Test Driven Development

David Ehringer

davidehringer.com

Contents

- [ ] What Is TDD?
- [ ] TDD Principals
- [ ] Tools
- [ ] Live Coding
- [ ] Best Practices And Smells
- [ ] Other (Tips, Resources, BDD, Etc.)

Background

What is TDD?

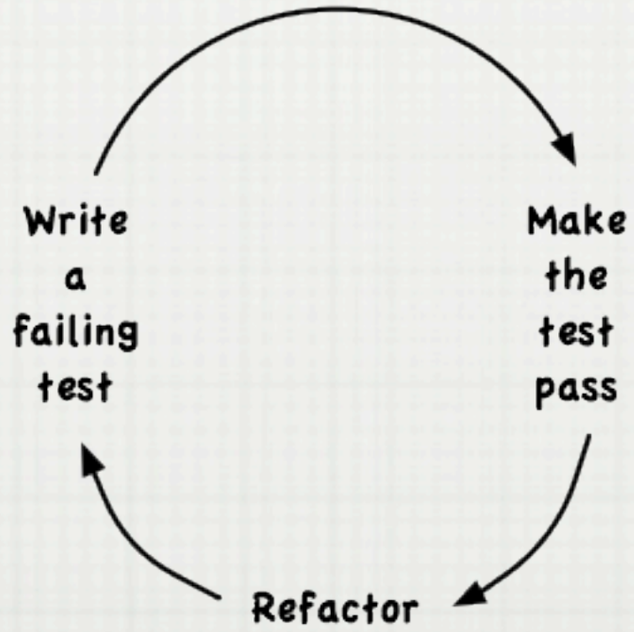
A Perspective On TDD

- ☐ TDD is Mostly about Design
- ☐ Gives Confidence
- ☐ Enables Change
- ☐ Is Automated
- ☐ Validates Your Design
- ☐ Is Documentation By Example
  - ☐ As Well As an Up To Date, "Live", And Executable Specification
- ☐ Provides Rapid Feedback
  - ☐ Quality of Implementation
  - ☐ Quality of Design
- ☐ Forces Constant Integration
- ☐ Requires More Discipline
- ☐ Brings Professionalism to Software Development
- ☐ Isn't The Only Testing You'll Need To Do
- ☐ Developers Write Tests

Result

- ☐ Higher Quality
- ☐ Flexibility
- ☐ Readability
- ☐ Maintainability
- ☐ Predicability
- ☐ Simplicity
- ☐ The Hard Stuff And Surprises Are Tackled Early On
- ☐ Both Internal And External Quality Are Maintained
- ☐ A Well-Designed Application

The Basic Cycle



The Three Laws of TDD

- [ ] 1: You May Not Write Production Code Until You Have Written A Failing Unit (or Acceptance) Test
- [ ] 2: You May Not Write More Of A Unit (or Acceptance) Test Than Is Sufficient To Fail, And Compiling Is Failing
- [ ] 3: You May Not Write More Production Code Than Is Sufficient To Pass The Currently Failing Test

Ever so slightly adapted from Professionalism and Test-Driven Development by Robert "Uncle Bob" C. Martin

Fundamental Principles

- ☐ Think About What You Are Trying To Do
- ☐ Follow The TDD Cycle And The Three Laws
- ☐ Never Write New Functionality Without A Failing Test
- ☐ Continually Make Small, Incremental Changes
- ☐ Keep The System Running At All Times
  - ☐ No One Can Make A Change That Breaks The System
  - ☐ Failures Must Be Address Immediately

In reality, unwavering adherence the cycle and laws can sometimes be very hard. In my opinion, it doesn't always have to be all or nothing (especially when you are just learning). But I've found that when I don't write tests first, I usually regret it afterwards. Also, TDD isn't applicable for every single scenario or line of code.

"TDD in a Nutshell"

As we develop the system, we use TDD to give us feedback on the quality of both its implementation ("Does it work?") and design ("Is it well structured?"). Developing test-first, we find we benefit twice from the effort. Writing Tests:

- makes us clarify the acceptance criteria for the next piece of work - we have to ask ourselves how we can tell when we're done (Design);
- encourages us to write loosely coupled components, so they can easily be tested in isolation and, at higher levels, combined together (Design);
- adds an executable description of what the code does (Design); and,
- adds to a complete regression suite (Implementation)

