Software Architecture

Bertrand Meyer, Michela Pedroni

ETH Zurich, February-May 2010

Lecture 9: Agile methods

(with material developed 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 or heavyweight)

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
The waterfall model

1. Feasibility study
2. Requirements
3. Specification
4. Global design
5. Detailed design
6. Implementation
7. V & V
8. Distribution
Waterfall risk profile

C. Larman Agile & Iterative Development A Manager guide Addison Wesley 2003
p. 58
Criticism on process-oriented

**Requirements:**
- difficult to define in the beginning of a project
- may change over time
- may not capture what customer wants (which may change over time)

**Planning:**
- difficult because requirements change

**Design and implementation:**
- reveals problems only late in the project

**Testing:**
- reveals problems only at the end of the project
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.
Process-oriented was found to be
- heavy weight
- bureaucratic
- slow
- demeaning
- inconsistent with how developers perform effective work

Software development seen as prototype-style manufacturing

Reaction to this: lightweight methods (e.g. Scrum, XP described in the mid 1990s)

In 2001, lightweight methods became agile methods and the agile manifesto defined principles
“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.

agilemanifesto.org
13 Agile principles

1. Satisfy customers (early & continuous delivery)
2. Embrace changing requirements
3. Deliver working software frequently (weeks - months)
4. Business people & developers collaborate daily
5. Build projects around motivated individuals
6. Face-to-face conversations
7. Progress = working software
8. Agile processes promote sustainable development
9. Constant pace for all stakeholders
10. Technical excellence and good design
11. Goal is simplicity (maximize work not done)
12. Self-organizing teams
13. Reflection on team effectiveness and improvements
### Agile methods: overview of basic concepts

<table>
<thead>
<tr>
<th>Principles:</th>
<th>Practices:</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Iterative development</td>
<td>➢ Evolutionary requirements</td>
</tr>
<tr>
<td>➢ Support for change</td>
<td>➢ Customer on site</td>
</tr>
<tr>
<td>➢ Customer involvement</td>
<td>➢ User stories</td>
</tr>
<tr>
<td>➢ Primacy of code</td>
<td>➢ Pair programming</td>
</tr>
<tr>
<td>➢ Self-organizing teams</td>
<td>➢ Design &amp; code standards</td>
</tr>
<tr>
<td>✓ Technical excellence</td>
<td>➢ Test-driven development</td>
</tr>
<tr>
<td>✓ Search for simplicity</td>
<td>➢ Continuous refactoring</td>
</tr>
</tbody>
</table>

Shunned: “big upfront requirements”; plans; binding documents; diagrams (e.g. UML); non-deliverable products

➢ Continuous integration
➢ Timeboxing
➢ Risk-driven/client-driven development
➢ Daily tracking
➢ Servant-style manager
Iterative development

- Iteration activities: usually all but with varying weights
  - self-contained mini-projects
    - specification
    - planning
    - implementation
    - continuous integration (across teams)
    - testing
    - release (usually internal)

- Iteration goal: a stable, integrated, tested, partially complete (internal) release

- Iteration cycles: short (some weeks)

- Changes to requirements by externals: For a new iteration only

Not a new idea (see Microsoft’s Daily Build, cluster model)
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*
Disciplines across iterations

Iterative Development
Business value is delivered incrementally in time-boxed cross-discipline iterations.

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
## Agile methods: overview of basic concepts

<table>
<thead>
<tr>
<th>Principles:</th>
<th>Practices:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Iterative development</td>
<td>➢ Evolutionary requirements</td>
</tr>
<tr>
<td>➢ Support for change</td>
<td>➢ Customer on site</td>
</tr>
<tr>
<td>➢ Customer involvement</td>
<td>➢ User stories</td>
</tr>
<tr>
<td>➢ Primacy of code</td>
<td>➢ Pair programming</td>
</tr>
<tr>
<td>➢ Self-organizing teams</td>
<td>➢ Design &amp; code standards</td>
</tr>
<tr>
<td>✓ Technical excellence</td>
<td>➢ Test-driven development</td>
</tr>
<tr>
<td>✓ Search for simplicity</td>
<td>➢ Continuous refactoring</td>
</tr>
<tr>
<td></td>
<td>➢ Continuous integration</td>
</tr>
<tr>
<td></td>
<td>✓ Timeboxing</td>
</tr>
</tbody>
</table>

