Test Driven Development

ScrumAlliance®

Certified ScrumMaster®

ScrumAlliance®

Certified Scrum Developer

ScrumAlliance®

Certified Scrum Product Owner

- http://agilemanifesto.org/

Test Driven Development

Agenda

Why Test Driven Development

Test Driven Development Principle

Demo

Hands On

Why Test Driven Development

Clean and Simple Design

TDD helps us to pay attention to the right   issue at the right time so we can make our   design cleaner . We can refine our design as we learn.

Significantly increase the flexibility of your system, allowing you to keep it clean.

Create a system design that has extremely low coupling.

Courage

TDD enables to gain confidence in code over time . As test accumulate , we gain confidence in the behavior of the system.

Fear of making change disappear

Technical Debt

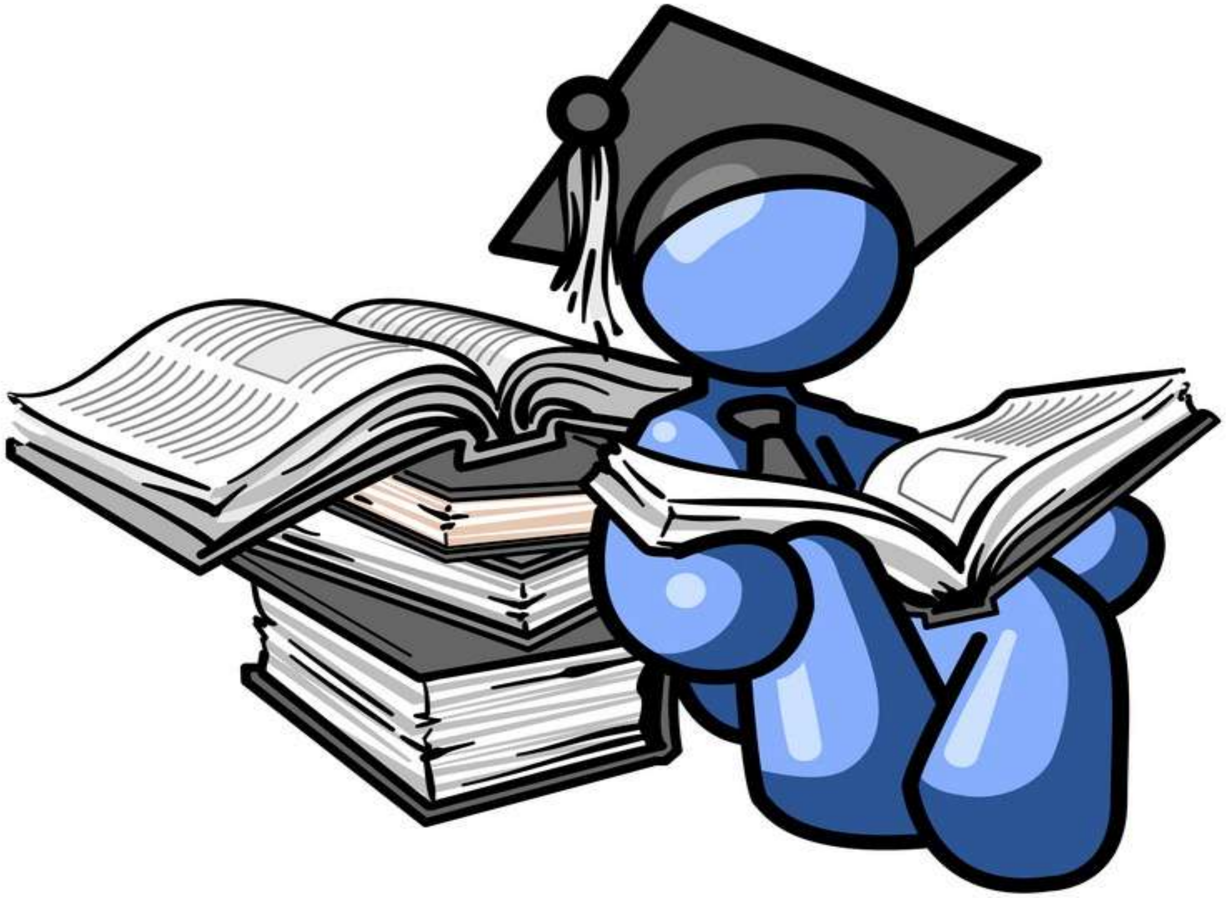
Technical Debt

Doing things the ***quick and dirty*** way sets us up with a technical debt, which is similar to a financial debt.

Like a financial debt, the technical debt incurs ***interest*** payments, which come in the form of the extra effort that we have to do in future development because of the ***quick and dirty design choice***.

We can choose to continue paying the interest, or we can pay down the principal by ***refactoring*** the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future.

Documentation



Create a suite of documents that fully describe the low level behavior of the system .