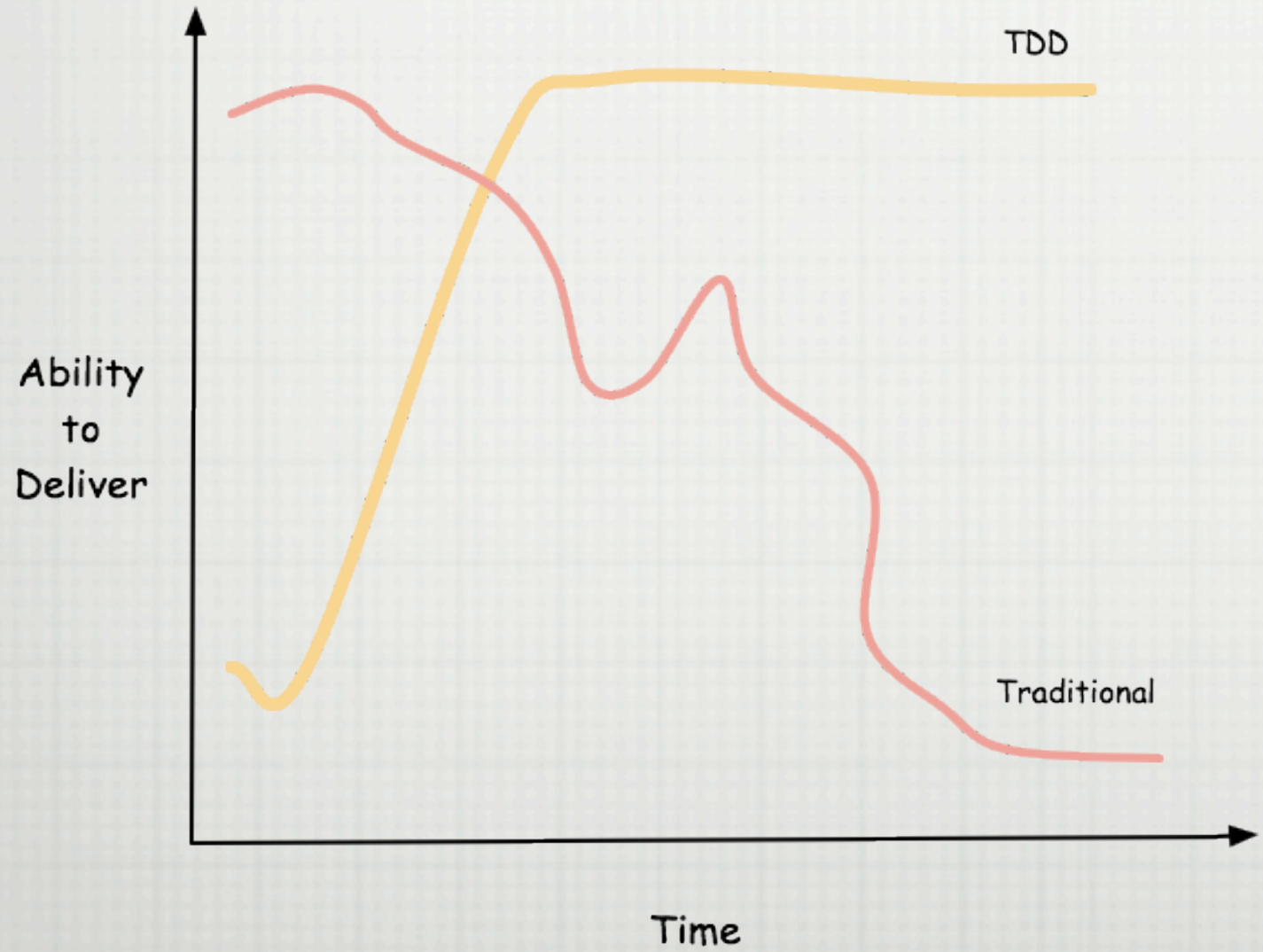
whereas running tests:

- detects errors while the context is fresh in our mind (Implementation); and,
- lets us know when we've done enough, discouraging "gold plating" and unnecessary features (Design)

From Growing Object-Oriented Software, Guided By Tests

But How Much Longer Does TDD Take?