Shunned:

“big upfront requirements”; plans; binding documents; diagrams (e.g. UML); non-deliverable products

➢ Risk-driven/client-driven development
➢ Daily tracking
➢ Servant-style manager
 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.
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?
## Agile methods: overview of basic concepts

<table>
<thead>
<tr>
<th>Principles</th>
<th>Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Iterative development</td>
<td>✓ Evolutionary requirements</td>
</tr>
<tr>
<td>✓ Support for change</td>
<td>✓ Customer on site</td>
</tr>
<tr>
<td>⊚ Customer involvement</td>
<td>✓ User stories</td>
</tr>
<tr>
<td>⊚ Primacy of code</td>
<td>✓ Pair programming</td>
</tr>
<tr>
<td>⊚ Self-organizing teams</td>
<td>✓ Design &amp; code standards</td>
</tr>
<tr>
<td>✓ Technical excellence</td>
<td>✓ Test-driven development</td>
</tr>
<tr>
<td>✓ Search for simplicity</td>
<td>✓ Continuous refactoring</td>
</tr>
<tr>
<td></td>
<td>✓ Continuous integration</td>
</tr>
<tr>
<td>Shunned:</td>
<td>✓ Timeboxing</td>
</tr>
<tr>
<td>“big upfront requirements”; plans; binding</td>
<td>✓ Risk-driven/client-driven development</td>
</tr>
<tr>
<td>documents; diagrams (e.g. UML); non-deliverable</td>
<td>✓ Daily tracking</td>
</tr>
<tr>
<td>products</td>
<td>✓ Servant-style manager</td>
</tr>
</tbody>
</table>
Customer on site

One or more customers sit (full-time) with the development team

- decisions on requirements and priorities
- explanations of features to the programmers
- help plan an iteration
- write acceptance tests in collaboration with a developer
User stories

“User stories are a reminder to have a conversation with your stakeholders”

Features, fixes, non-functional requirements

Front of Card

As a student I want to purchase a parking pass so that I can drive to school

Priority: No. 1
Estimate: 4

Back of Card

Confirmation:
The student must pay the correct amount
One pass for one month is issued at a time
The student will not receive a pass if the payment isn’t sufficient
The person buying the pass must be a currently enrolled student
The student may only buy one pass per month.

Copyright: 2005-2009 Scott W. Ambler
User stories

EXTREME PROGRAMMING

I CAN'T GIVE YOU ALL OF THESE FEATURES IN THE FIRST VERSION.

AND EACH FEATURE NEEDS TO HAVE WHAT WE CALL A "USER STORY."

OKAY, HERE'S A STORY: YOU GIVE ME ALL OF MY FEATURES OR I'LL RUIN YOUR LIFE.
## Agile methods: overview of basic concepts

### Principles:
- Iterative development
- Support for change
- Customer involvement
- 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/client-driven development
- Daily tracking
- Servant-style manager

### Shunned:
“big upfront requirements”; plans; binding documents; diagrams (e.g. UML); non-deliverable products
Pair programming

All code is produced by two programmers at one computer:
- they rotate the input devices
- pairs change frequently
- observer reviews code in real time and considers tests

Benefits:
- cross learning
- peer pressure
- help when programmer is stuck

In XP, pairs change: code and design standards need to be followed.
Test-Driven Development: basic cycle

1. Add a test
2. Run all tests and check that 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
Continuous refactoring

Getting architectural software design right is difficult.

Some agile methods (XP in particular) reduce or even avoid design thought before implementation.

Instead they advocate continuous refactoring of code (about 25% of total effort).

