Lecture 23: Agile Development and Extreme Programming

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2003

Credit where Credit is Due

- The material for this lecture is based on content from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin
- As such, some of this material is copyright © Prentice Hall, 2003

Goals for this lecture

- (Very) Briefly introduce the concepts of Agile Design and Extreme Programming
- Agile Design is a design framework
- Extreme Programming is one way to “implement” agile design
  - Other agile life cycles include SCRUM, Crystal, feature-driven development, and adaptive software development
  - See <http://www.agilealliance.org/> for pointers

Agile Development (I)

- Agile development is a response to the problems of traditional “heavyweight” software development processes
  - too many artifacts
  - too much documentation
  - inflexible plans
  - late, over budget, and buggy software

Agile Development (II)

- A manifesto (from the Agile Alliance)
  - "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value
    - individuals and interactions over processes and tools
    - working software over comprehensive documentation
    - customer collaboration over contract negotiation
    - responding to change over following a plan
  - That is, while there is value in the items on the right, we value the items on the left more

Agile Development (III)

- From this statement of values, agile development has identified twelve principles that distinguish agile practices from traditional software life cycles
- Lets look at five of them
  - Deliver Early and Often to Satisfy Customer
  - Welcome Changing Requirements
  - Face to Face Communication is Best
  - Measure Progress against Working Software
  - Simplicity is Essential

Deliver Early and Often to Satisfy Customer

- MIT Sloan Management Review published an analysis of software development practices in 2001
  - Strong correlation between quality of software system and the early delivery of a partially functioning system
    - the less functional the initial delivery the higher the quality of the final delivery!
  - Strong correlation between final quality of software system and frequent deliveries of increasing functionality
    - the more frequent the deliveries, the higher the final quality!
- Customers may choose to put initial/intermediate systems into production use; or they may simply review functionality and provide feedback

Welcome Changing Requirements

- Welcome change, even late in the project!
- Statement of Attitude
  - Developers in agile projects are not afraid of change; changes are good since it means our understanding of the target domain has increased
  - Plus, agile development practices (such as refactoring) produce systems that are flexible and thus easy to change

Face to Face Communication is Best

- In an agile project, people talk to each other!
  - The primary mode of communication is conversation
    - there is no attempt to capture all project information in writing
  - artifacts are still created but only if there is an immediate and significant need that they satisfy
    - they may be discarded, after the need has passed

Measure Progress against Working Software

- Agile projects measure progress by the amount of software that is currently meeting customer needs
  - They are 30% done when 30% of required functionality is working AND deployed
- Progress is not measured in terms of phases or creating documents

Simplicity is Essential

- This refers to the art of maximizing the amount of work NOT done
  - Agile projects always take the simplest path consistent with their current goals
  - They do not try to anticipate tomorrow's problems; they only solve today's problems
  - High-quality work today should provide a simple and flexible system that will be easy to change tomorrow if the need arises

Extreme Programming

- Extreme Programming (XP) takes commonsense software engineering principles and practices to extreme levels
  - For instance
    - "Testing is good?"
    - then
    - "We will test every day" and "We will write test cases before we code"
- As Kent Beck says extreme programming takes certain practices and "sets them at 11 (on a scale of 1 to 10)"

XP Practices

- The best way to describe XP is by looking at some of its practices
  - There are fourteen standard practices, we'll look at six important ones
    - Customer Team Member
    - User Stories
    - Pair Programming
    - Test-Driven Development
    - Collective Ownership
    - Continuous Integration

Customer Team Member

- The "customer" is made a member of the development team
  - A customer representative should be "in the same room" or at most 100 feet away from the developers
  - "Release early; Release Often" delivers a working system to the client organization; in between, the customer representative provides continuous feedback to the developers

User Stories (I)

- We need to have requirements
- XP requirements come in the form of "user stories" or scenarios
  - We need just enough detail to estimate how long it might take to support this story
    - avoid too much detail, since the requirement will most likely change; start at a high level, deliver working functionality and iterate based on explicit feedback

User Stories (II)

- User stories are not documented in detail
  - we work out the scenario with the customer "face-to-face"; we give this scenario a name
  - the name is written on an index card
    - developers then write an estimate on the card based on the detail they got during their conversation with the customer
- The index card becomes a "token" which is then used to drive the implementation of a requirement based on its priority and estimated cost

Pair Programming

- All production code is written by pairs of programmers working together at the same workstation
  - One member drives the keyboard and writes code and test cases; the second watches the code, looking for errors and possible improvements
  - The roles will switch between the two frequently
  - Pair membership changes once per day; so that each programmer works in two pairs each day
    - this facilitates distribution of knowledge about the state of the code throughout the entire team
- Studies indicate that pair programming does not impact efficiency of the team, yet it significantly reduces the defect rate!
  - [Laurie Williams, 2000] [Alistair Cockburn, 2001] [J. Nosek, 1998]

Test-Driven Development

- All production code is written in order to make failing test cases pass
  - First, we write a test case that fails since the required functionality has not yet been implemented
  - Then, we write the code that makes that test case pass
  - Iteration between writing tests and writing code is very short; on the order of minutes
- As a result, a very complete set of test cases is written for the system; not developed after the fact

Collective Ownership

- A pair has the right to check out ANY module and improve it
  - Developers are never individually responsible for a particular module or technology
- Contrast this with Fred Brook's conceptual integrity and the need for a small set of "minds" controlling a system's design
  - Apparent contradiction is resolved when you note that XP is designed for use by small programming teams; I haven't seen work that tries to scale XP to situations that require 100s or 1000s of developers

Continuous Integration

- Developers check in code and integrate it into the larger system several times a day
- Simple Rule: first one to check-in "wins"; everyone else merges
- Entire system is built every day; if the final result of a system is a CD, a CD is burned every day; if the final result is a web site, they deploy the web site on a test server, etc.
  - This avoids the problem of cutting integration testing to "save time and money"