"If it doesn't have to work, I can get it done a lot faster!" - Kent Beck paraphrased



My Experience
- Initial Progress Will Be Slower
- Greater Consistency
- Less Experienced Developer Learning/Ramp-up Time
- I personally think I can develop faster using TDD than without, but it took a while to get there
- Long-term cost is drastically lower

Studies
- Takes 15-30% longer
- 45-90% fewer bugs
- Takes 10x as long to fix a bug in later phases
- One study with Microsoft and IBM: http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

Some Design Principals

- ☐ Loosely coupled, highly cohesive
- ☐ Dependency Injection/Inversion of Control (IOC)
- ☐ Single Responsibility Principal
- ☐ Strive for Simplicity
- ☐ Open/Closed Principal
- ☐ Law of Demeter ("Tell don't ask.")
- ☐ Modularization
- ☐ Separation of Concerns
- ☐ Encapsulation
- ☐ Favor Immutability
- ☐ Domain Driven Design
- ☐ Decentralized Management and Decision Making
- ☐ Strict Dependency Management
- ☐ Clean Code
- ☐ Ubiquitous Language
- ☐ Eliminate Duplication

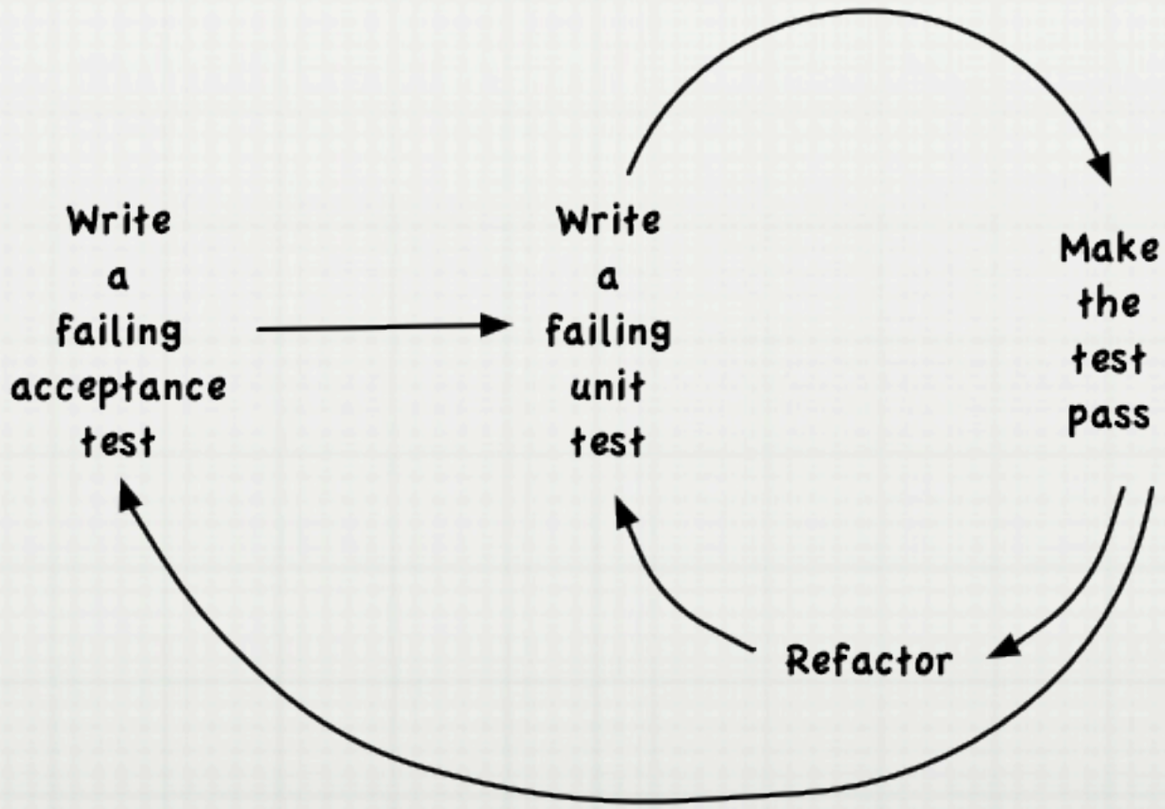
What About Architecture and Design?

- ☐ TDD Does Not Replace Architecture Or Design
- ☐ You Need To Do Up Front Design And Have A Vision
- ☐ But, You Should Not Do "Big Up Front Design"
- ☐ Defer Architectural Decisions As Long As Is "Responsibly" Possible
- ☐ TDD Will Inform And Validate (or Invalidate) Your Design Decisions
- ☐ TDD Will Almost Undoubtedly Uncover Weaknesses And Flaws In Your Design...Listen To Them!

