Software Engineering
Prof. Dr. Bertrand Meyer
March 2007 – June 2007

Software development: “Process” vs “agile”
with material by Marco Piccioni

Three cultures of software development

- Process
- Agile
- Object

The first two are usually seen as exclusive, but all have major contributions to make

Process-oriented

(Sometimes called formal)
Examples:
- Waterfall model (from 1970 on)
- Military standards
- CMM, then CMMI
- ISO 9000 series of standards
- Rational Unified Process (RUP)
- Cluster model

Overall idea: to enforce a strong engineering discipline on the software development process
- Controllability, manageability
- Traceability
- Reproducibility

Agile

Extreme Programming (XP)
Lean Programming
Test-Driven Development (TDD)
Scrum

This lecture (today and tomorrow)

- 1. The case for agile methods
- 2. Process-oriented methods
- 3. Towards a combination

The case for agile methods
(or: the Cubicle Strikes Back)
"The agile manifesto"

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.

Scheme 1: predictable manufacturing

Assembly-line production is possible:

- Define specifications and constructions steps
- Build some instances and perform measurements
- On the basis of that experience, estimate & schedule future production

Scheme 2: new model development

Each model specific, evolving process:

- Requirements change between races
- Static reasons (specific tracks)
- Dynamic reasons (weather, competitors)
- High level of competition
- Continuous experimenting

Prototypes rather than products

Assembly-line vs prototype

<table>
<thead>
<tr>
<th>Assembly-line manufacturing</th>
<th>Prototype-style manufacturing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify, then build</td>
<td>Hard to freeze specifications</td>
</tr>
<tr>
<td>Reliable effort and cost estimates are possible, early on</td>
<td>Estimates only become possible late, as empirical data emerge</td>
</tr>
<tr>
<td>Con identify schedule and order all activities</td>
<td>Activities emerge as part of the process</td>
</tr>
<tr>
<td>Stable environment</td>
<td>Many parameters change; need creative adaptation to change</td>
</tr>
</tbody>
</table>

What about software?

In the agile view, most software development is not a predictable, mass-manufacturing problem, but falls under the new product development model

Agile methods: basic concepts

**Principles:**
- Iterative development
- Customer involvement
- Support for change
- Primacy of code
- Self-organizing teams
- Technical excellence
- Search for simplicity

**Practices:**
- Evolutionary requirements
- Customer on site
- User stories
- Pair programming
- Design & code standards
- Test-driven development
- Continuous refactoring
- Continuous integration
- Timeboxing
- Risk-driven development
- Daily tracking
- Servant-style manager

Shunned: "big upfront requirements"; plans; binding documents; diagrams (e.g. UML); non-deliverable products
Another view: lean programming

- Eliminate waste
- Minimize inventory
- Maximize flow
- Pull from demand
- Empower workers
- Meet customer requirements
- Do it right the first time
- Abolish local optimization
- Partner with suppliers
- Create a culture of continuous improvement

*See www.poppendieck.com

Manager’s role in agile development

The manager does not:
- Create a work breakdown structure, schedule or estimates
- Tell people what to do (usually)
- Define and assign detailed team roles
- Abolish local optimization
- Partner with suppliers
- Build in tests; build in change
- Fret about value, not scope

The manager does provide:
- Coaching
- Service and leadership
- Resources
- Vision
- Removal of impediments
- Promotion of agile principles

Iterative development

- Each iteration is a self-contained mini-project
- Iteration goal: a release, that is a stable, integrated and tested partially complete system
- All software across all teams is integrated into a release each iteration
- Most iteration releases are internal
- During each iteration, there should be no changes from external stakeholders

Iterative development

Not a new idea (see Microsoft’s Daily Build, cluster model)
Avoid “big bang” effect of earlier approaches
Short iteration cycles

The waterfall model

Waterfall risk profile

C. Larman Agile & Iterative Development A Manager guide Addison Wesley 2003 p. 58
Risk-driven vs. client-driven planning

What would you choose to implement first?
- The riskiest, most difficult tasks...
- or
- What the client perceives as his highest business value?

