



Hands on Agile: Effective Software Development

Boise Code Camp

nTier Training, LLC

info@nTierTraining.com 

www.nTierTraining.com 



Objectives

- Understand the motivation for (almost) **EVERYTHING** we do in IT
- Remind ourselves how critical it is to **DO OO**
- Review / Primer on **TDD**
- Discuss purpose of **UML**
- Show the importance of integrating **OO, UML, TDD** in your Agile process
- Demonstrate some of the principles in a case study



Hands on Agile: Effective Software Development Motivation

Motivation

Object Oriented Programming
Test Driven Development
Unified Modeling Language
Agile Development
Case Study



Competitive Advantage

- As Thomas Friedman describes in his (must read) book, "The World is Flat," IT is about competitive advantage
- "The ROI for OO comes when you can:
 - *quickly respond to your customer's changes* and
 - *lower their total cost of ownership*, thereby giving them a
 - *competitive advantage* in the marketplace." Ed Lance - 2001
- *KEY: quickly respond to requirements changes*
- NOTE: Base hits win ball games
- So what's your TCO



Hands on Agile: Effective Software Development Object Oriented Programming

Motivation
Object Oriented Programming
Test Driven Development
Unified Modeling Language
Agile Development
Case Study



OO First Principles

Code to the interface, not the implementation

Find what varies and encapsulate it

Hide by convention, reveal by need

What you hide you can change

Favor composition over inheritance

Open-closed principle

The OOP Equation

$$\text{OOP} = \text{Principles} + \text{Practices}$$

OO Principles

- Coupling
- Cohesion
- Redundancy
- Encapsulation
- Testable

Patterns

- GoF
- J2EE patterns

OO Practices

- Intentionally couple
- Code to the interface
- Hide by convention
- Separate instantiation from implementation
- Favor composition
- Encapsulate variation
- Refactor regularly
- Code by intention
- Remove redundancy
- Practice testing



Hands on Agile: Effective Software Development Test-Driven Development

Motivation
Object Oriented Programming
Test-Driven Development
Unified Modeling Language
Agile Development
Case Study



Test-Driven Development - Broadly Defined

- ***Test-driven development*** (TDD) is a technique in which testing the production code is of primary importance
- Typically, the tests are written **before** the production code
- Code is not considered done until all tests pass
- All requirements are captured in tests
 - When the code passes the tests, you have satisfied the requirements
 - However, you may (**will**) still refactor your code to make it better

What Does TDD Really Mean?

- **Test-Oriented** Development
 - All design up front
 - You have lots of tests, but you don't always write them first
- **Test-Driven** Development
 - All design up front
 - You write tests before you write code
- Test-Driven **Design** (XP)
 - Test-first approach as a design technique – **no** design up front
 - The tests are a bonus, but the idea is to design as you go

What Does TDD Really Mean?

- Test-Driven **Development and Design** (nTier's philosophy)
 - Some initial design up front
 - Test-first approach to drive new code and changes
 - Design will evolve, because you learn from the tests, the tests can point out various code smells, etc.
 - It may also change to make the code more testable

- What does TDD mean at your organization?

- Test-driven development relies on an automated testing tool
 - Since tests run so often, need rapid response to small changes
- Testing framework needs to make writing tests easy
- Most Java developers use *JUnit*
- **Kent Beck** was one of the TDD pioneers, and created JUnit
 - He also wrote a book: *Test-Driven Development by Example*

■ Driving

- Driving is not about getting the car pointed in the right direction
- Driving is about constant adjustment...a little this way, a little that way, forever...you always pay attention...you are always prepared to change things

■ Pulling a bucket of water from a well

- When the bucket is small, a free-spinning crank is fine
- When the bucket is big and gets full, you get tired on the way up
- You need a ratchet mechanism that enables you to rest between bouts of cranking
 - The heavier the bucket, the closer the teeth need to be on the ratchet
- The tests in TDD are the teeth of the ratchet
 - The tougher the programming problem, the smaller the steps



Test-Driven Development Cycle - Overview

1. Add a test
2. Run all tests and see the new one fail **RED**
3. Write some code – **"make it work"**
4. Run all tests and see them all succeed **GREEN**
5. Refactor the code – **"make it right"**
Sometimes your refactoring breaks something **REFACTOR**
~~**REFACTOR**~~
6. Repeat
Sometimes you repeat only step 5

