Growing Object-Oriented Software

Chapter 1: What is the Point of Test-Driven Development?

"One must learn by doing the thing; for though you think you know it, you have no certainty until you try." - Sophocles

Software as a Learning Process

Almost all software projects are attempting something that nobody has done before (or at least nobody in the organization has done before).

Everyone involved in the project has to learn as it progresses. They need a process that will help them cope with uncertainty as their experience grows - to anticipate unanticipated changes.

Feedback is the Fundamental Tool

The best approach a team can take is to use empirical feedback to learn about the system and its use.

Every time a team deploys, its members have an opportunity to check their assumptions against reality.

Without deployment, the feedback is not complete.

Feedback is the Fundamental Tool

At each deployment we can:

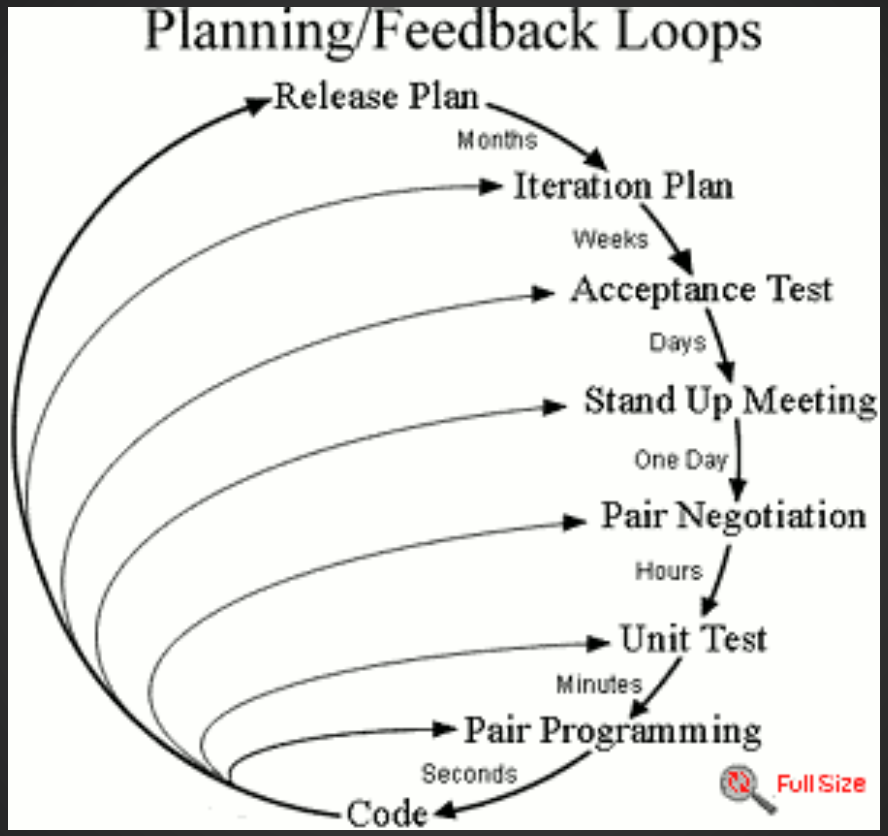
- Measure how much progress we're really making
- Detect and correct any errors
- Adapt the current plan in response to what we've learned.

Feedback Loops

Development can be broken up into a system of nested feedback loops, such as:

- Pair Programming
- Unit Tests
- Acceptance Tests
- Daily Meetings
- Iterations
- Releases
- etc...

Feedback Loops



http://www.extremeprogramming.org/map/images/loopsml.gif

Feedback Loops

Each loop exposes the team's output to empirical feedback so that the team can discover and correct any errors or misconceptions.

The nested feedback loops reinforce each other; if a discrepancy slips through an inner loop, there is a good change an outer loop will catch it.

Feedback Loops

The inner loops are more focused on the technical detail: what a unit of code does, whether it integrates with the rest of the system.

The outer loops are more focus on the organization and the team: whether the application serves its users' needs, whether the team is as effective as it could be.

Incremental and Iterative Development

In a project organized as a set of nested feedback loops, development is incremental and imperative.

Incremental and Iterative Development

Incremental development builds a system featured by feature. Each feature is always implemented as an end-to-end "slice" through all relevant parts of the system.

Iterative development refines the implementation of features in response to feedback until they are good enough.

Practices That Support Change

First, constant testing is needed to catch regression errors, and allow addition of new features without breaking existing ones.

