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
- 1 -

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.
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>Can 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>

C. Larman Agile & Iterative Development *A Manager guide* Addison Wesley 2003
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

“Documentation that is not part of the final program”

Iterative development

Decide as late as possible

Build in tests; build in change

Fret about value, not scope

*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

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

- Feasibility study
- Requirements
- Specification
- Global design
- Detailed design
- Implementation
- V & V
- Distribution
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

![Bar chart showing requirements change for different project sizes in FP](chart.png)

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

Copyright © 2003 United Feature Syndicate, Inc.
Test-Driven Development: basic cycle

After Kent Beck

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 commission 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>

C. Larman Agile & Iterative Development A Manager guide Addison Wesley 2003
Scrum lifecycle: development & release

Sprint Backlog: Feature(s) assigned to sprint

Backlog items expanded by team

every 24 hours

Scrum: 15 minute daily meeting. Teams member respond to basics:
1) What did you do since last Scrum Meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?

Product Backlog: Prioritized product features desired by the customer

New functionality is demonstrated at end of sprint
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
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
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
www.cetus-links.org
www.xprogramming.com
www.gilb.com
www.craigrarman.com
www.controlchaos.com
www.pols.co.uk/agile-zone/papers.html
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: up-front-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

DaimlerChrysler: The Best Team in the World

Chet Hendrickson, DaimlerChrysler

- Team: 10 programmers, 15 total
- Application: large-scale payroll system
- Time: four years

The C3 project began in January 1995 under a fixed-priced contract that called for a joint team of Chrysler and contract partner employees. Most of the development work had been completed by early 1996. Our contract partners had used a very GUI-centered development methodology, which had ignored automated testing. As a result, we had a payroll system that had

Figure 2.2 We're the King of the World!
"None of this actually matters" (Stephens & Rosenberg)

None of this actually matters, because building a payroll systems was C3’s secondary goal. I don’t think anyone has written about this before, mostly because it happened before Ron Jeffries joined the team. The team’s original charter, and it was reiterated when the decision to bring in Kent Beck was made, was to learn how to use object technology, to learn how to manage projects that use it and if we built a new payroll system, that would be gravy.

There can be no question that we achieved the first two. New software at DaimlerChrysler is being written using objects (if you can call Java objects). Management was not happy that we didn’t replace the old payroll systems, but they didn’t ride us out of town on a rail. C3 alumni have gone on to lead development efforts in areas central to the company’s success, areas such as cost management, vehicle manufacturing and personnel management. Reports of XP’s demise at DaimlerChrysler have been greatly exaggerated. In fact, we are beginning to see a second generation of conference speakers come out of DaimlerChrysler. At the 2001 JavaOne conference Dave Boehme, gave a talk about how his team, with the help of a C3 alumnus, turned around a large scale J2EE project.

As best as I can tell, the decision to stop C3 development was made, not because we were wasting the company’s time and money, but because it was time to use what we had learned on more important problems. We did not work for a payroll processing company, we worked for an automobile maker.

The techniques learned on C3 are now being used on projects that impact the bottom line. As a stockholder, I think it is a good thing.

With all that being said, what really happened at C3?

I don’t think that there is one simple answer to that question. The best answer is that we stopped providing value to our customer. Elsewhere on this page the bifurcation of our customer is discussed. The fact that we had two customers, with different goals, means that we violated the principle of a single, end-to-end customer. Let this be a lesson to you: Your customer.

