Extreme Programming Considered Harmful for Reliable Software Development

Status:	Approved
Author(s):	Gerold Keefer
Version:	1.0
Last change:	06.02.02 21:17
Project phase:	Implementation
Document file name:	ExtremeProgramming.doc
Approval Authority:	Gerold Keefer
Distribution:	Public
Security Classification:	None
Number of pages:	1

© 2002 AVOCA GmbH

Ein unbegreiflich Licht erfüllt den ganzen Kreis der Erden.
Es schallet kräftig fort und fort.
Ein höchst erwünscht Verheißungswort.
Wer glaubt, soll selig werden.

Aria, BWV 125

1 Motivation

Following my positive experience with software development in a large telecommunication software project whereby I inspected and unit tested the code of my partner and vice versa, I came across websites that recommended programming in pairs as a practice of a software development method called “Extreme Programming”. I thought this practice to be about the same what we did in that project and recommended it to others. It was about three years later that I became hired as the quality manager of a project that involved around thirty developers and was one of several subprojects of a large SAP migration.
On this project one contractor team consisting of about seven programmers intended to develop a Java framework that others would use in the Extreme Programming fashion. At that time I was unbiased towards Extreme Programming, because I simply did not know much about it. What I did know was that a project of that size required substantial planning, analysis, and architectural groundwork up front. This conflicted with the view of that team, which lead to ongoing debates. The chief architect and I clearly favoured the software engineering approach and the customers project leader stood somewhere in the middle, he saw advantages in knowledge transfer to his staff when using pair programming. During the risks analysis that I conducted, conflicting software development philosophies were consequently identified as the topmost risk. In general, we experienced a very slow progress from this team during the development of an architectural prototype. Eventually their leader had to confess that they would not be able to deliver the required work products and apologized for the wasted time and resources. Initially it took an application based on the framework around 40 minutes start up and the memory consumption was unacceptable, clearly exceeding the most moderate performance and resource constraints we had.

During the last two years I noticed an ever-growing influence of Extreme Programming on mainstream software development, due to its increasing popularity. This popularity was fuelled by support from respected people like Tom DeMarco and recurrent claims that put Extreme Programming to the level of alchemy, such as one article reporting on an organization employing Extreme Programming that moved from CMM Level 1 to CMM Level 4 within 5 months [Griffin01CMMLevel4].
Everyone was reading Extreme Programming books, Extreme Programming became a widespread conference topic and as I approached the local university campus the first questions I heard was: ”Extreme Programming is great – isn’t it?”.
Later on I became involved in many heated debates in Usenet newsgroups, most notably comp.software.extreme-programming, about the validity of the claims and practices of Extreme Programming and investigated deeper. Quite soon I learned that most of them were based on rather thin grounds.

My motivation for writing this paper is to put together the information and references I have about the facts of Extreme Programming and how they relate to traditional Software Engineering practices. I will state reasons, albeit no scientific proofs, why I consider Extreme Programming to be harmful for reliable software development and why it provides not many answers to the significant questions that contemporary software development methods have to answer. Secondly, the Extreme Programming hype and its variant in the “Agile Methods” packaging is so dominant, that even if we wanted to ignoring it is no longer an option: My company will give tutorials on “Agile Methods”, because I consider the SCRUM-Method as recommendable.

As you can see from the reference section, the practices of Extreme Programming have been explained in many publications. In this paper I will not focus on iterating them. I will directly address the harm as well as the benefit I see accompanying the method.

© 2002 AVOCA GmbH

2 Bias

Everyone who followed my comments and thoughts on the Extreme Programming newsgroup will see that I have my opinion about major aspects of Extreme Programming firmly set. In other words I am biased. I consider if appropriate to define my bias before we move further.

My degree is in telecommunication engineering and I clearly favour an engineering approach to software development. During the past 5 years I was mainly involved in quality assurance and testing consulting tasks with my company. I moved into the quality assurance direction shortly after experiencing all the pitfalls of a non-quality oriented development environment:

- Reliance on verbal communication.
- Intuition-based decision making.
- High dependency on individual skills.
- Insufficient planning.
- No sound approach to system verification and validation.