Frequent manual testing is impractical, so we must automate testing as much as we can to reduce the costs of building, deploying, and modifying versions of the system.

Practices That Support Change

Second, the code needs to be as simple as possible, so it is easy to understand and modify.

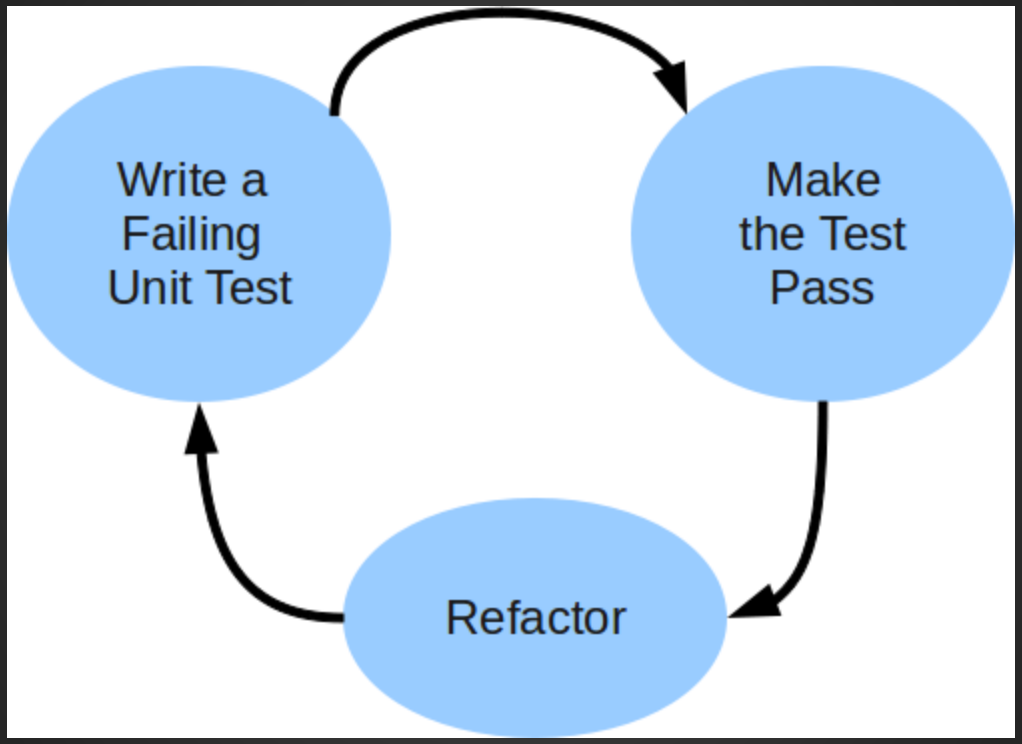
Simplicity takes effort, so we constantly refactor our code as we work with it - to improve and simplify its design, to remove duplication, and to ensure that it clearly expresses what it does.

Test-Driven Development

The cycle at the heart of TDD is: write a test; write some code to get it working; refactor the code to be as simple as possible.

TDD can give us feeback on the quality of both implementation ("Does it work?") and design ("Is it well structured?")

The TDD Cycle



http://www.kitware.com/blog/files/15_788403459_TestDrivenDevelopment.png

Writing Tests

- Makes us clarify the acceptance criteria for the next piece of work
- Encourages us to write loosely coupled components, so they can be easily tested in isolation and, at higher levels, combined together
- Adds an executable description of what the code does
- Adds to a complete regression suite

Running Tests

- Detects errors while the context is fresh in our mind
- Lets us know when we've done enough, which discourages the addition of unnecessary features

The Golden Rule of TDD

Never write new functionality without a failing test.