Timeboxed iterative development

- Set iteration end date, no change permitted
- If requests cannot be met within timebox:
  - Place lower priority requests back on wish list
  - Never move a deadline
  - Never ask developers to work more to meet a deadline

Iterations may typically last from 1 to 6 weeks

Parkinson’s law*

Work expands so as to fill the time available for its completion

*C. Northcote Parkinson: *Parkinson’s Law, or The Pursuit of Progress*, 1957

Arguments for timeboxing

For developers:
- More focus (to limit Parkinson’s law)
- Forced to tackle small levels of complexity
For managers:
- Early forcing difficult decisions and trade-offs
- Better skill assessment of people involved and better balance and optimization provided
For stakeholders:
- They see the actual progress of the application every iteration end

Arguments against upfront requirements

- Details are too complex for people to grasp
- Stakeholders are not sure what they want
- They have difficulty stating it
- Many details will only be revealed during development
- As they see the product develop, stakeholders will change their minds
- External forces cause changes and extensions (e.g., competition)

Requirements uncertainty

Actual use of requested features

Never: 45%
Seldom: 19%
Occasionally: 16%
Often: 13%
Always: 7%

J. Johnson, XP2002

Requirements in practice, the agile view

Realistic approach, based on 200+ SW projects:

- Requirements always change
- Developers get complete specifications only 5% of the times
- On average, design starts with 58% requirements specified in detail


Evolutionary requirements analysis

Do we need to know all the functional requirements to start building a good core architecture?

- Agile answer: the architect needs most nonfunctional or quality requirements (e.g. load, internationalization, response time) and a subset of functional requirements

User stories

Test-Driven Development: basic cycle

1. Add a test
2. Run all tests and check the new one fails
3. Implement code to satisfy functionality
4. Check that new test succeeds
5. Run all tests again to avoid regression
6. Refactor code

* Test Driven Development: By Example, Addison-Wesley

TDD: a first assessment

For:
- Central role to tests
- Need to ensure that all tests pass
- Continuous execution

But:
- Tests are not specs
- Risk that program pass tests and nothing else

Stay tuned...
Scrum practices

- Self-directed and self-organizing teams of max 7 people
- No external addition of work to an iteration, once chosen
- Daily team measurement via a stand-up meeting called “scrum meeting”
- 30 calendar-day iterations
- Demo to stakeholders after each iteration

Scrum lifecycle

- Planning
- Staging
- Development
- Release

Scrum lifecycle: planning

Purpose:
- Establish the vision
- Set expectation
- Secure funding

Activities:
- Write vision
- Write budget
- Write initial product backlog
- Estimate items
- Exploratory design and prototypes

Scrum lifecycle: staging

Purpose:
- Identify more requirements and prioritize enough for first iteration

Activities:
- Planning
- Exploratory design and prototypes

Sample product backlog

<table>
<thead>
<tr>
<th>Requirement</th>
<th>N.</th>
<th>Category</th>
<th>Status</th>
<th>Pri</th>
<th>Est. (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log credit payments to AR</td>
<td>17</td>
<td>feature</td>
<td>underway</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Process sale cash scenario</td>
<td>97</td>
<td>use case</td>
<td>underway</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>Slow credit payment approval</td>
<td>12</td>
<td>issue</td>
<td>not started</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Sales commissions calculation</td>
<td>43</td>
<td>defect</td>
<td>complete</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Lay-away plan payments</td>
<td>88</td>
<td>enhance</td>
<td>not started</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>PDA sale capture</td>
<td>53</td>
<td>technology</td>
<td>not started</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Process sale c.c. scenario</td>
<td>71</td>
<td>use case</td>
<td>underway</td>
<td>5</td>
<td>30</td>
</tr>
</tbody>
</table>