- Provides concrete evidence that the software works
 - *"If the bar is green, the code is clean"*
- Shortens the programming loop
 - Enables (and encourages) you to take small steps when coding
 - Provides rapid, fine-grained, concrete feedback (every few mins.)
 - If you break something, find out immediately
 - Encourages experimentation
- Helps you write cohesive code – small steps means small methods – that are **easily callable and testable**
 - By focusing on the tests first, you must imagine how the functionality will be used by clients (think like a client)
 - **Focus first on the interface**, then the implementation

- You are forced to think of the software in terms of **small units** that can be written and tested **independently**
 - This leads to smaller, more focused classes (cohesion), better coupling, and cleaner interfaces
- Your designs must consist of **highly cohesive, loosely coupled** components, just to make testing easier
 - Great news! This also makes evolution/maintenance of the system easier, e.g., when handling new/changed requirements
- Many TDD developers claim to have reduced their use of the debugger to zero (or close to it)
 - Debugging is the antithesis of automated testing – requires extensive manual inspection of variables, call stacks, etc.

Benefits of TDD

- Writing tests first implies **100% code coverage**

- When you have to write tests first for everything you do, you are far **less likely to add extraneous capabilities**
 - Helps to control developer-driven scope creep
 - *"Maximize the amount of work NOT done"*

- Code-based documentation
 - Tests automatically provide **sample client code**
 - Most programmers prefer to learn the basics of a class or operation from sample code than from written documentation
 - Tests not meant to provide complete documentation
 - But they do form an important part of it

- Taking small steps (get to green as quickly as possible) points out redundancies and other code smells
 - This allows you to have a clear initial direction as to what to refactor
- Shows you ways that your **design should evolve**
 - You see too much coupling in the client
 - You see better APIs than originally designed
 - You see things in the code that indicate when a design pattern(s) might be appropriate
 - Remember Beck: constant adjustment...a little this way, a little that way...you always pay attention...you are always prepared to change things



Hands on Agile: Effective Software Development UML

Motivation
Object Oriented Programming
Test Driven Development
Unified Modeling Language
Agile Development
Case Study



Unified Modeling Language

- Agile requires good UML

- Think in pictures

- You must communicate quickly

- Good UML is about communication
 - Every UML diagram is a promise of a conversation
 - You almost never draw a perfect picture
 - GIVE UP HOPE OF PREDICTING REQUIREMENTS

- Who needs to know UML



Hands on Agile: Effective Software Development Agile Development

Motivation
Object Oriented Programming
Test Driven Development
Unified Modeling Language
Agile Development
Case Study



Process Control - Defined vs. Empirical

- It is typical to adopt a plan-driven, well-defined approach when the underlying mechanisms are well understood
- What happens if we think they are well understood and later it turns out that we did not understand them at the outset at all?
- What happens when requirements start changing on us?
 - "Hey, that wasn't in the plan..."



Origins of Agile Software Development

- Agile developed as a reaction to process-driven methodology
 - For many, the appeal is the rebuttal to the bureaucracy involved
- Compromise between no process and too much process
 - Just enough process to gain a reasonable payoff
- Significant differences in emphasis from engineering methods
 - **Less document-oriented**
 - In many ways, rather code-oriented



Characteristics of Agile Software Development

- **Adaptive** rather than **predictive**
 - Regular adaptation to changing circumstances

- Two common denominators:
 - **Time-boxed**
 - **Belief that all requirements can't be captured up front**

- **People-oriented** rather than **process-oriented**
 - Close, daily cooperation between business people and developers
 - Face-to-face conversation is best
 - Projects built around motivated individuals – who should be trusted
 - Self-organizing teams

Project Vision Is the Driver

- Waterfall approach
 - Estimate cost and schedule
 - Fix "features"
- Agile approach
 - Fix costs and schedule
 - Estimate "features"
- Plan-driven
- Vision-driven

The **plan** creates costs/schedule estimates

The **vision** creates feature estimates

The Agile Manifesto

- In 2001, a group got together in Snowbird, Utah to discuss the growing field of what were then called "lightweight methods"
- The term *agile* was used for this new breed of methods
- They wrote the *Manifesto for Agile Software Development*, setting out the values and principles of these agile processes
- <http://agilemanifesto.org>