Test-Driven Development

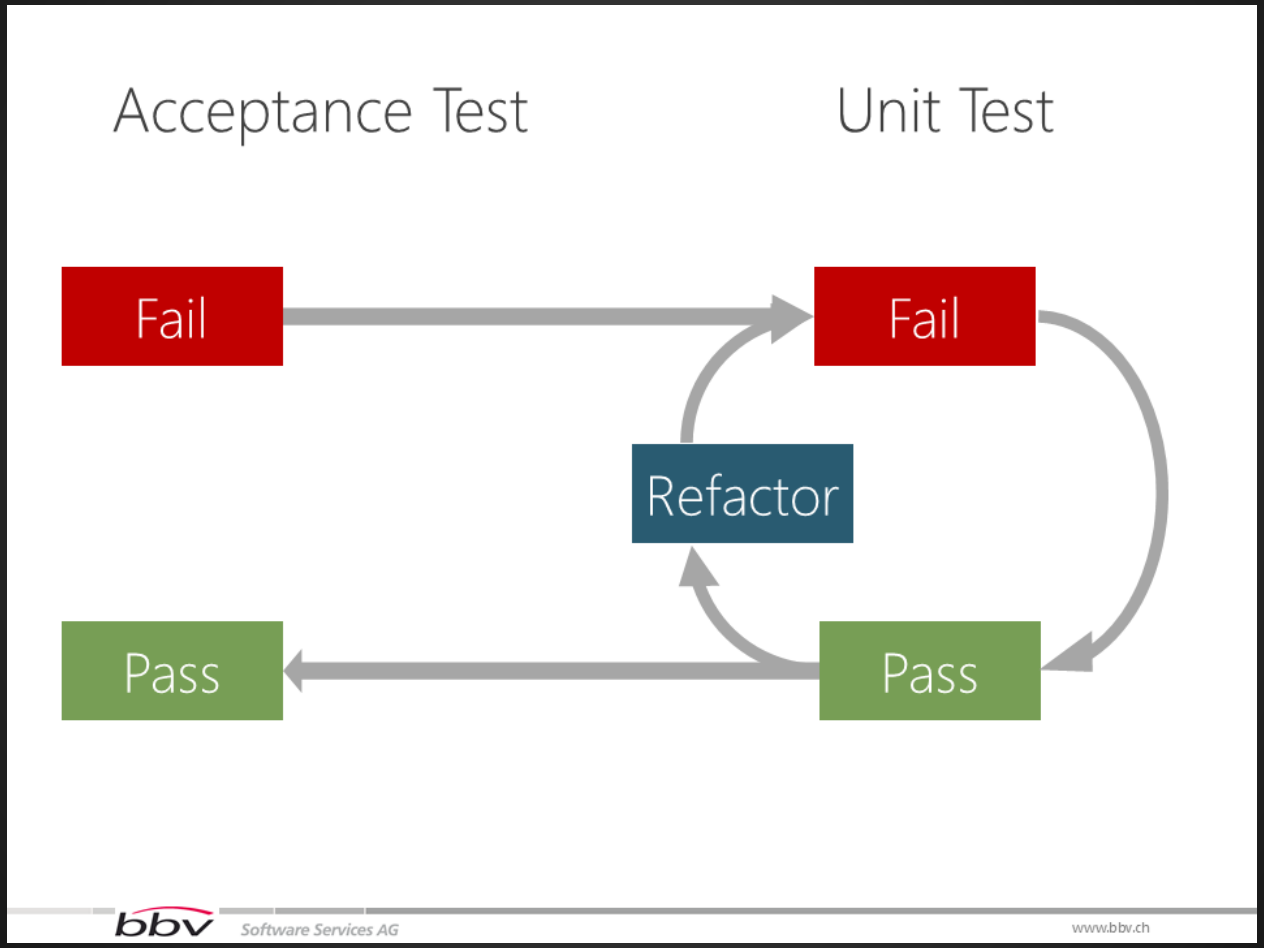
The effort of writing a test first gives us rapid feedback about the quality of our design ideas - that making code accessible for testing often drives it towards being cleaner and more modular.

The Bigger Picture

When we're implementing a feature, we start by writing an acceptance test, which exercises the functionality we want to build.

Underneath the acceptance test, we follow the unit test / implement / refactor cycle to develop the feature.

The Bigger Picture



http://www.planetgeek.ch/wp-content/uploads/2012/06/ATDD-cycle.png

The Bigger Picture

The outer loop is a measure of demonstrable progress, and the growing suite of tests protects us against regression failures when we change the system.

Acceptance tests are distinguished between tests we're working on, and tests for features that have been finished (which must always pass).

The Bigger Picture

The inner loop supports the developers. The unit tests help us maintain the quality of code and should pass soon after they've been written.

Failing unit tests should NEVER be committed to the source repository.

Testing End-to-End

Whenever possible, an acceptance test should exercise a system end-to-end without calling any internal code.

An end-to-end test interacts with the system only from the outside: through its user interface, by sending messages as if from third-party systems, by invoking its web services, by parsing reports, etc.

Testing End-to-End

An automated build, usually trigger by someone checking code into the source repository, will:

- check out the latest version
- compile and unit-test the code
- integrate and package the system
- perform a production-like deployment
- exercise the system through its external access points

Levels of Testing

- Acceptance: Does the whole system work?
- Integration: Does our code work against code we can't change?
- Unit: Do our objects do the right thing, and are they convenient to work with?