Those practices lead to

- rework as a norm,
- error prone systems,
- non-maintainable systems,
- frustrated staff, and
- few heroes.

Out of those experiences I am convinced that for software projects of any size explicit management of quality is imperative for reliable success. This is my bias.

3 Benefits

The Extreme Programming approach to testing:
My prediction is that in the long run Extreme Programming will enjoy more respect as a vehicle for the widespread introduction of testing worth its name, than for anything else. I have substantial doubts about how Extreme Programming addresses testing in detail as described for example in [Crispin01Testing]. One concern is that 100% test automation in most cases is not feasible and frequent releases will therefore add manual test effort. However, there is no denial of the fact that it brought testing basics to the masses.
The word was written down in the testing bibles ([Beizer90STT], [Myers79AST]) decades ago, but Extreme Programming was the gospel that spread it. Would you have succeeded in convincing anybody to put 30 or 40 % of the project resources into testing, before the arrival of Extreme Programming?

The visualization aspects:
Besides the testing benefit, I see remarkable benefits in trying to visualize the project to all stakeholders involved. The first measure trying to achieve this is the system metaphor. Despite the fact that I consider it as naïve to assume that you can replace the system architecture by a system metaphor for anything else but trivial systems, as suggested in [Beck00XP], the idea to express the system visually is good and something I have been proposing for a long time.
Despite the fact that I am not very ecstatic about the 3 x 5 cards to write down stories, I feel very positive about the visualization aspect of the project whiteboard. Transferred to the corporate intranet, these ideas are very useful.

The people orientation:
Extreme Programming sees itself as a “humanistic discipline of software development”. Project problems are mainly people problems, as Gerald M. Weinberg once pointed out: “No matter what the problem is, it's always a people problem”. However, it is highly doubtful to solve people problems with

© 2002 AVOCA GmbH

a process. If that was possible, you could establish the right process and all problems were solved. This is a false impression Extreme Programming seems to create. Even worse, there are indications that Extreme Programming leads to a tendency to blame the people, not the process [Griffin01Toxic].

4 Dubious Values and Practices

The "80% benefit with 20% work"-rule:
I have seen too many 80/20 software systems. At the very heart those systems are insincere, because they give a false impression of reliability, safety, or security to the user and as the system fails not only the systems credibility is gone, but also a part of the credibility of software technology as a whole.
Prof. Alan V. Oppenheim of the MIT stated in [Oppenheim92Edu]:
"I realized many years ago when doing a major project on my home that you can paint 90 percent of a room in 10 percent of the time. The walls are easy. It is careful attention to the trim and other fine details that makes the difference." This is in line with the observation that the C3 project was initially highly productive, but got cancelled after it did not meet the expected schedule.

The embrace change value:
Change is good and it is a need of the markets. However, a sound system cannot be established in an environment of constant and unbound transitions. This will undermine its conceptual integrity. The impression that changes are easy with software is one of the major reasons for software troubles. Fuelling that impression with dubious cost of change calculations, as in [Beck00XP], does not help.
Ancient fortresses have been built on the top of hills, to allow identifying visitors in advance. Avoiding up-front analysis is like putting yourself into the valley, where the visitor is at your doorstep before you notice.

The practice of refactoring:
Hand in hand with the "embrace change"-value goes the practice of refactoring: After initial coding the code gets beautified and the design improved. Although it is a good idea to keep source code as readable as possible and the design sound, there is not much point in doing afterwards what can be done from the start. Considering the fact that Extreme Programming requires a functioning unit test suite all the time, refactoring will lead to considerable extra effort in adapting this test suite. Moreover, any code change is an opportunity for error injection.