Scrum lifecycle: development & release

- Sprint Planning (Product Management)
- Sprint Backlog (Product Backlog & Use Cases)
- Sprint Planning (Sprint Planning Meeting)
- Daily Stand-Up Meeting for sprint planning
- Sprint Review (Sprint Review Meeting)
- Sprint Retrospective (Sprint Retrospective Meeting)
- Done items verified and demoed by Product Owner
- Not done will be marked red and actioned
XP practices: about people

- Team typically works in an open space.
- Stakeholders are mostly available
- Every developer chooses his tasks (iteration planning game)
- Pair programming
- No overtime (sustainable pace)
- Documentation: reduced to bare minimum

XP lifecycle

- Exploration
- Planning
- Iterations to first release
- Productizing
- Maintenance

XP lifecycle: exploration

Purpose:
- Enough well-estimated user stories for first release
- Feasibility ensured

Activities:
- Prototypes
- Exploratory proof of technology programming
- Story card writing and estimating

XP lifecycle: planning

Purpose:
- Agree on date and stories of first release

Activities:
- Release planning game
- Story card writing and estimating

XP lifecycle: iterations to first release

Purpose:
- Implement a tested system ready for release

Activities:
- Testing and programming
- Iteration planning game
- Task writing and estimating

XP lifecycle: productizing

Purpose:
- Operational deployment

Activities:
- Documentation
- Training
- Marketing
XP lifecycle: maintenance

**Purpose:**
- Enhance, fix
- Build major releases

**Activities:**
- May include this phases again, for incremental releases

What about tools?

- Try to keep things as simple as possible
- Only if they really help productivity and information sharing
- Ideal situation: one relatively simple tool that seamlessly embraces all software lifecycle
- Examples: No tool (white board + camera or video), Eiffelstudio, IBM Jazz project

Agile methods links

- [www.agilealliance.com](http://www.agilealliance.com)
- [www.cetus-links.org](http://www.cetus-links.org)
- [www.xprogramming.com](http://www.xprogramming.com)
- [www.glib.com](http://www.glib.com)
- [www.craiglarman.com](http://www.craiglarman.com)
- [www.controlchaos.com](http://www.controlchaos.com)
- [www.pot.co.uk/agile-zone/papers.html](http://www.pot.co.uk/agile-zone/papers.html)
- [www.eiffelroom.com](http://www.eiffelroom.com)

Not everyone is gaga about XP

Criticisms of XP

- Hype not backed by evidence of success
- Loony ideas (e.g. pair programming)
- “What's good is not new, what's new is not good”
- Rejection of proven software engineering techniques
- Lack of design (disdained in favor of refactoring)
- Lack of documentation (disdain of "big upfront requirements")
- Unfairly favors developer over customer
- Complicates contract negotiations

Pair programming criticism

"Pair programming is necessary in XP because it compensates for a couple of practices that XP shuns: upfront-design and permanent documentation.
It makes up for the fact that the programmers are (courageously) making up the design as they code."

*(Ron Jeffries: "I think maybe concentration is the enemy. Seriously. If you're working on something that is so complex that you actually need to concentrate, there's too much chance that it's too hard")

*Slightly abridged*
At first

(Stephens & Rosenberg)

"None of this actually matters"

(Stephens & Rosenberg)

The cycle

(Stephens & Rosenberg)

Contract-Driven Development

Andreas Leitner
with Arno Fiva
(ETH)

\[ \text{CDD} = \text{TDD} \rightarrow \text{WTC}\]

Use contracts as specifications and test oracles

*Writing Test Cases

Specified but unimplemented routine

Running the system and entering input
The error is flagged at run time as a contract violation.

This has become a test case.

Another execution, another error.

This has become a second test case.

Editing the class.

Correcting an error.
Recompiling

Test cases are run again silently in the background

One bug corrected, the other not

Attempting a fix

Still doesn’t work

We fix this (or think so)
The guardian angel is, however, watching.

They fixed the bugs, had many children and lived happy ever after.