Agile Manifesto Mission Statement

- We are uncovering better ways of developing software by doing it and helping others to do it

- Through this work we have come to value:
 - 1. Individuals and interactions** over processes and tools
 - 2. Working software** over comprehensive documentation
 - 3. Customer collaboration** over contract negotiation
 - 4. Responding to change** over following a plan

- While there is value in the items on the right, we value more the items on the left (the ones in bold)

<http://agilemanifesto.org>



Agile Manifesto Principles

- Highest priority is to satisfy customer through **early and continuous delivery** of valuable software
- **Welcome changing requirements**, even late in development
 - Agile processes harness change for the customer's competitive advantage
- **Deliver working software frequently**, from weeks to months, with preference to shorter timescales

<http://agilemanifesto.org>

Agile Manifesto Principles

- Working software is the primary measure of progress
- Promote **sustainable development**
 - Sponsors, developers, and users should be able to maintain a constant pace indefinitely
- Continuous attention to **technical excellence and good design**
- Simplicity, the art of **maximizing the amount of work NOT done**

<http://agilemanifesto.org>

Communication and Sharing The Risk

- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**
- Business people and developers must **work together daily** throughout the project



By Clark & Vizdos

© 2007 implementingscrum.com

Even Journalism Is Becoming Agile

- Blog-style journalism is contributing to the decline of the printed newspaper
 - It's agile journalism!
- Old school paper journalism:
 - Assess need/desire for story
 - Hit the streets, research it
 - Write it up
 - Submit to editor
 - Revise
 - Print
 - Story is "done" (in the past)
- Blog-style online journalism:
 - Hit streets, research story
 - Write it up and push it out
 - Faster time to market
 - Reader comments come in
 - Adding relevant info
 - Refuting facts or evidence
 - Comments on comments
 - Learn and adjust as you go
 - Story is ongoing, "alive"



- Agile requires STRONG OO
 - Let's say you identify 50 Use Cases
 - Turn the first 5 into Use Case scenarios
 - Build a class diagram and other UML as needed (e.g. sequence diagrams etc.)

- Assume we successfully complete everything we've designed in the first iteration

- Second iteration – 5 more Use Cases

- Ummm...what happens to my perfect OO design laid out in my class diagram in the first iteration



Integrating Agile

- Agile requires UML for communication
- Just enough and no more
- A promise of a conversation
- Train your entire team



Integrating Agile

- Agile requires adapting
- You can't do it like XYZ company's successful project
- Don't become a process zealot
- Which Agile is best for you?
- Parable of the Yacht salesman

A Few Final Points

- Agile development is **not**:
 - Sloppy
 - "Cowboy coding"
 - "Winging it"
 - Undocumented chaos

- Agile can fail
 - **U**nderstanding, **A**cceptance, **S**upport (UAS)
 - Right vocabulary, wrong practices
 - Change it 1-2 variables at a time

- Which form of Agile is best for me?
 - Remember how to sell yachts?



**Hands on Agile:
Effective Software Development
Case Study - Boot Camp Course**

Motivation
Agile Development
Test-Driven Development
Case Study



Case Study - Boot Camp

- Teach TDD

- Confirm a good OO foundation

- Confirm UML core

- Explain Agile and train students on one Agile methodology

- Learn by doing – 80% of the Boot Camp is hands on
 - Use a socratic teaching method
 - DTCR
 - Review code live

Erich Gamma on Learning OO

- I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior...
- You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general...