The simplicity value:
Simplicity is a universal and highly respected design principle. However, in most cases it is a complex process that finally comes up with a simple and elegant solution. "As simple, as possible, but not simpler", once stated by Albert Einstein and often refrained by Niklaus Wirth [Gutknecht00Wirth] is fundamentally superior to Extreme Programming's "the simplest thing that could possibly work."

The practice of pair programming:
Whatever might be stated in Extreme Programming publications, such as "There have been several studies conducted on the topic of pair programming (See the work of Laurie Williams at the University of Utah, lwilliam@cs.utah.edu). These studies show that pairing causes no loss of productivity at all, while significantly decreasing defect rate, code size, and job dissatisfaction." in [Martin00Pair], the facts are:

1. There have been three studies on pair programming: One in a maintenance environment on a 45 minute task [Nosek98Pair] and two in student settings [Nawrocki01Pair] and [Williams00Diss], [Williams00Strengthening], [Williams00Classroom], [Williams01Student].
2. In all three studies the total effort pairs spent on their assignments was higher than that of solo programmers.
3. The most recent study [Nawrocki01Pair] suggests that "[…] pair programming is rather expensive technology." and "XP-like pair programming appears less efficient than it is reported by J.T. Nosek [5] and L. Williams et al. [6]. Only in one case (Program 4) reduction of average development time for XP2 (to 77% of the time needed by XP1) was similar to that reported by J. T. Nosek (reduction to 71%) but still it is far from the level of 50% mentioned by L. Williams

> et al. Moreover, in case of Program 4 one of the students misunderstood the assignment and this distorted the results."

So far there is no evidence about the usefulness of this practice in general, particularly compared to the use of reviews and inspections. In addition there are several reports that indicate that pairing leads to faster exhaustion [Dittert01Pair]. On the other hand no one will deny that pairing in certain situations, such as educational settings or debugging sessions, is good common sense.

The practice of having an on-site customer:
Everybody would welcome a customer like the one Extreme Programming proposes on its projects. The problem is that in real-world projects such a person hardly ever exists and various experience reports, such as [Dittert01Pair], [Hendrickson01Customer], and [Grenning01Launch], underline this. The first Extreme Programming customer in history left the (C3-)project and could not be adequately replaced.

The practice of having no documentation:
Extreme Programming made it into the headlines partly by proposing "no documentation" on related web sites. Later on this slogan was changed to "documentation as needed" and oxymorons like "XP is not anti-documentation; it just recognizes that documentation has a cost and that not creating it might be more cost-effective." [Grenning01Launch] Who would not like to be "cost-effective" and avoid the cumbersome task of documenting software systems? However, considering maintenance and usage aspects, proposing "no documentation" is clearly professional malpractice.

The practice of having 40-hour weeks:
I would welcome this practice, albeit it compromises Brook's "as few minds as possible"-rule. What I am critical about is that those who propose the practice seem to avoid it frequently [Schuh01Recovery], [Hendrickson01Customer]. This sounds like a reputation booster to unfairly attract attention.

The practice of collective code ownership:
It has been reported that collective code ownership works fine in some environments. However, the idea is based on altruism and therefore severely limited in scale and depends on suitable team settings. If the team culture gets disrupted by internal or external causes, the collective ownership will become non-ownership in due time.

The "everything-is-in-the-code"-attitude:
At its heart Extreme Programming is a code-centric approach. "The design is in the code", "the code documents itself", or "the requirements are documented with code for test case execution" are repeatedly heard assertions. The reality of this attitude is a return to garage duo programming Fred Brooks tells us about in [Brooks95Mythical]. Brooks estimated that the effort to transform such programs into a generally useful software system takes nine times the effort initially required to have a running program. A precise estimate would depend on the specific environment and there may be environments that are happy with only the code, but in general a software system is more than code and Extreme Programming does not describe how to create work products beyond code.

The "test cases are requirements"-recommendation:
Some of the proponents of Extreme Programming consider acceptance test cases to be sufficient as a substitute for documented requirements. Documenting requirements is hard work even in a natural language. It is highly doubtful if expressing them in Java or C++ will be an improvement to that situation.

