Lecture Overview

- Agile methods

Agile Methodologies

- Agility can be defined as the ability to create change and respond to change
- This is very relevant in the software development business as it is the nature of this business that change is central to everything that happens
- Change is driven by innovation, technical advancement, new business strategies or simply by a demanding community who expect the latest and best of breed solutions

Agile software processes

- Increase responsiveness of software teams
  - Changing requirements
  - Strong customer involvement
- Reduce overhead to speed up development
  - Focus on people
  - Focus on delivering software
- Individuals and Interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The “Code and Fix” Model

“Fire, fire, fire, fire, aim again, fire, fire, fire--there is no time for ‘ready’.”

-- Gary Hamel, Leading the Revolution

Heavyweight, Disciplined Process

Ready, ready, ready, ready, aim, aim, aim... fire.

Adaptive, Lightweight (Agile) Process

“Get it right the last time.”

Ready, aim, fire, ready, aim, fire, ... fire.
Common Characteristics of Agile Methodologies

- Adaptive to requirements changes...embracing change
- Customers just don’t know
- Developers just don’t understand
- Iterative/Exploratory
- Process itself adapts to project...disciplined, yet flexible
- Rely on collaboration/innovation through group interaction
- Simplicity...“Just do it.”
- Minimal documentation (efficiency, low inertia)

Human Nature

Undisciplined, Code-and-Fix

Adaptive, Lightweight

Disciplined, Heavyweight

Agile Alliance

- In early 2001, the Agile Alliance was formed by a group of similar thinking industry experts
- They have stated their values in the form of a manifesto
- http://www.agilealliance.org

The Agile Manifesto - I

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

The Agile Manifesto – II

- Our highest priority is to satisfy the 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 a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.

The Agile Manifesto – III

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
The Agile Manifesto – IV

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Various Agile Methods Available

- We will see some of these in next lecture
  - Adaptive Software Development (ASD)
  - Agile Modeling
  - Crystal methods
  - Dynamic System Development Methodology (DSDM)
  - eXtreme Programming (XP)
  - Feature Driven Development
  - Lean Development
  - Scrum

Key approaches & players

- Extreme programming: Kent Beck, Ron Jeffries, Martin Fowler
- Scrum: Ken Schwaber, Mike Beedle, Jeff Sutherland
- DSDM: Arie van Bennekum, Jennifer Stapleton
- Crystal methods, Agile Software Development: Alistair Cockburn
- Adaptive Software Development, Agile Software Development Ecosystems: Jim Highsmith
- Feature driven development: Jeff De Luca, Peter Coad, Jon Kern
- Lean Development: Bob Charette

Data on Agile Methodologies

- Evidence is now appearing around the increase in usage and acceptance of agile
  - This chart is the result of a survey carried out by the Cutter Consortium which involved 200 IT Managers throughout the world
  - 54% used their own ‘in-house’ developed methodology that could be described as agile
  - For those not using in-house methodologies spread of usage where it is clear that XP is the most common approach

Research

- Some books...
  - Addison Wesley – XP series
  - Agile software development series
  - Conferences
    - www.xpuniverse.com
- Some people...
  - Don Wells: http://www.extremeprogramming.org
  - Martin Fowler: http://www.martinfowler.com
  - Ken Schwaber: http://www.controlchaos.com
  - Alistair Cockburn: http://crystalmethodologies.org
  - Ron Jeffries: http://www.xprogramming.org
  - Scott Ambler: http://www.extreme-modeling.com

Agile methods

- Some basic facts
  - Producing code is required to deliver a system
  - Money spent on analysis and design are wasted if the system is never used
  - Business requirements have to be the drivers for software development
  - Requirements change
**Extreme Programming**

*(Beck, Fowler, Jeffries, …)*

From Beck, Page 70

**Scrum**

www.controlchaos.com

- Focus on management issues
- Product backlog stable over Scrum timeframe
- Short iterations (20 days)

**Crystal Methods**

- "Software development is a cooperative game of invention and communication."
  - Alistair Cockburn
- People and communication-centric
- Adapt your methodology to your project's need
- Use the least disciplined methodology that will still succeed
- Two base techniques
  - Methodology-tuning technique
  - Reflection workshop technique ↔ retrospective
- Incremental development
  - 4 months or less
- Clear ownership model of work products
- Have regression testing framework, and do some kind of peer review

**Crystal Structure**

- Two ways to differentiate projects
  - Number of people in the project
  - Consequence of errors