Low level documentation that executes

Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently

each person integrates at least daily - leading to multiple integrations per day.

Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Continuous Delivery

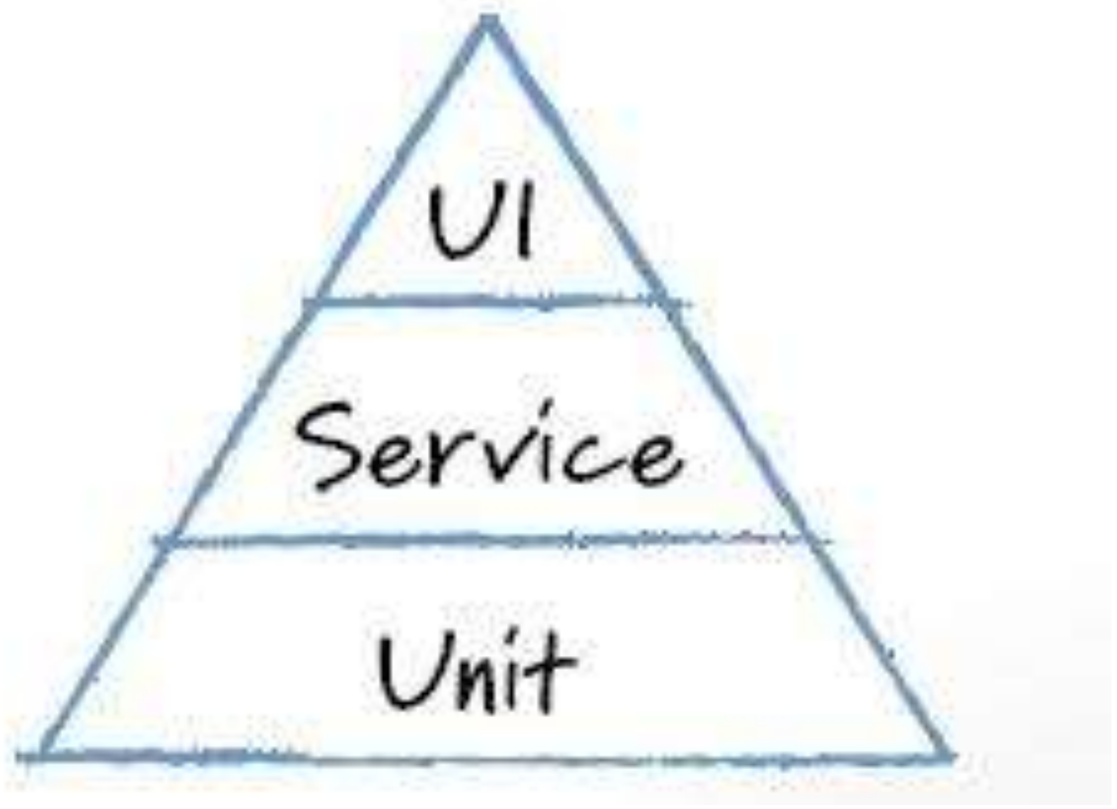
What is continuous delivery ?

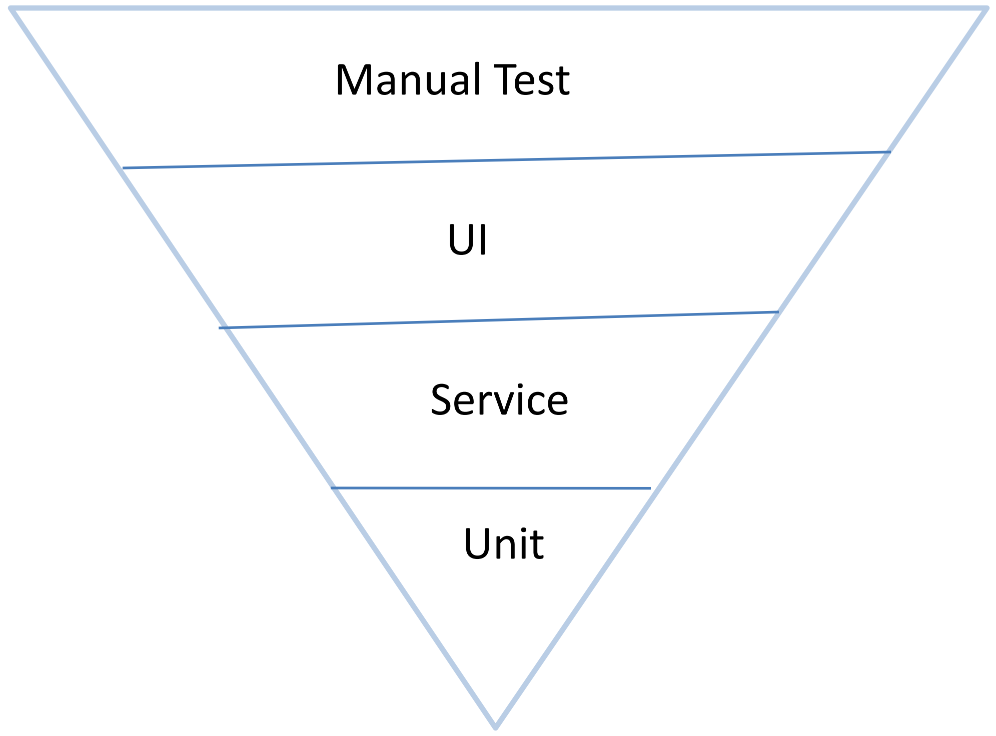
reduce the **cost** , **time** , **and risk** of
delivering **incremental changes**
to users