The non-specialization recommendation:
Like every other industry we have and will have specialization in software development. Due to the high technology turnover, the depth of specialization is certainly limited. But software development has passed by the grass root times were everyone was supposed to do everything. Peter Drucker recently and rightfully pointed out "effective knowledge is specialized". Due to ambiguous statements on this

issue from the Extreme Programming experts [Hendrickson01Customer], the impression is that they are fuzzy about their own point.

5 C3 Revisited

The “Chrysler Comprehensive Compensation System”, C3 in short, may well be the most often cited and referenced software development project in history and it was the “proof by example” of the Extreme Programming inventors. Of the 20 participants named in [Beck98C3], five subsequently published a current total of three books ([Beck00XP], [Jeffries00Installed], [Beck01PlanningXP]) about extreme programming and a myriad of articles ([Jeffries99ET], [BeckIEEE99Embrace], [Williams00Strengthening], [Hendrickson01Customer], [Haungs01Pair]). Numerous other articles and books reference the project, for example [Cockburn01Select] or [Williams00Strengthening].

Despite the heaps of information given, I have ceased the search for a clear picture about what really happened in C3. The exaggerations are obvious. In [Haungs01Pair] it is stated that the project “was to scale up to pay hundreds of thousands of people every week” or in [Cockburn01Select] there is talk about “The recently completed Chrysler Comprehensive Compensation System (C3) experience.”
I found the most credible information in [Beck98C3], [BeckIEEE99Embrace] (the framed report by Chet Hendrickson) and [Hendrickson01Customer]:

- January 1995: C3 was launched under a fixed price contract with a joint Chrysler and contractor team.
- Since March 1996: C3 was conducted in Extreme Programming style after Kent Beck arrived, according to Ron Jeffries. At that time the fix price contractor had failed to deliver a working product and the project was in a mess, particularly it is reported that there was no sound testing in place.
- August 1998: C3 was paying a pilot group of around 10 000 people.
- February 2000: C3 was cancelled after being nearly four years in Extreme Programming mode.

Other facts that are credibly reported:

- C3 experienced a high productivity in the first 30 weeks after Ken Beck arrived and the staff was significantly scaled down, following his advice.
- C3 was supposed to serve as a payroll application for a total of 87 000 employees and get operational by mid 1999 [Beck98C3].
- C3 was at no point in time used to pay more than around 10 000 employees, but has proven to be capable to pay another 20 000.
- It is reported in [Hendrickson01Customer] that the first person to play the customer role of Extreme Programming left the project due to burn out after a few months and could not be adequately replaced.
- Extreme Programming was retired as a development method after this project at the then DaimlerChrysler Company.

Considering the fact that for a long time all of the major Extreme Programming experts were involved and they used Smalltalk, which is considered to be most suitable for Extreme Programming due to its dynamic typing, the project outcomes are not particularly convincing.

6 Missing Answers

So far Extreme Programming has failed to give answers to a range of important questions contemporary software development faces:

1. **How does Extreme Programming work with fixed scope, fixed price, and fixed schedule contracts?** Extreme Programming works similar to a subscription. As a customer you

subscribe iteration by iteration. This certainly gives you a clear picture about costs and functionality for the next few weeks, but what about total project costs, total project effort, and the overall project schedule, something a customer could be interested in?

2. **What about changing interfaces to co- or subprojects?** If you welcome change all the time you will have to change interfaces all the time. This is fine as long as you change internal interfaces. As soon as you have to change the published external interfaces you are in trouble. Bob Wyman, a former Senior Product Manager for Applications Programmability at Microsoft, pointed out: “Even if you're not publishing to millions of programmers, you may still be defining interfaces that will be used outside your organizational scope (perhaps by dozens or hundreds of other programmers) and the managers of those other projects will not tolerate your screwing up their projects because you wanted to make the interfaces ‘better’.”