<table>
<thead>
<tr>
<th>Life (L)</th>
<th>Essential Money (E)</th>
<th>Discretionary Money (D)</th>
<th>Comfort (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comfort (C)</td>
<td>Essential Money (E)</td>
<td>Discretionary Money (D)</td>
<td>Life (L)</td>
</tr>
<tr>
<td>Clear</td>
<td>Yellow</td>
<td>Orange</td>
<td>Red</td>
</tr>
<tr>
<td>3-8</td>
<td>10-20</td>
<td>25-50</td>
<td>50-100</td>
</tr>
</tbody>
</table>

**Adaptive Software Development**

*(James Highsmith)*

- Short iterations
- Deliverable-centric instead of task-centric
- “Controlling the edge of chaos”
**Feature-Driven Development**  
(Coad, Lefevbre, Deluca)

- Deliver frequent, tangible, working results that are "useful in the eyes of the client"
- A feature defines a task
- Group features into business-related sets
- Focus on delivering results every two weeks
- Track and report progress by feature progress

**The five FDD processes**

- Develop an overall model
- Build a detailed, prioritized features list
- Plan by feature
- Design by feature
- Build by feature

**FDD Process**

![FDD Process Diagram]

**Agile Modeling**  
(Ambler, "Rational")

- "The scope of AM is just modeling, not the full development process"  
  - Scott Ambler
- AM should be fitted into a full lifecycle methodology
- Agile Modeling Principles
  - Software Is Your Primary Goal
  - Embrace Change
  - Enabling Your Next Effort Is Your Secondary Goal
  - Model With A Purpose
  - Maximize Stakeholder Investment
  - Incremental Change
  - Quality Work
  - Rapid Feedback
  - Travel Light

**Models**

- A framework for RAD plus some guidance on how to apply it
  - Methods are kept at a high level
  - Can be tailored for any environment.
- There are no prescribed techniques.
- Philosophy behind DSDM
  - Development is team effort, and must combine
    - users knowledge of business.
    - and technical skills of IT professionals.
  - High quality demands fitness for purpose as well as technical robustness.
  - Development can be incremental.
  - The law of diminishing returns applies.
DSDM Overview

- The whole method is based on nine principles
- The outline of these principles include:
  - Active user involvement
  - DSDM teams must be empowered to make decisions.
  - The focus is on frequent delivery of products.
  - Fitness of business purpose is the essential criterion.
  - Iterative and incremental development is necessary.
  - All changes are reversible.
  - Requirements are baselined at a high level.
  - Testing is integrated throughout the lifecycle.
  - A collaborative and cooperative approach between stakeholders is essential.

Lecture Overview

- Extreme programming

Fundamentals of XP

- Distinguish between decisions made by business stakeholders and developers
- Simplistic - keep design as simple as possible “design for today not for tomorrow”
- Write automated test code before writing production code and keep all tests running
- Pair programming
- Very short iterations with fast delivery

Extreme programming

- Set of SE practices that produce high-quality software with limited effort
- Light-weight software development process
  - Replaces documentation with communication
  - Focuses on source code and testing
  - Controversial - “Hacking”?
  - Strong productivity improvements
- Developed by industry practitioners
  - “proven at cost conscious companies like Bayerische Landesbank, Credit Swiss Life, DaimlerChrysler, First Union National Bank, Ford Motor Company and UBS.” XP Web Site
  - http://www.extremeprogramming.org

Why is XP controversial?

- No specialists
  - Every programmer participates in architecture, design, test, integration
- No up-front detailed analysis and design
- No up-front development of infrastructure
- Not much writing of design & implementation documentation beside tests and code
When can XP be used?

- Small projects:
  - 5-10 developers, maybe 20
- Developer and customer representative are co-located
- Problems:
  - Point-and-go culture
  - Testing takes hours to execute

XP Principles – I

- Philosophy:
  - "Take known good practices and push them to extremes"
  - "If code reviews are good, we'll review code all the time"
  - "If testing is good, we'll test all the time"
  - "If design is good, we'll make it part of everybody's daily business"

XP Principles – II

- "If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality"
- "If architecture is important, everybody will work defining and refining the architecture all the time"
- "If integration testing is important, then we'll integrate and test several times a day"
- "If short iterations are good, we'll make the iterations really, really short – seconds and minutes and hours, not weeks and months and years"
- "If customer involvement is good, we'll make them full-time participants"

XP: The 12 Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Testing
- Refactoring

- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-hour Week
- On-site Customer
- Coding Standards

The XP Practices

The Planning Game

- Use stories to facilitate knowledge transfer
- Put decisions in the hands of the person with the best knowledge:
  - business decisions — Customer
  - software decisions — Developer
- Plan only as far as your knowledge allows
  - next iteration or next release

Small Releases

- Supports quick feedback from users
- Simplify the tracking of metrics
  - stories per iteration — project velocity
- Increase the manageability of the project for the customer
  - But complicate user conservation of familiarity
The XP Practices
Metaphor
- Ground all discussions in a single shared story of how the whole system works
- Provide an overarching view of the project
- Connect program to work process