Categories Of Tests

- ☐ Unit
- ☐ Integration
- ☐ Acceptance
  - ☐ Can Encompass "system" Or "functional" Tests
  - ☐ End-to-end Is Desirable When Feasible
- ☐ Learning Or Exploratory

The Extended Cycle



From Growing Object-Oriented Software, Guided By Tests

Tools Of The Trade

- ☐ Unit Test Framework
- ☐ Mocking Framework
- ☐ Automated Build Tool
- ☐ Acceptance Test Framework (often The Same As The Unit Test Framework)
- ☐ Source Code Management
- ☐ Build And Continuous Integration Servers
- ☐ Test Coverage Tools
- ☐ Code Quality Tools
- ☐ For Java-based Apps, Consider Using A Dynamic Language (e.g. Groovy) To Write Tests If You Are Comfortable With It

Tools Of The Trade - Recommendations

- ☐ Unit Test Framework - JUnit 4, Hamcrest Matchers Are Recommended For Improved Readability And Expressiveness
- ☐ Mocking Framework - Mockito
- ☐ Automated Build Tool - Maven
- ☐ Acceptance Test Framework - Web Services: SoapUI, Web App: Selenium
- ☐ Source Code Management - Depends On The Project Needs
- ☐ Continuous Integration - Bamboo Is Nice, Should Be Simple And Easy To Use And Support Your Build Tool
- ☐ Test Coverage Tools - Cobertura Or Clover (soapUI Pro For Web Services)
- ☐ Code Quality Tools - PMD, FindBugs, Checkstyle

Some Notes On Your Build

- ☐ Should Be Fully Automated
- ☐ Should Run Anywhere And Be Decoupled From The IDE
- ☐ You Should Be Able To Build Your Artifact In One Command

"mvn package"

- ☐ You Should Be Able To Run Your Unit Tests In One Command

"mvn test"

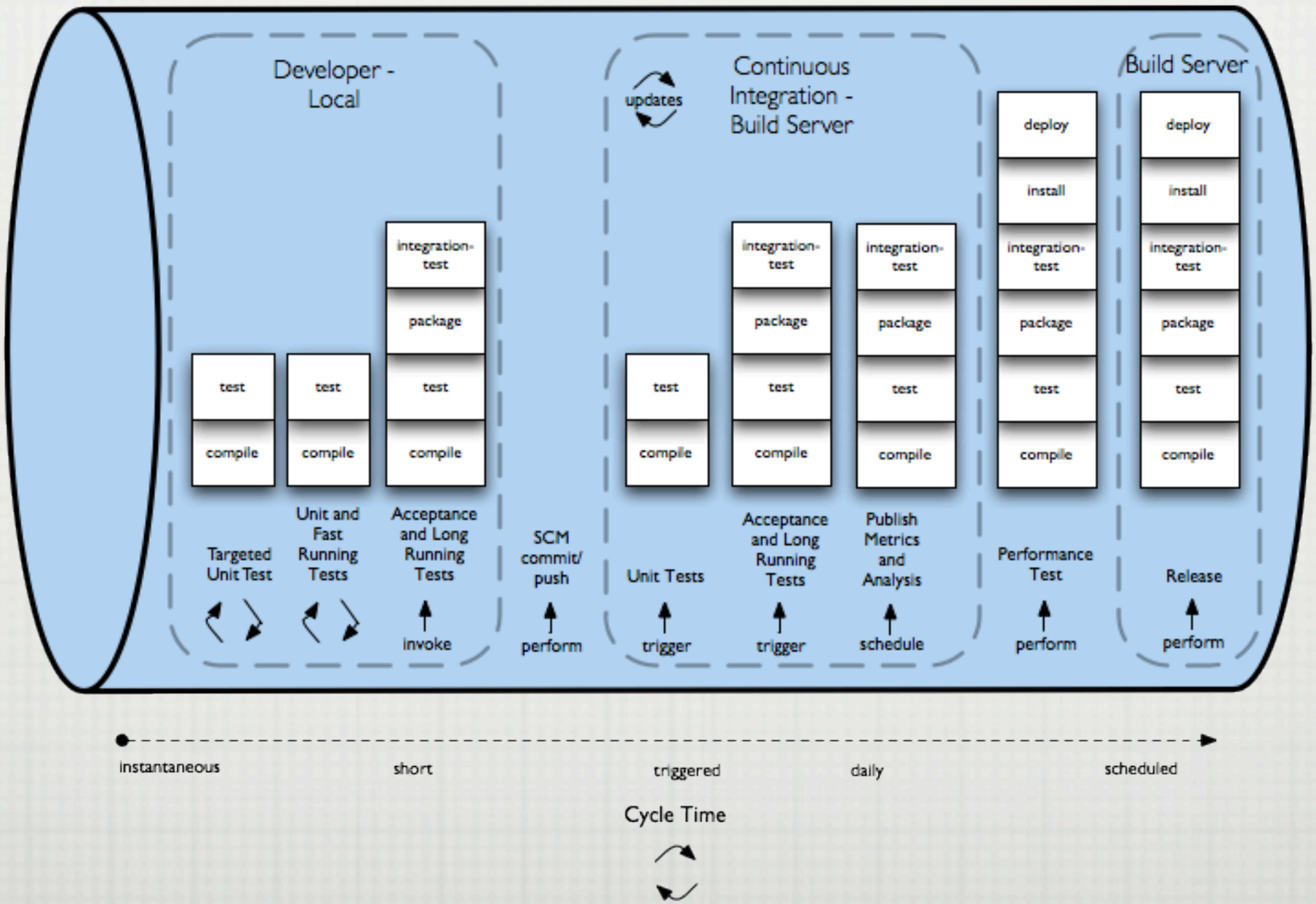
- ☐ You Should Be Able To Run You Acceptance Tests In One Command