3. **What about non-functional requirements, like performance or security?** One cannot arbitrarily change non-functional requirements in the course of a project. Performance or security have to be analysed and designed before implementation, at least to a certain degree. Functional and non-functional requirements influence each other, which limits the amount of change your system can bear.

4. **What about distributed development settings?** The “one room setting” is supposed to be the ideal Extreme Programming environment. It is common sense that high levels of noise and frequent interruptions that are the consequence of such a setting lead to a decrease in quality. There is credible empirical evidence supporting this [DeMarco97Peoplewarbe].Close to all of the larger companies that I know subcontract parts of their development to remote locations, which makes such an environment impossible. There are certainly people courageous enough to try Extreme Programming in such settings, although with limited success [Eckstein 01 Resistance].

5. **What about integration of COTS?** Most of the software systems today are not developed from scratch anymore. Commercial of the shelf software (COTS) products are integrated during the development effort. Extreme programming gives little advice on how to handle those situations.

6. **What about large projects?** If typical beginner’s mistakes in software development projects with less then 10 person years of effort are avoided, the results are quite satisfying. The real challenges are with large-scale projects that experience substantial diseconomies of scales and get cancelled frequently.

7 Alternatives

Following my criticism I was routinely asked what I would propose as alternatives to Extreme Programming practices. Here are some answers:

- In contrast to Pair Programming, I see much more sense in practicing what I call “Mutual Programming”: A pair of developers does mutual QA on each others work products: If one partner is finished with coding the other partner inspects and tests the code and vice versa. Compared to Pair Programming there are several advantages, such as applicability in remote settings, measurement of effectiveness, and better ergonomics.

- In contrast to launching a project without a reasonable amount of requirements, I recommend sound prototyping as a cost effective way to investigate on functional and non-functional requirements of the system, as well as possible solutions. Requirements turnover cannot be avoided but it needs to be managed.

© 2002 AVOCA GmbH

- In contrast to “no-documentation” we have to think about how we can automate more of the documentation task and thereby avoid the routinely experienced inconsistencies. Work product documentation with XML is a promising approach to achieve this.

- In contrast to hoping that somebody capable of filling the customer role will appear on the horizon, how about setting up a concise requirements specification, such as the one proposed by [Robertson99MastReq]. This specification may not be complete and might not get completed until the end of the project. If a critical mass of requirements is there it serves its purpose.

- In contrast to relying on testing alone, establish a sound review or inspection process for your work products. Reviews and inspections have repeatedly shown to be more cost effective than testing.

You will find a wealth of useful Software Engineering knowledge in the SW-CMM and its successor the CMMI-SE/SW [Keefer01CMMI]. There is no need to practices all of it in each and every project. Use this material as a reference and adapt it to the specific needs.

What to do about large projects?
A first recommendation is to read [Brooks95Mythical] and avoid them whenever you can. A second recommendation is to look after the CMM Level 2 and 3 key process areas. Particularly Requirements Management, Configuration Management and Project Planning. Implementing those KPA results in a development task that has lost the constant flux that makes software difficult to handle in a traditional engineering way and the unbound temptation for change -a major source of trouble- is limited. A sound risk management process such as the one I describe in [Keefer01Risk] is also mandatory for large projects.
On top of this I recommend something I have been promoting for half a decade with not much success. The idea is pretty old and has been written down around 500 BCE:

The Book of Army Management says: On the field of battle, the spoken word does not carry far enough: hence the institution of gongs and drums. Nor can ordinary objects be seen clearly enough: hence the institution of banners and flags.

Gongs and drums, banners and flags, are means whereby the ears and eyes of the host may be focused on one particular point.

The host thus forming a single united body, is it impossible either for the brave to advance alone, or for the cowardly to retreat alone. This is the art of handling large masses of men.