Acceptance Tests

Help us understand and agree on what we are going to build next.

They also help us make sure we have not broken any existing features as we continue developing.

Acceptance Tests

"Developers are responsible for proving to their customers that the code works correctly, not customers proving the code is broken."

http://www.extremeprogramming.org/introduction.html

Integration Tests

Refers to tests that check how some of our code works with code from outside the team that we can't change, such as a persistence mapper.

Integration tests make sure that any abstractions we build over third-party code work as we expect.

External and Internal Quality

External quality is how well the system meets the needs of its customers and users (is it functional, reliable, available, etc.)

Internal quality is how well the system meets the needs of its developers and administrators (is it easy to understand, easy to change, etc.)

External and Internal Quality

Internal quality is what lets us cope with continual and unanticipated change.

The point of maintaining internal quality is to allow us to modify the system's behavior safely and predictably.

Acceptance Tests

Running acceptance tests tells us about the external quality of the system, and writing them tells us how well we understand the domain.

However, end-to-end tests do not tell us how well we've written the code.

Unit Tests

Writing unit tests give us a lot of feedback about the quality of our code, and running them tells us that we haven't broken any classes.

However, unit tests don't give us enough confidence that the system as a whole works.

Integration Tests

Fall somewhere in the middle of unit tests and acceptance tests, in terms of internal and external quality.

External and Internal Quality

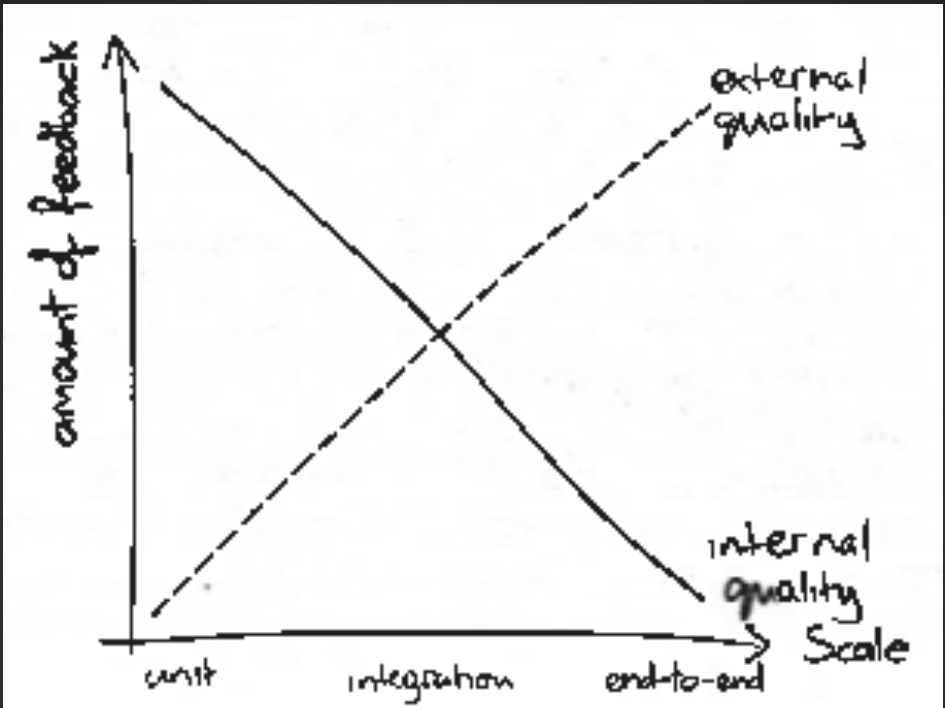


Figure 1.3 *Feedback from tests*

http://meekrosoft.files.wordpress.com/2010/10/gooagbt-chart.png

External and Internal Quality

Thorough unit testing helps us improve the internal quality because, to be tested, a unit has to be structured to run outside the system in a test fixture.

For a class to be easy to unit test, the class must have explicit dependencies that can be easily substituted, and clear responsibilities that can be easily invoked and verified.

Coupling and Cohesion

Elements are *coupled* if a change in one forces a change in the other. For example, if two classes inherit from a common parent, then a change in one class might affect the other.

An element's *cohesion* is a measure of whether its responsibilities form a meaningful unit. For example, a class that parses both dates and URLs is not coherent.