"mvn integration-test"

- ☐ Test Coverage And Code Quality Tools Should Be Integrated Into The Build

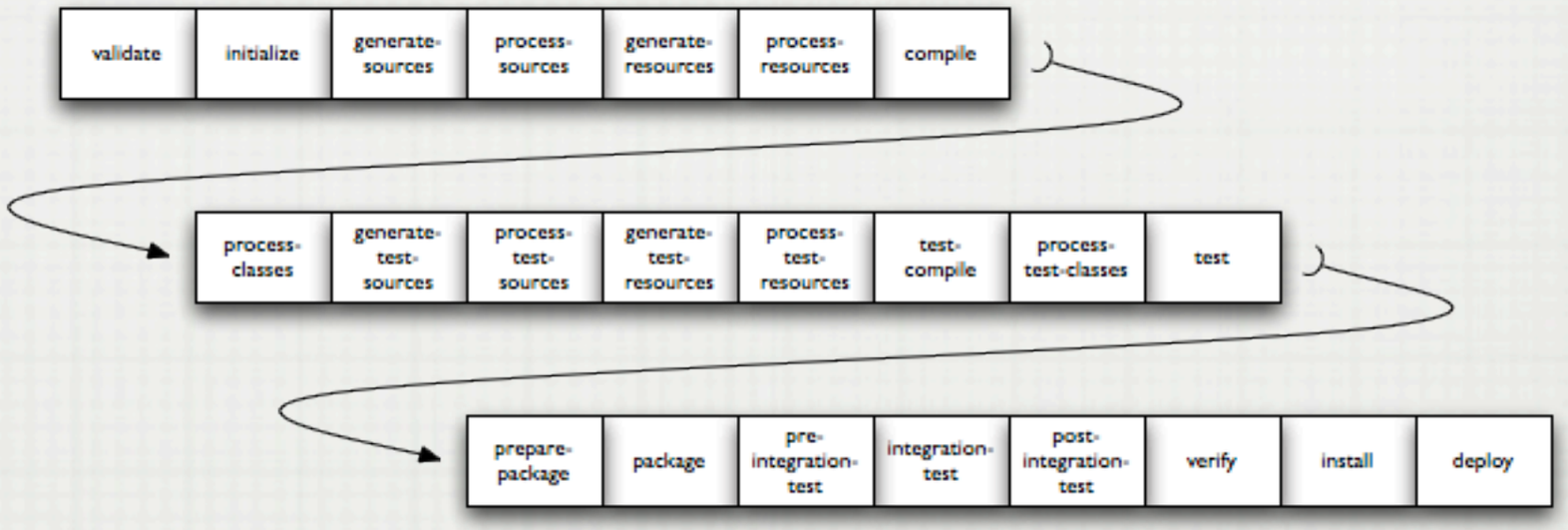
Build Pipeline

Example Build Pipeline



Maven Build Lifecycle

Maven Default Lifecycle



Mocking

- [ ] Limit The Scope Of Your Test To The Object Under Test
- [ ] Mocks Mimic The Behavior Of Objects (the Dependencies Of Your Object Under Test) In Predictable Ways
- [ ] Mocks Contains Assertions Themselves
- [ ] Prefer Mocking Of Behavior Rather Than Data (strictly Stubs)
- [ ] Be Careful: Mocks Can Couple The Test To The Underlying Implementation Of The Code Being Tested