every requirement is a hypothesis

- Software always production ready
  - release tied to business needs , not IT constraints
- Minimize the lead time from idea to live
  - Concept to cash

Test Pyramid




Manual Test
UI
Service
Unit

The Continuous Delivery Maturity Model

	Base	Beginner	Intermediate	Advanced	Expert
Culture & Organization	• Prioritized work • Defined and documented process • Frequent commits	• One backlog per team • Share the pain • Stable teams • Adopt basic Agile methods • Remove boundary dev & test	• Extended team collaboration • Component ownership • Act on metrics • Remove boundary dev & ops • Common process for all changes • Decentralize decissions	• Dedicated tools team • Team responsible all the way to prod • Deploy disconnected from Release • Continuous improvement (Kaizen)	• Cross functional teams • No rollbacks (always roll forward)
Design & Architecture	• Consolidated platform & technology	• Organize system into modules • API management • Library management • Version control DB changes	• No (or minimal) branching • Branch by abstraction • Configuration as code • Feature hiding • Making components out of modules	• Full component based architecture • Push business metrics	• Infrastructure as code
Build & Deploy	• Versioned code base • Scripted builds • Basic scheduled builds (CI) • Dedicated build server • Documented manual deploy • Some deployment scripts exsists	• Polling builds • Builds are stored • Manual tag & versioning • First step towards standardized deploys	• Auto triggered build (commit hooks) • Automated tag & versioning • Build once deploy anywhere • Automated bulk of DB changes • Basic pipeline with deploy to prod • Scripted config changes (e.g. app server) • Standard process for all environments	• Zero downtime deploys • Multiple build machines • Full automatic DB deploys	• Build bakery • Zero touch continuous deployments
Test & Verification	• Automatic unit tests • Separate test environment	• Automatic integration tests	• Automatic component tests (isolated) • Some automatic acceptance tests	• Full automatic acceptance tests • Automatic performance tests • Automatic security tests • Risk based manual testing	• Verify expected business value
Information & Reporting	• Baseline process metrics • Manual reporting	• Measure the process • Static code analysis • Scheduled quality reports	• Common information model • Traceability built into pipeline • Report history is available	• Graphing as a service • Dynamic test coverage analysis • Report trend analysis	• Dynamic graphing and dashboards • Cross silo analysis

Companies doing Continuous Delivery

amazon

Google

YAHOO!

flickr

twitter

facebook

Etsy

Test Driven Development Principle

TDD Mantra

Red – Write a little test that does not work , and perhaps does not even compile first.

Green – Make the test work quickly , committing whatever sins necessary in the process.

Refactor – Eliminate all of the duplication created in merely getting the test to work.

Keeping Test Clean

Test code is just as important as production code. It is not second class citizen.

<u>What makes test clean</u>

Readability , Clarity , simplicity and density of expression

Arrange

Act

Assert

F.I.R.S.T

Fast

Tests should be fast. They should run quickly.

When tests run slow, you won’t want to run them frequently. If you don’t run them frequently, you won’t find problems early enough to fix them easily. You won’t feel as free to clean up the code. Eventually the code will begin to rot.

Independent

Tests should not depend on each other.

One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like.

When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

<u>Repeatable</u>

Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, without a network.

If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.

<u>Self-Validating</u>

The tests should have a boolean output. Either they pass or fail.

You should not have to read through a log file to tell whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

<u>Timely</u>

The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass.

If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

Tools

- Visual Studio
- Resharper

- RhinoMock
- Fitness
- Specflow

Just do it!

Coding Dojo

What is coding dojo

A Coding Dojo is a meeting where a bunch of coders get together, code, learn, and have fun

The real point of going to a dojo is to improve your skills

- http://www.meetup.com/OsloCodingDojo/photos/1237021/#21061984

Demo!

Convert Arabic number to Roman Number

Rules

- Write test before writing any production code
- Everyone must work in pair
- One person should write one failing test and other should make that test pass by writing as minimum code as possible
- Refcator code when necessary . You can only refactor code when all tests are in green
- Write elegant code

Questions?