Figure 2-10. Building a payroll system was C3’s secondary goal.
The cycle

Figure 2-1. Activities during and after C3

(Stephens & Rosenberg)
Contract-Driven Development

CDD = TDD — WTC*

Use contracts as specifications and test oracles

*Writing Test Cases
Specified but unimplemented routine

class BANK_ACCOUNT

inherit

redefine

default_create

don

balance := 300

done

end

feature -- Initialization

default create is
don

balance := 300

done
end

feature -- Access

balance: INTEGER

feature -- Basic operations

deposit (an_amount: INTEGER) is
do

deposit

balance increased: balance = old balance + an_amount

done

withdraw (an_amount: INTEGER) is
do

withdraw

balance decreased: balance = old balance - an_amount

withdraw

end

end

Invariant

balance_not_negative: balance >= 0
Running the system and entering input

(Erroneous)
The error is flagged at runtime as a contract violation.
Another execution, another error

Status: implicit exception pending
Code: 4 (Postcondition violated)
Tag: balance_decreased
This has become a second test case
Editing the class

```plaintext
bank_account

deposit amount: INTEGER) is
  do
    ensure
      balance increased: balance > old balance
      deposited: balance = old balance + amount
  end

withdraw (amount: INTEGER) is
  do
    ensure
      balance decreased: balance < old balance
      withdrawn: balance = old balance - amount
  end

invariant
  balance not negative: balance >= 0
end
```
Correcting an error

deposit (an_amount: INTEGER) is
  do
    balance := balance + an_amount
  ensure
    balance_increased: balance > old balance
    deposited: balance = old balance + an_amount
  end

withdraw (an_amount: INTEGER) is
  do
    ensure
      balance_decreased: balance < old balance
      withdrawn: balance = old balance + an_amount
  end

invariant
  balance_not_negative: balance >= 0
end

System
  name: bank_account
  target: bank_account
  config file: /home/aleitner/eclipse/cdd_es/Src/examples
  location: /home/aleitner/eclipse/cdd_es/Src/examples
  compilation: /home/aleitner/eclipse/cdd_es/Src/examples
Root class
  APPLICATION (root_cluster): make_and_launch

Test Cases
  root_cluster
    deposit
      Test case #01: F
    withdraw
      Test case #02: F
deposit (an_amount: INTEGER) is
  do
    balance := balance + an_amount
  ensure
    balance_increased: balance > old balance
deposited: balance = old balance + an_amount
end

withdraw (an_amount: INTEGER) is
  do
    ensure
      balance_decreased: balance < old balance
withdrawn: balance = old balance + an_amount
end

Invariant
  balance_not_negative: balance >= 0
end
Test cases are run again silently in the background.
One bug corrected, the other not
Attempting a fix

```plaintext
deposit (an_amount: INTEGER) is
  do
    balance := balance + an_amount
  ensure
    balance_increased: balance > old balance
    deposited: balance = old balance + an_amount
  end

withdraw (an_amount: INTEGER) is
  do
    balance := balance - an_amount
  ensure
    balance_decreased: balance < old balance
    withdrawn: balance = old balance - an_amount
  end

invariant
  balance_not_negative: balance >= 0
end
```

Context: root_cluster BANK_ACCOUNT withdraw

Degree 8: Examining System
Degree 5: Parsing Classes
Degree 4: Analyzing Inheritance
Degree 3: Checking Types
Degree 2: Generating Byte Code
Degree 1: Generating Metadatas
Managing System Changes
Eiffel Compilation Succeeded
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.
Process-based approaches: RUP
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>

C. Larman Agile & Iterative Development A Manager guide Addison Wesley 2003
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
- Customer on site
- 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

<table>
<thead>
<tr>
<th>Focus on abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse (consumer <em>and</em> producer)</td>
</tr>
<tr>
<td>Seamless development</td>
</tr>
<tr>
<td>Reversibility</td>
</tr>
</tbody>
</table>

**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

("Primacy" of code, but in a more far-reaching sense than in plain XP)

Contracts as a guide to analysis, design, implementation, maintenance, management

Continuous testing based on contracts

Seamless development with Eiffel & contracts

Single notation, tools, concepts, principles

Continuous, incremental development

Keep model, implementation, documentation consistent

Reversibility

Example classes:

- PLANE, ACCOUNT, TRANSACTION...
- STATE, COMMAND...
- HASH_TABLE...
- TEST_DRIVER...
- TABLE...
The cluster model

Cluster 1

Cluster 2
The cluster model

Cluster 1

Cluster 2

Cluster n
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
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