TDD By Example

Time For Some Live Coding

But First... A Few More Words

- ☐ Iteration 0/Walking Skeleton
- ☐ Build, Deploy, Test End-to-end
  - ☐ Includes Process As Well As Infrastructure

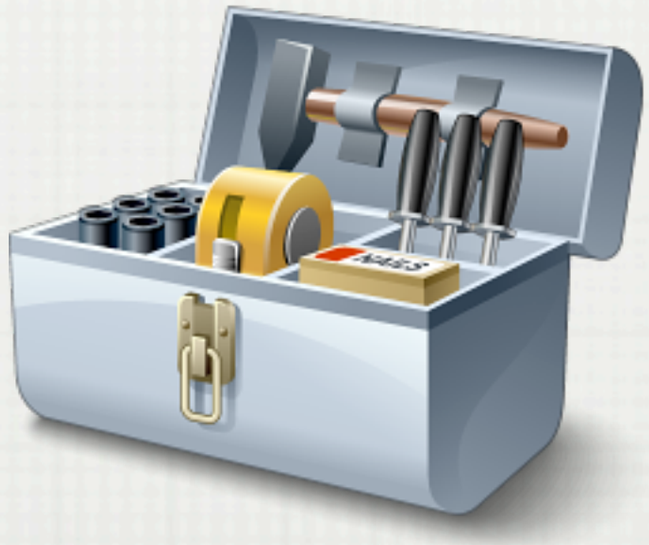
The Example

- [ ] A Trivial Customer Management App
- [ ] Web-service, Business Logic, Database
- [ ] Starts With A Full, Walking Skeleton

Back To The Examples

- ☐ Starting With An Acceptance Test
- ☐ Unit Tests To Fill In The Details
- ☐ Refactoring Along The Way
- ☐ Components/Modules And Large Projects

Code Coverage and Quality Tools



Non-Live Examples

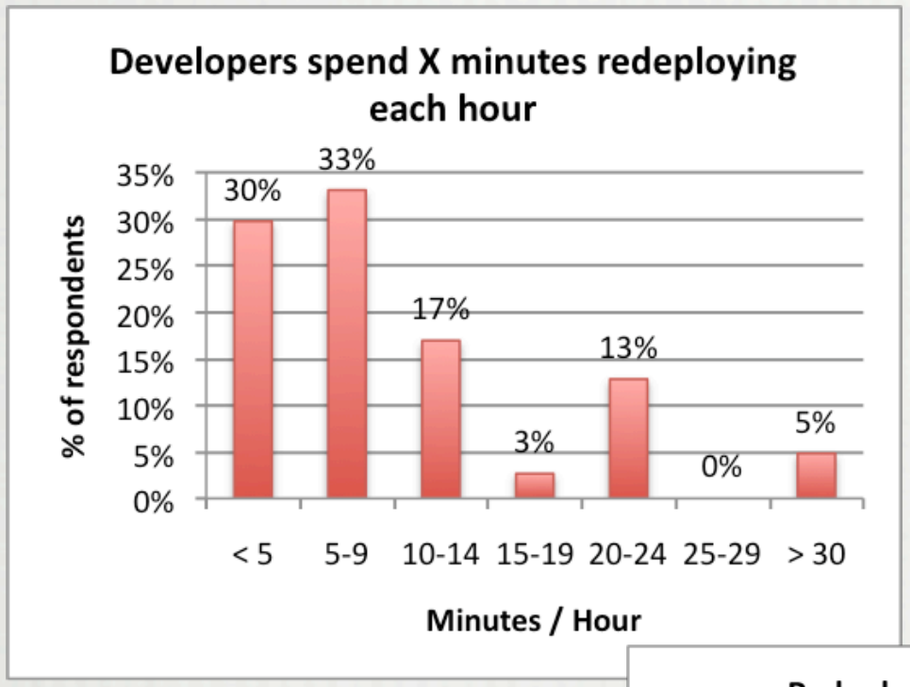
- [ ] Learning/Exploratory
- [ ] Mocking
- [ ] Test Data Builders
- [ ] Others?