<http://www.artima.com/lejava/articles/gammadpP.html>

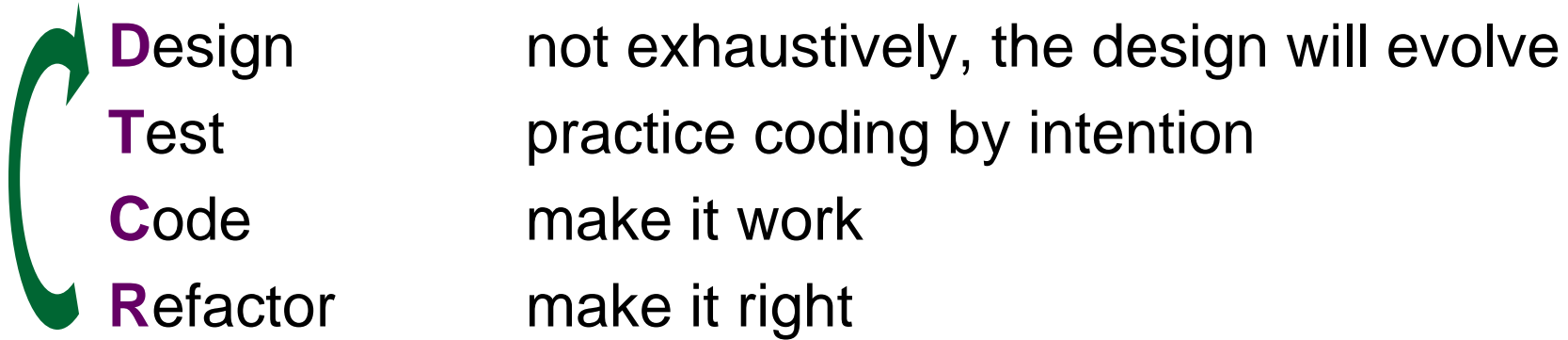


Our Approach in the Boot Camp Project

- Socratic method
 - We don't teach you a pattern and give you a lab
 - We want you to get your head around the problem
 - You should feel a little pain first
- We will use an agile development technique known as **test-driven development (TDD)**
 - This involves writing the tests before writing the code
 - But there's more to it than that
- Coding will be done in **short iterative cycles**
 - Forces you to get good code written faster (it really works)
 - Also serves to measure your progress
 - Likely requires an adjustment to how you normally do things
- You will make mistakes (yeah, we will too)

TDD+Agile as a Coding Practice

- You've been working with TDD
- You've heard about agile development
- How can you put them together in practice when coding?
- The **D-T-C-R** code cycle **Do The Code Right**



- This is a **15-20 minute** time box, at the end of which:
 - You should have something working
 - *"Delivery working software frequently"*

Improving Your Estimates

- You will estimate how many cycles you need to do each lab
 - You are not expected to know this number exactly
 - *"Agile is adaptive rather than predictive"*

- We will measure your progress after each cycle
 - *"Working software is the primary measure of progress"*
 - You can revise your estimate after each cycle

- The customer (instructors) is in the room
 - *"Close cooperation between business people and developers"*

- The project should be moving forward at all times
 - *"Promote sustainable development"*

blue page



Boot Camp Project - Just in Time Shipping (JITS)

- JITS is software for a mail kiosk
- First models will be in lobbies of post offices for use during busy times or when the main part of the post office is closed
- You can mail boxes or letters
- We'll be "discovering" new requirements as we go

Erich Gamma on Teaching Design Patterns

- The other question was how we should teach patterns. Not that I know exactly what you should do, but I think what you should *not* do is have a class and just enumerate the 23 patterns. This approach just doesn't bring anything.
- You have to feel the pain of a design which has some problem. I guess you only appreciate a pattern once you have felt this design pain.
- It's an eye opener to realize that oh, actually this pattern, factory or strategy, is a solution to my problem. And I think that's the really interesting way to teach.

<http://www.artima.com/lejava/articles/gammadpP.html>



Erich Gamma on Teaching Design Patterns

- When I started teaching it was really boring, because I was just enumerating the patterns. I found it far more interesting to try to motivate with real examples how to apply patterns.
- In other words, you really need to present the problem in a realistic context – synthetic examples do not fly.
- This is definitely the way I'd recommend that people use patterns. Do not start immediately throwing patterns into a design, but use them as you go and understand more of the problem.
- Because of this I really like to use patterns after the fact, refactoring to patterns.

<http://www.artima.com/lejava/articles/gammadpP.html>



Section Review

- Agile development is about adapting and integrating key resources

- The BEST Agile
 - Is the one you customize
 - Leverages OO, Patterns and Principles
 - Utilizes UML for communication
 - Incorporates Test Driven Development
 - Focuses on delivery of working software above all else
 - Doesn't lose sight of ROI

- TDD + Agile = D T C R
 - Success is daily – but it's also every 20 minutes

- OO principles and design patterns are best taught in context



End

- This slide is intentionally devoid of any useful content