TDD = Test first development + refactoring
# Agile methods: overview of basic concepts

**Principles:**
- Iterative development
- Support for change
- Customer involvement
- 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/client-driven development
- **Daily tracking**
- **Servant-style manager**

**Shunned:**
- “big upfront requirements”; plans; binding documents; diagrams (e.g. UML); non-deliverable products
Self-organizing teams

Development team organizes

- how to fulfill the goals
- how to solve problems
- how to plan the work

What does the manager do then?
Manager’s role in agile development

<table>
<thead>
<tr>
<th>The manager does not:</th>
<th>The manager does provide:</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Create a work breakdown structure, schedule or estimates</td>
<td>➢ Coaching</td>
</tr>
<tr>
<td>➢ Tell people what to do (usually)</td>
<td>➢ Service and leadership</td>
</tr>
<tr>
<td>➢ Define and assign detailed team roles</td>
<td>➢ Resources</td>
</tr>
<tr>
<td></td>
<td>➢ Vision</td>
</tr>
<tr>
<td></td>
<td>➢ Removal of impediments</td>
</tr>
<tr>
<td></td>
<td>➢ Promotion of agile principles</td>
</tr>
</tbody>
</table>
Daily tracking and communication

A tracker collects task and user story progress on foot by talking to the developers

“XP is about people, not computers.” (Ron Jeffries)

Direct communication is favored over e-mail (and others) → Developers and customers need to be collocated
2 agile methods + 1 process-based approach

1. Scrum
2. Extreme Programming
3. Rational Unified Process

(We have seen some elements of each already)
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 (called Scrum Sprints)
- Demo to stakeholders after each iteration
- Each iteration, client-driven adaptive planning
- Scrum team in one room

- Not discussed: rigor, specific work products
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
### 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>
Scrum lifecycle: staging

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

Activities:
- Planning
- Exploratory design and prototypes
Scrum lifecycle: development & release

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?

“chickens and pigs”
- decisions in 1 hour
- gone in 1 day

“blocks”:
- decisions in 1 hour
- gone in 1 day

New functionality is demonstrated at end of sprint

Sprint review:
- demo
- ppt forbidden

Product Backlog:
Prioritized product features desired by the customer

Sprint Backlog:
Feature(s) assigned to sprint
Backlog items expanded by team

every 24 hours
30 days
### Sample Sprint backlog

<table>
<thead>
<tr>
<th>Task description</th>
<th>Resp.</th>
<th>Status</th>
<th>Hrs work remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>362</td>
</tr>
<tr>
<td>Meet to discuss goals</td>
<td>JM/SR</td>
<td>completed</td>
<td>322</td>
</tr>
<tr>
<td>Move calculations</td>
<td>AW</td>
<td>not started</td>
<td>317</td>
</tr>
<tr>
<td>Get GEK Data</td>
<td>TN</td>
<td>completed</td>
<td>306</td>
</tr>
<tr>
<td>Analyse GEK Data - Title</td>
<td>GP</td>
<td>in progress</td>
<td></td>
</tr>
<tr>
<td>Analyse GEK Data - Parcel</td>
<td>TK</td>
<td>completed</td>
<td></td>
</tr>
<tr>
<td>Define and build db</td>
<td>BR/DS</td>
<td>in progress</td>
<td></td>
</tr>
</tbody>
</table>

*C. Larman Agile & Iterative Development A Manager guide Addison Wesley 2003*
XP practices: about people

- Team typically works in an open space.
- Stakeholders are mostly available on site.
- Every developer chooses his tasks (iteration planning game, user story cards).
- Pair programming, TDD, continuous integration.
- No overtime (sustainable pace).
- Short iterations (1 - 3 weeks).
- Documentation: reduced to bare minimum (favors oral communication).
- Design: Light modeling (no up-front design work).
Why eXtreme?