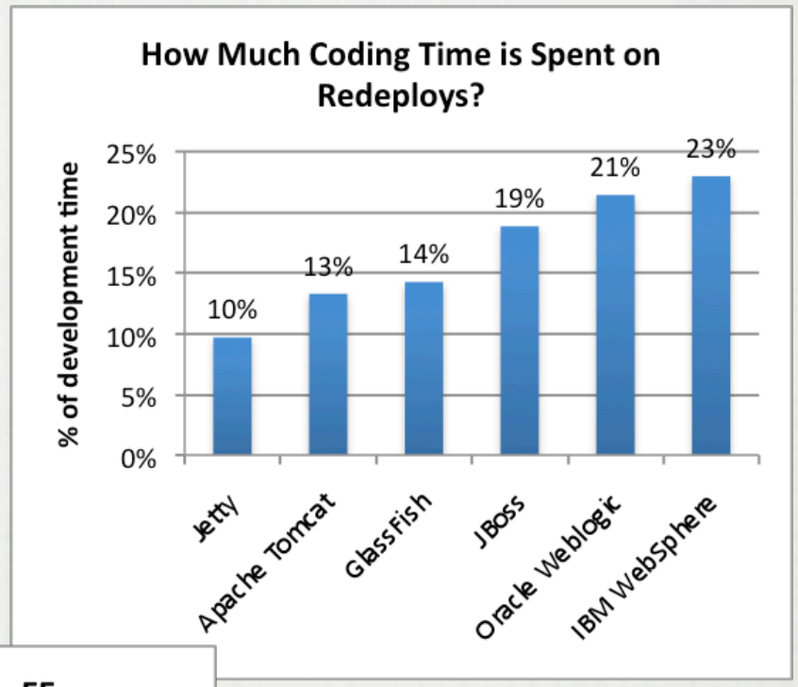
Best Practices: F.I.R.S.T

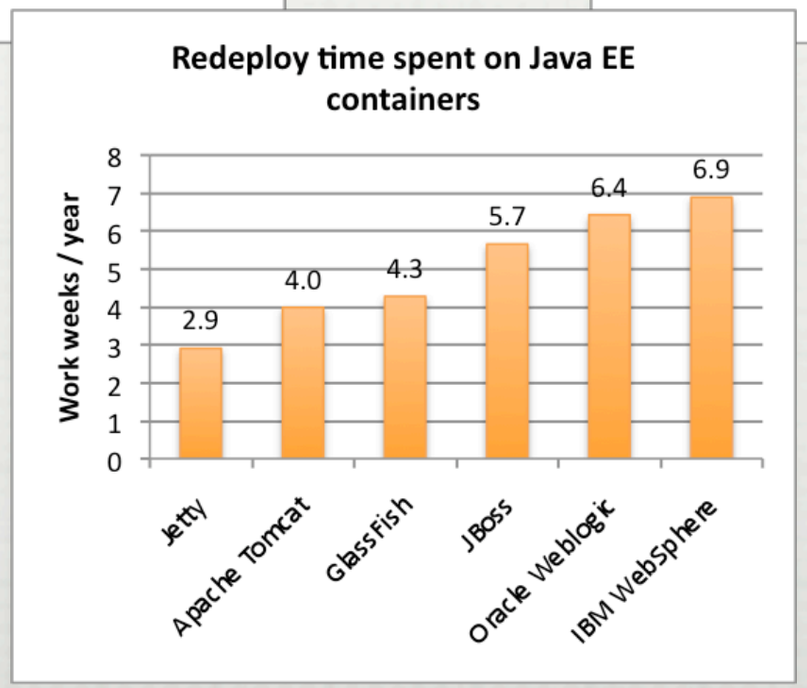
- ☐ Fast
- ☐ Independent
- ☐ Repeatable
- ☐ Self-validating
- ☐ Timely

From Object Mentor, www.objectmentor.com

Speed: Traditional Development







- 5 mins per coding hour becomes 6000 minutes annually (2.5, 40-hour workweeks)
- 9 mins per coding hour becomes 10800 minutes annually (4.5 weeks)
- 14 mins per coding hour becomes 16800 mins annually (7 weeks)

Survey by ZeroTurnaround
http://www.zeroturnaround.com/blog/java-ee-container-redeploy-restart-turnaround-report/

Speed Is Critical

- [ ] Developers Need Immediate Feedback
- [ ] Tests Must Run Fast
- [ ] Consider Using Jetty And HSQLDB For Automated Unit And Acceptance Testing
  - [ ] Depending On Your Framework, It Can Be Easy To Use Jetty/HSQLDB For Local Development And Automated Testing And Deploy To Websphere/Oracle In "Real" Environments
  - [ ] Make Sure You Do Continuous And Early Integration