Sun Tzu, The Art of War

In large software projects we routinely experience a dispersion of views. In traditional engineering projects the visibility of the product under development, such as a bridge or building, allows for the continuous re-focusing and congruence of the participants views, a very powerful passive way of Quality Management.
In order to transfer this beneficial situation from traditional engineering fields to software engineering, we must seek for key metrics of the project and constantly visualize them to the stakeholders involved. This will give our software a face. A beautiful one if we are doing well and an ugly one if we are

© 2002 AVOCA GmbH

performing badly. One rather simple example of this idea employed at Hewlett-Packard can be found at [Pearse99Endgame].

8 Conclusions

As a victim of the Byzantine German tax system, that I consider to be one of the most complicated on the planet, I am critical about rules and bureaucracy that routinely get bypassed by some privileged few. I certainly know about the shortcomings of formal software development approaches, such as the CMM/CMMI or ISO. However, in order to successfully develop software we have to limit the degrees of freedom our nice languages, tools, and personalities offer and select from all the practices and techniques available and formal quality models are useful guides.
I used to participate in competition swimming and consequently one of my favourite spare time activities is going for a swim at the local pool. It is much more convenient to train when lanes are in place which limit the degrees of freedom, but increase the total capacity of the pool by putting some order in place. A software process should do the same: avoid collisions and still let individuals perform.

There is no denial to the fact that certain individuals and teams are capable of producing quality software without a solid process or explicit quality assurance. However, this is not the norm. General recommendations have to target what can be expected on average and on average teams will not be able to reliably develop software using the twelve Extreme Programming practices alone. I was pretty much amazed as one of the Extreme Programming experts explained how to prioritise customer requirements. The description was about six pages long. It indicates that in real-life the explicitly stated simple practices are heavily supported by sophisticated, implicit knowledge and wisdom. Good for you if this knowledge is at hand, in all other cases you are lost.
If your project is small, fast delivery of "something running" is top priority, your business environment is volatile, the professional level of your developers is high, and the technical constraints are rather shallow, Extreme Programming might be an option for you. In most other cases running into a silver bullet that is likely to kill your common sense approach to software development will seriously hurt you.

On the third of February 2001 a tourist got caught in an avalanche in the Swiss Alps near the village of Ayer. In the rescue effort that followed six people of the rescue team were caught in a second avalanche and two of them died. Asked about what to change in the future a senior member of the rescue team later recommended to hold a short meeting before the start of each rescue mission that would define the course and the limits of action for the respective accident.
Considering the fact that the survival rate of avalanche victims declines rapidly with progressing time [Tschirky00Avalanche], it seems that even if time is by far the most dominant factor for your project and the environment is highly volatile, you should plan thoroughly.

9 Acknowledgements

I would like to thank David Lightstone for his highly appreciated input, as well as Hanna Lubecka and Ivaylo Tzotchev for their valuable review of this document.

“Experience is a dear teacher,
but fools will learn at no other.”

Benjamin Franklin

10 References

[Beck98C3]
K. Beck, et. al., 1998, *Chrysler goes to "Extremes"*, Distributed Computing, 10/1998.

[BeckIEEE99Embrace]
K. Beck, 1999, *Embracing Change with Extreme Programming*, IEEE Computer, 10/1999.

[Beck00XP]
K. Beck, 2000, *Extreme Programming Explained*, Addison-Wesley.

[Beck01PlanningXP]
K. Beck, 2001, *Planning Extreme Programming,* Addison-Wesley.

[Beizer90STT]
B. Beizer, 1990, *Software Testing Techniques,* Thomson Computer Press.

[Brooks95Mythical]
F. Brooks, 1995, *The Mythical Man-Month,* Position Paper, Addison Wesley.

[Cockburn01Select]
A. Cockburn, 2000, *Selecting a Project's Methodology,* IEEE Software, 04/2000.

[Crispin01Testing]
L. Crispin, 2000, Extreme Rules of the Road*,* STQE magazine 04/2000.

[DeMarco97Peopleware]
T. DeMarco, T. Lister, 1997, *Peopleware,* Dorset House.

[Dittert01Pair]
K. Dittert, *XP und "Pair Programming"*, OBJEKTspektrum 04/2001.