Beck: “turn the dial up to 10”

- Testing is good: do it from day 1 (TDD)
- Code reviews are good: do it instantly (pair programming)
- Frequent integration is good: 24/7 with automated, continuous integration process on dedicated build machine
- Short iterations are good: make them 1 - 3 weeks
- Customer involvement is good: bring customers on site
- Communication is good: pair programming, customer on site, remove documentation overhead
XP lifecycle

- Exploration
- Planning
- Iterations to first release
- Productizing
- Maintenance
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
XP lifecycle: maintenance

Purpose:
- Enhance, fix
- Build major releases

Activities:
- May include this phases again, for incremental releases
XP programmer typical workday

- Stand up meeting at 9:00
- Pair up - Quick design session
- Test
- Code
- Refactor
- Integrate or Toss
- Go home at 17:00
- Q&A
A process-based iterative approach: RUP

Rational Unified Process (RUP):
- Iterative and incremental SW development process
- Extensible framework to be customized
- Based on Spiral model (Boehm), Objectory (Jacobson)
- Unified Process (UP), sometimes names used interchangeably
- Agile Unified Process (AUP) available
RUP practices

- Risk-driven requirements handling using use cases
- Visual modeling (sketching on the whiteboard)
- Develop in short timeboxed iterations
- Focus on component architectures
- Continuous measurement of quality factors
- Up to 50 *artifacts*, all optional
  - “less is better”
  - tailored to the project’s/team’s needs in Development Case
## 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 (days)
  - requirements workshop, 10% of reqs captured, “top ten” reqs list, vision draft
  - Goal: identify scope, vision, priorities, risks, first plan

- Elaboration (iterations...):
  - core elements programmed & tested, reqs workshops, design, programming, testing, reqs might change
  - Goal: stabilize vision, requirements, architecture

- Construction (iterations...):
  - build remainder of the system, alpha testing, performance tuning, document creation, reqs change only rarely
  - Goal: system ready for deployment

- Transition (iterations):
  - release candidate for feedback, distribution, education
  - Goal: system deployed
RUP phases

**Iterative Development**

Business value is delivered incrementally in time-boxed cross-discipline iterations.

<table>
<thead>
<tr>
<th>Inception</th>
<th>Elaboration</th>
<th>Construction</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>C1</td>
</tr>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>C2</td>
</tr>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>C3</td>
</tr>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>C4</td>
</tr>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>T1</td>
</tr>
<tr>
<td>I1</td>
<td>E1</td>
<td>E2</td>
<td>T2</td>
</tr>
</tbody>
</table>

- **Business Modeling**
- **Requirements**
- **Analysis & Design**
- **Implementation**
- **Test**
- **Deployment**

Main intentions

- Attack risks early and continuously
- Deliver customer value early and often
- First focus on software development, then on documentation
- Accommodate change early
- Work component-oriented: reuse
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
Not everyone is gaga about XP

Matt Stephens and Doug Rosenberg

Extreme Programming Refactored: The Case Against XP

- Cuts through the hype and tells "the other side of the story" about Extreme Programming (XP)
- Provides a thorough and systematic analysis of XP practices, and separates the "agile" from the "fragile"
- Proposes better ways of achieving XP's agile goals that are applicable to a much wider range of projects

The XP Series

Pete McBreen

Foreword by Kent Beck
Criticisms of XP

- Customers on site: difficult to find them
- Lack of documentation: difficult for maintenance
- Refactoring: design changes result in updates of tests, may introduce faults
- 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”
- Adaptive planning: complicates contract negotiations
- Distributed teams?
- Large projects/teams?
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, because building a payroll system 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 goal on, one customer. Let this be a lesson to you. Your customer...
IMHO (Michela’s)

Very difficult to empirically validate approaches and practices

- Too many unknown/uncontrollable variables

What might work for one project, might not work for another
What might work for one team, might not work for another
What might work for one organization, might not work for another
Agile methods links

www.agilealliance.com
www.cetus-links.org
www.xprogramming.com
www.gilb.com
www.craiglarman.com
www.controlchaos.com
www.pols.co.uk/agile-zone/papers.html
www.eiffelroom.com