The XP Practices
Simple Design
- Design Embodies only the needed complexity and no more
  - emphasis on top-down or bottom-up design as needed to meet this iteration's stories
  - extra complexity removed when discovered
- Simpler designs are easier to modify, maintain, and describe
  - decreases the cost of design changes
  - But no notion of product line architecture

The XP Practices
Testing
- Unit tests verify the programmer's work
  - must be done by programmer
  - constant testing makes finding new bugs faster and easier
- Functional tests verify that a story is complete
  - developed by customer
  - tests define functional requirements

The XP Practices
Refactoring
- Procedure for implementing iterative design
  - behavior-preserving
  - improves communication among developers
  - adds flexibility to the programming process
- Design is important – do it all the time
  - software development process is a design process
  - But redesign much more expensive for large systems

The XP Practices
Pair Programming
- All code is written by two programmers at a single machine
- Inspections are important, so do them all the time
- Increase implicit knowledge transfer
- Decrease cycle time, but increase effort

The XP Practices
Collective Ownership
- Code to belongs to the project, not to an individual engineer
  - anyone can change any code anywhere
  - no personal ownership of modules
  - no egoless programming either
- Everyone is permitted access to all the code so everyone has a stake in knowing all of the code
- As engineers develop required functionality, they may browse into and modify any class.
- They are responsible for keeping all the Unit Tests running (and writing new ones for new functionality).
  - “You break it, you fix it.”
- Requires deserved trust
  - But still has scalability problems
The XP Practices
Continuous Integration

- The system always works
  - there is always something to be released
- Similar to rapid releases
  - fast feedback to developers on problems
  - no 'big bang' integration disasters
  - Pair writes up unit test cases for User Story
  - Pair codes up User Story
  - Pair unit tests code to 100%
  - Pair integrates
  - Pair runs ALL unit test cases to 100%
  - Pair moves on to next User Story with clean slate and clear mind
  - Should happen often!
  - Prevents Integration Hell

The XP Practices
40-hour Week

- No heroes
- Knowledge can only be transferred at a limited rate
- Work for sustained speed, not a single sprint
  - never work overtime a second week in a row
- Kent Beck says, "...fresh and eager every morning, and tired and satisfied every night..."
- Burning the midnight oil kills performance
  - Tired developers make more mistakes, which slows you down more in the long run
- If you mess with people's personal lives (by taking it over), in the long run the project will pay the consequences

The XP Practices
On-site Customer

- A real, live user available full-time to answer questions as they occur
- Programmers don't know everything
- Business knowledge is the key to a successful business project

The XP Practices
Coding Standards

- Communication occurs through the code
- Common standard promotes understanding of other developers' code
- Helps promote team focus

The XP Practices
How everything fits together

From Beck: XP, Page 70

XP in more detail
What is refactoring?

- Changing the code without changing its functionality
- Goal: make code easier to maintain
- Refactoring book by Fowler
- http://www.refactoring.com

Why refactor?

- To improve the design of software
- To make code easier to understand
- To help find bugs
- As a result
  - Coding becomes faster
  - Outside source code documentation less required

When do you refactor?

- When the code “smells bad”
  - Repeating code
  - Code difficult to understand
  - Long methods
  - Long message chains
  - Switch statements instead of polymorphism
  - ...

How do you refactor?

1. If code smells bad
   - 2. make a change
   - 3. test the system.
   - 4. if it works
     - goto (1)
     - else goto (2)
   - Refactoring is heavily dependent upon
     - automated test drivers and
     - working code.

Refactor Mercilessly

- Improve the design of existing code without changing functionality
  - Simplify code
  - Opportunity for abstraction
  - Remove duplicate code
  - Relies on testing to ensure nothing breaks in the process of refactoring.

The Planning Game

- Recognition
  - You can’t know everything when you start
    - What a realization!
  - Customers will learn about what they want by looking at what you’ve done so far
  - Developers will learn about the domain and technology as the project progresses.
  - Developers will learn what questions to ask once they start
    - Early on, they don’t know what they don’t know
**User Story**

- Written on a card
- “chunk of functionality that is coherent in some way to the customer”
- Example
  - Employees who are sick more than 3 days go on DAP (Disability Absence Plan). They are paid from their full pay for 390 working days, and then 70% pay up through 270 days. DAP euros paid must be kept separate from regular pay euros, for accounting purposes.
- Given priority/value by the customers based on business value
- Given estimate (1, 2, 3 weeks) by the developers
- Given risk value by developers
- Doing risky items first mitigates risk
- The card is a commitment to talk more later.