[Eckstein 01 Resistance]
J. Eckstein, 2001, *Resistance to change: Issues with early and often delivery?,* Position Paper, OOPSLA 2001.

[Grenning01Launch]
J. Grenning, *Launching Extreme Programming at a Process-Intensive Company,* IEEE 06/2001.

[Griffin01Toxic]
A. Griffin, 2001, *Managing Problem People in XP Implementation,* XP Universe 2001.

[Gutknecht00Wirth]
J. Gutknecht, et al., 2000, *The school of Niklaus Wirth: the art of simplicity*, dpunkt Verlag.

[Haungs01Pair]
J. Haungs, 2001, *Pair Programming on the C3 project,* IEEE Computer, 02/2001.

[Hendrickson01Customer]
C. Hendrickson, 2001, *Will Extreme Programming kill your customer?,* Position Paper, OOPSLA 2001.

[Jeffries99ET]
R. Jeffries, 1999, *Extreme Testing*, STQE Distributed Computing, 2/1999.

© 2002 AVOCA GmbH

[Jeffries00Installed]
R. Jeffries, A. Anderson, C. Hendrickson, K. Beck, 2000, *Extreme Programming Installed*, Addison Wesley.

[Keefer01CMMI]
G. Keefer, 2002, The CMMI in 45 Minutes, Technical Paper.

[Keefer01Risk]
G. Keefer, 2002, A CMMI Compatible Risk Management Process, PSQT 2002 Presentation.

[Martin00Pair].
R. C. Martin, *Interview with Robert C. Martin*, ObjectView, Issue 4.

[Myers79AST]
G. J. Myers, 1979, *The Art of Software Testing,* John Wiley & Sons.

[Nawrocki01Pair]
J. Nawrocki, A. Wojciechowski, *Experimental Evaluation of Pair Programming,* ESCOM 2001 Paper.

[Nosek98Pair]
J. T. Nosek, *The Case for Collaborative Programming*, Communications of the ACM 04/1998.

[Oppenheim92Edu]
Alan V. Oppenheim, *A Personal View of Education,* IEEE Signal Processing 04/1992.

[Pearse99Endgame]
T. Pearse, T, Freeman, P. Oman, 1999, *Using Metrics to Manage the End Game of a Software Project,* Proceedings of the Sixth International Software Metrics Symposium.

[Robertson99MastReq]
J. and S. Robertson, 1999, *Mastering the Requirements Process,* Addison-Wesley.

[Schuh01Recovery]
P. Schuh, *Recovery, Redemption and Extreme Programming,* IEEE Software 06/2001.

[Tschirky00Avalanche]
F. Tschirky, B. Brabec, M. Kern, 2000, *Avalanche Rescue Systems in Switzerland: Experience and Limitations*, Eidg. Institut für Schnee- und Lawinenforschung.

[Williams00Diss]
L. Williams, 2000, *The Collaborative Software Process,* PhD Dissertation.

[Williams00Strengthening]
L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries, 2000, *Strengthening the Case for Pair Programming,* IEEE Software 4/2000.

[Williams00Classroom]
L. Williams, R. R. Kessler, *Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom,* Journal on Software Engineering Education, 12/2000

[Williams00Benefits]
L. Williams, A. Cockburn, *The Costs and Benefits of Pair Programming,* XP2000 Conference.

[Williams01Student]
L. Williams, R. L. Upchurch, *In Support of Student Pair-Programming,* SIGCSE Conference 2001.