Rational Unified Process (RUP)

Process model designed by Rational (now IBM) on basis of
- Spiral model (Boehm)
- Objectory (Jacobson)

RUP practices
- Risk-driven requirements handling using use cases
- Visual modelling
- Develop in short timeboxed iterations
- Focus on component architectures
- Continuous measurement of quality factors
- Up to 50 artifacts, all optional

RUP: sample disciplines and artifacts

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Artifact (Workproduct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Vision</td>
</tr>
<tr>
<td></td>
<td>Use-Case Model</td>
</tr>
<tr>
<td>Design</td>
<td>Design model</td>
</tr>
<tr>
<td></td>
<td>Software Architecture Document</td>
</tr>
<tr>
<td>Project Management</td>
<td>Iteration Plan</td>
</tr>
<tr>
<td></td>
<td>Risk List</td>
</tr>
</tbody>
</table>
RUP lifecycle
- Inception
- Elaboration
- Construction
- Transition

RUP phases

- The good, the bad and the ugly

Retain from XP
- Principles:
  - Iterative development
  - Customer involvement
  - Support for change
  - Primacy of code
  - Self-organizing teams
  - Technical excellence
  - Search for simplicity
- Practices:
  - Evolutionary requirements
  - Customeraptive
  - User stories
  - Pair programming
  - Design & code standards
  - Test-driven development
  - Continuous testing
  - Continuous refactoring
  - Continuous integration
  - Timeboxing (where appropriate)
  - Risk-driven development
  - Daily tracking
  - Servant-style manager

Discard from XP
- Pair programming as an imposed practice
- Refusal to guarantee both functionality and delivery date
- Tests as a substitute for specifications

Retain against XP
- Key software engineering practices:
  - Extensive requirements process
  - Documentation
  - Upfront design
  - Specifications as subsuming tests
  - Role of manager
  - Commitment to both functionality and date
  - Design for generality
Retain from process-based approaches

- Engineering principles
- Documentation
- Identification of tasks
- Identification of task dependencies

Reject from process-based approaches

- Strict separation between analysis, design, implementation
- Ignorance of the central role of change in software
- Use cases as the principal source of requirements
- Tendency to "Big Bang" effect

The contribution of object technology

- Focus on abstractions
- Reuse (consumer and producer)
- Seamless development
- Reversibility
- Single Model principle:
  - The software is the model
  - The model is the software
  - The software includes everything relevant to the project
  - Tools automatically extract views

Seamless development with Eiffel & contracts

- Single notation, tools, concepts, principles
- Continuous, incremental development
- Keep model, implementation, documentation consistent
- Reversibility

The cluster model

Example classes:

- PLANE, ACCOUNT, TRANSACTION
- STATE, COMMAND
- HASH_TABLE
- TEST_DRIVER
- TABLE

Cluster model: focus & practices

- Clusters (set of related classes)
- Start with foundational clusters
- Mini lifecycles, each covering a cluster
- Seamlessness
- Reversibility
- Design by contract
- Current showable demo at every stage

Mini-lifecycle tasks

- Specification
- Design / Implementation
- Verification & Validation
- Generalization

Mini-lifecycle tasks: specification

- Identification of the data abstractions (classes) of the cluster
- Identification of constraints (class invariants)

Mini-lifecycle tasks: design & implementation

- Definition of the class architecture
  - Interface features
  - Contracts
- Definition of the relationships between classes
  - Client
  - Inheritance
- Finalization of classes

Mini-lifecycle tasks: verification & validation

- Static examination
- (Possibly) automated unit testing

Mini-lifecycle tasks: generalization

Goal: turn classes in potentially reusable software components via:

- Abstracting
- Factoring
- Documenting
Generalization

Prepare for reuse. For example:
- Remove built-in limits
- Remove dependencies on specifics of project
- Improve documentation, contracts...
- Abstract
- Extract commonalities and revamp inheritance hierarchy

Few companies have the guts to provide the budget for this

Software engineering principles

Quality pays
Architecture
Extendibility
Reusability
Reliability