
  − Results in fewer bugs
  − More productive than “debug-later programming”
• Test-driven programmers rarely need the debugger

 Sufficiency

• Produces higher quality code
  − Tests evolve as your understanding increases
  − Code evolves to satisfy the tests
• Provides only enough code to meet the tests
• Provides a high level of comfort that allows refactoring
Reward and Feedback

- Frequent feedback
- Make something work every programming session
- Check in functioning code every time

Test Driven Development

- Introduction
- How
- Why
- Tips
- Credits
- Katas

Learning Test-First Design

- A skill which must be practiced
  - Initially awkward
- Requires discipline
  - Peer pressure
  - “I know how to write the class, but I don’t know how to test it”
- It's an addiction rather than discipline
  - Kent Beck – Author of
    - Extreme Programming Explained
    - Test Driven Development
You Aren’t Going to Need It (YAGNI)

• Temptation to add methods or infrastructure
  – Fear that you might need it later
• Resist the temptation
• Let the user stories force you to add it later
  – If they do, it won’t be that much worse
  – If they don’t, you’ve saved time, effort, and complexity
• Generality is paid for when the need is proven

Leading The Witness

• Doesn’t eliminate all need for design
  – Combining design-up-front with TDD
  – Architecture design important
• Writing tests that move toward design
• Design patterns
• Refactoring toward patterns
• Knowing when to stop
  – When “good enough”

Toss It

• One of the best things a programmer can do
  – Throw out code
  – Throw out a design
• If you struggle with something all day
  – Pitch it
  – It’s only one day’s work
• “Don’t let the sun shine on bad code”
Pitch It

• If your design becomes more brittle over time
  – Pitch it
  – Don’t spend good money after bad
• Try incremental refactoring first
  – But when that’s not feasible, pitch it
  – You’ll save time and money

Working

• Create a list of tests you want to write
  – Start with the simplest
  – Add tests as you go
• End the day with a failing test
  – Tells you where to begin
  – Start your day by doing something
• Write the asserts first so they lead the code

Testing a System of Objects

• Sure, unit tests work fine for a simple class. But what about a class that collaborates with other classes?
• Home alarm system example
  – Front panel with LEDs, push buttons, times square display
  – Phone line
  – The hardware won’t be ready for 3 months (one week before delivery)
Collaborators

- Most classes being tested need collaborators
  - e.g., Panel and phone line.
- Sometimes you can test with the real collaborators
- Sometimes you can’t or shouldn’t
  - The hardware is not ready, or it is slow, or hard to control
  - It is difficult to get the response needed from the collaborator
  - Interactive
  - Client/Server
- Impersonate collaborators with a Mock Object

Mock Objects

- Remove unwanted dependencies with an interface

Thin Interfaces
Test Driven Development

- Introduction
- How
- Why
- Tips
- Credits
- Katas

Acknowledgements

- ObjectMentor
  - www.objectmentor.com
- "Agile Software Development: Principles, Patterns and Practices" by Robert C. Martin
- "Test-Driven Development by Example" by Kent Beck
- Wikipedia
- Code Katas by the Pragmatic Programmers
  - http://www.pragmaticprogrammer.com/