© 2002 AVOCA GmbH

11 Additional Information on References

Reference	Projects/Experiment Information	Organisation(s)
[Beck98C3]	C3	FirstClass Software
http://www.xprogramming.com/publications/dc9810cs.pdf		
[BeckIEEE99Embrace]	Acxiom, C3, Tariff System, VCAPS	FirstClass Software
http://www.cs.tamu.edu/course-info/cpsc609/spring98/lively/rx070.pdf		
[Beck00XP]	C3	FirstClass Software
[Beizer90STT]	-	Analysis
[Brooks95Mythical]	OS/360	Univ. of NC, Chapel Hill
[Cockburn01Select]	C3 and others (non-XP)	Humans and Technology
http://www.eee.metu.edu.tr/~bilgen/Cockburn647.pdf		
[Crispin01Testing]	-	Agile Development LLC.
http://www.testing.com/agile/crispin-xp-article.pdf		
[DeMarco97Peopleware]	-	Atlantic System Guild
[Dittert01Pair]	Finance Industry	OOcon Informatik
http://www.oocon.de/xp_erfahrungen.pdf		
[Eckstein 01 Resistance]	Distributed Development	Objects in Action
http://www.coldewey.com/publikationen/conferences/oopsla2001/agileWorkshop/eckstein.pdf		
[Grenning01Launch]	DPS	ObjectMentor
http://www.objectmentor.com/resources/articles/UsingXPBigProCo.pdf		
[Griffin01Toxic]	-	Escrow.com
http://groups.yahoo.com/group/extremeprogramming/files/xpproblems.html		
[Griffin01CMMLevel4]	-	Escrow.com
http://www.xpuniverse.com/2001/pdfs/XPU02.pdf		
[Gutknecht00Wirth]	-	ETH Zurich
[Haungs01Pair]	C3	Oracle
[Hendrickson01Customer]	C3	ThoughtWorks
http://www.coldewey.com/publikationen/conferences/oopsla2001/agileWorkshop/hendrickson.html		
[Jeffries99ET]	C3	ObjectMentor
http://www.xprogramming.com/publications/SP99%20Extreme%20for%20Web.pdf		
[Jeffries00Installed]		ObjectMentor
[Keefer01CMMI]	-	AVOCA
http://www.avoca-vsm.com/downloads/downloads.html		
[Keefer01Risk]	-	AVOCA
http://www.avoca-vsm.com/downloads/downloads.html		
[Nawrocki01Pair]	Student Experiment 1999/2000	Poznan Univ. of Technology
http://dialspace.dial.pipex.com/town/drive/gcd54/conference2001/papers/nawrocki.pdf		
[Nosek98Pair]	45 Minute Task	Temple Univ., Philadelphia
[Oppenheim92Edu]	Wall Painting	MIT
http://allegro.mit.edu/dspg/education.pdf		
[Pearse99Endgame]	HP LaserJet Firmware	Hewlett-Packard
[Robertson99MastReq]	-	Atlantic System Guild
http://www.systemsguild.com/GuildSite/Robs/Template.html		
[Schuh01Recovery]	3-Tier Web Project	ThoughtWorks
Re-publication of: http://www.xp2001.org/xp2001/conference/papers/Chapter7-Schuh.pdf		

[Tschirky00Avalanche]	-	FISAR
http://www.slf.ch/info/unfallstatistik-en.pdf		
[Williams00Diss]	C3, Student Experiment Fall 1999	Univ. of Utah
http://www.cs.utah.edu/~lwilliam/Papers/dissertation.pdf		
[Williams00Strengthening]	C3, Student Experiment Fall 1999	Univ. of Utah, ObjectMentor
http://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.PDF		
[Williams00Classroom]	C3, Student Experiment Fall 1999	Univ. of Utah
http://collaboration.csc.ncsu.edu/laurie/Papers/csed.pdf		
[Williams00Benefits]	Student Experiment Fall 1999	Univ. of Utah, Humans and Technology
http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF		
[Williams01Student]	Student Experiment Fall 1999	Univ. of Utah, Univ. of Massachusetts
http://collaboration.csc.ncsu.edu/laurie/Papers/WilliamsUpchurch.pdf		

Links current February, 1st 2002.

12 Contact Data

AVOCA GmbH
Kronenstr. 19,
D-70173 Stuttgart
Germany
Phone: +49 711 22 71 374
Fax: +49 711 22 71 375
E-mail: gkeefer@avoca-vsm.com
http://www.avoca-vsm.com

© 2002 AVOCA GmbH