Best Practices: Consistency

- ☐ Test Environments Should Be Scripted And Consistent
- ☐ Test Data Should Be Scripted And Consistent
- ☐ Database
  - ☐ In-memory Databases
    - ☐ Fast
    - ☐ Reliable
    - ☐ Easily Scriptable
    - ☐ HSQLDB
- ☐ Messaging
  - ☐ ActiveMQ Is A Good Alternative For Tests
  - ☐ Simple To Set Up

Best Practices: Test Naming and Structure

- ☐ The Name Of The Test Should Describe A Feature Or Specification
- ☐ The Name Should Clearly Describe The Purpose Of The Test
  - ☐ Frequently Read The Test Documentation
- ☐ Each Test Should Be For A Single Concept
- ☐ Strive For One Assertion Per Test
  - ☐ Or As Few As Possible
  - ☐ More Than One May Indicate More Than One Concept Is Being Tested
- ☐ What Does The Object Do, Not What Is The Object

Best Practices

- ☐ Only Specify What Is Important
- ☐ Even Seemingly Trivial Tests Are Important
- ☐ Treat Your Test Code Like Production Code
  - ☐ Maintain
  - ☐ Refactor
- ☐ Expressiveness And Readability Of Tests
- ☐ Check In To Source Control Frequently
- ☐ Consistent Test Structure
  - ☐ Setup/Execute/Verify/Teardown
  - ☐ Given/When/Then

Best Practices: Managing Test Data

- ☐ Objects
  - ☐ Avoid Mocking Values
  - ☐ Test Data Builders
- ☐ Persistent
  - ☐ In-memory Database
  - ☐ Scripted Data

Smells

- ☐ Doing The Opposite Of The Best Practices :-)
- ☐ Just Making It Work Isn't Good Enough
- ☐ Should Be Able To Run Tests Independently
- ☐ Tests Should Never Depend On Each Other

Tips: Eclipse Shortcuts Related To Testing

- ☐ Ctrl+1: Quick Fix
- ☐ Ctrl+Shift+X, T : Run As Unit Test (Use In Any Context: Method, Class, Package, Project, Etc.)
- ☐ F11: Launch Last Test
- ☐ F3: Open Declaration
- ☐ Alt+Shift+3: Refactoring Menu
- ☐ Alt+Shift+S: Code Completion/Generation/Format Menu
- ☐ Ctrl+e: Switch Between Open Windows
- ☐ Eclipse Static Import Favorites

These are Windows specific. On Mac you usually, but not always, substitute Command for Ctrl

Tips: Some Other Eclipse Shortcuts

- ☐ Ctrl-Space: Code Completion
- ☐ Ctrl-Shift-F: Format Code
- ☐ Ctrl-Shift-O: Organize Import
- ☐ Ctrl-Shift-T: Open Type
- ☐ Ctrl-Shift-R: Open Resource
- ☐ Ctrl-/: Toggle Comment
- ☐ Ctrl-Shift-W: Close All Open Windows
- ☐ Ctrl-N: New Wizard
- ☐ Ctrl-F7: Move Between Open Views
- ☐ Ctrl+Shift+L: Show Shortcuts
- ☐ Alt+left, Alt+right: Toggle Windows

TDD Challenges

- ☐ Discipline Is Required
- ☐ Developers Are Often Stubborn And Lazy
- ☐ It Is Hard For Developers To Drop Bad Habits
- ☐ Management Doesn't Often Understand "Internal Quality"

Some Resources Related to TDD

- ☐ Should Be On Every Professional Developers Bookshelf
  - ☐ Refactoring (Fowler)
  - ☐ Domain Driven Design (Evans)
  - ☐ Clean Code ("Uncle Bob" Martin)
- ☐ Recommended
  - ☐ Growing Object-Oriented Software, Guided By Tests (Freeman, Pryce)
  - ☐ Implementation Patterns (Beck)
- ☐ Haven't Read But, Given The Authors, Should Be Good
  - ☐ Test Driven Development: By Example (Beck)
    - ☐ The Book That "Rediscovered" TDD
  - ☐ XUnit Patterns (Meszaros)

By No Means An Exhaustive List But Should Keep You Busy For A While

Behavior Driven Development

- ☐ "TDD Done Right"
- ☐ Stories And Specifications
- ☐ Ubiquitous Language
- ☐ Given/When/Then
- ☐ Should/Ensure
- ☐ easyb (Groovy) Is Nice If You Are Working In The Java World