**The Planning Game**

- Pieces: User Stories (prioritized/valued and estimated)
- Goal
  - Put greatest possible value of stories into production over the life of the game
- Story Driven Commitment: Calculate release date
- Date Driven Commitment: Pick stories
- Ignore dependencies between stories

**On-Site Customer**

- Customer available on site to clarify stories and to make critical business decisions.
- Developers don’t make assumptions
- Developers don’t have to wait for decisions
- Face to face communication minimizes the chances of misunderstanding
- Remember, the original user story was a commitment for a later conversation . . .

**Testing**

- Unit Testing
- Acceptance Testing

**Unit Tests**

- Code Unit Test First
- When developers go to release new code, they run all the unit tests, not just theirs, on the integration machine.
- The tests must run at 100%.
- If any test fails, they figure out why and fix the problem.
- The problem certainly resides in something they did ... since they know the tests ran at 100% the last time anything was released.

**Acceptance Tests**

- Test cases “extracted from” customer
- Test system end-to-end
- Tells the customer and the developers if the system has the features it is supposed to have
- Don’t have to run at 100%
- Progress used to measure “Project Velocity”
  - What % of the customer’s acceptance test cases run?
Automating software testing

- Manual software testing is time consuming
- Software testing has to be repeated after every change (regression testing)
- Write test drivers that can run automatically and produce a test report
- Junit testing
  - A small testing framework written in Java.
  - A series of extensible classes that do much of the testing grunt work for us. (i.e. counting and reporting errors and failed tests, running tests in batch, etc.)
  - Very handy for Extreme Testing...
  - Developed by Kent Beck and Erich Gamma

Pair Programming I

- Coding is traditionally split amongst the development team with each taking a particular function or specification and generating the code and unit tests
- In XP all code is built by two programmers, sitting side by side, at the same machine
- This practice ensures that all production code is reviewed by at least one other programmer, and results in better design, better testing, and better code
- Knowledge transfer amongst the team is also a major advantage when using pair programming
  - Both programmers in the pair are familiar with the code and have either written the code or has been actively involved as the programmer watching the code generation.

Pair Programming II

- All production software in XP is built by two programmers, sitting side by side, at the same machine
- Inspection occurs in real time, while the cost of change is low.
- 100% of the production code is reviewed by at least one other programmer
- Results in resume quality code that developers are proud of
  - better design, better testing, and better code
- Most formal inspections, if they occur at all, occur after the code is completed when the cost of change is much higher (2 - 3 hours per defect).
- Remedial measures to correct defects and design flaws detected during system test or following release are typically very high (15 hours or greater).
- Development costs do not double
- Following learning curves to adapt to the paired programming environment (1 - 2 iterations) development time increase averages 15%

Pair Programming III

- Specialization leads to queues (you will block on the specialist at some point)
- the ‘Driver’ implements, focusing on the tactical.
- The navigator is more objective (strategic) – asks ‘why’ and provides explanations
- Healthy pairings have communications every 45 to 60 seconds.
- Pairs should switch roles often
- Pairs can break off for low complexity tasks
- Pairing is intense (suggest a break every 2 hours)

XP Design

- The Source Code is the Design
- The main thing is that documentation gets out of synch very quickly - so why bother putting a lot into it?
  - “Documentation is not understanding.”
  - “This form of combined design/development goes much faster than the longer cycle that is an inevitable part of paper design.”
  - “We discover good design by implementing the system.”
  - “We let the code tell us how the objects really want to interact.”
  - Many do some quick speculative design

You Aren’t Going to Need It

- You get an idea for a feature in ATaskClass
  - You don’t need it right now, but “Some day we’re going to need it...”
- At this moment you have a choice
  - Continue working on what you signed up to do, or begin working on something you didn’t sign up to do
- Tell yourself You Aren’t Going to Need It
  - Set aside your thoughts and fears about tomorrow and get back to work on today
  - Without a clear use for the feature, you don’t know enough about what is really needed. Spending time on it is speculative at best.
**Simplicity**

- Do The Simplest Thing That Could Possibly Work
- Implement a new capability in the simplest way you can think of that "could possibly work".
- Don’t build a lot of amazing superstructure, don’t do anything fancy...
- ... just put it in.

**Some limitations of XP**

- Co-location of team members required
- Scalability of the process: Small teams ⇒ small projects
- Distributed XP
  - Can we transfer the productivity benefits of XP to a distributed environment?
  - Communication replaces documentation
  - Open-source projects
- Virtual teams
  - Software development is more and more distributed
  - Size of projects
  - Scarse local resources
  - Outsourcing of tasks is often financially beneficial
  - Open-source success stories
- Extreme Programming Considered Harmful for Reliable Software Development