eXtreme Programming

CEPIS

(Council of European Professional Societies)
unites
32 Informatics Professional Societies
across Europe
and is the voice of more than
150,000 ICT Professionals in our continent
<http://www.cepis.org>

Coming issue:
"Information Retrieval"

eXtreme Programming
Guest Editor: Luis Fernández Sanz

Joint issue with NOVÁTICA and INFORMATIK/INFORMATIQUE

Presentation: eXtreme Programming

Luis Fernández Sanz, Guest Editor

This issue is focused on XP (eXtreme programming), one of the recent proposals in the software development field that has achieved a really important media impact among software practitioners. As a new way of improving the agility of software projects, XP relies on several principles (automated testing, pair development, etc.) that shorten the project life cycle between releases. But these principles are also devised to obtain a general improvement of software quality and user satisfaction, avoiding problems due to delays and exceeding budget. All these promises have raised a general interest in this development philosophy. Trying to satisfy the curiosity of our readers, we have decided to publish an interesting set of paper intended to contribute to a deeper understanding of XP, the pros and the cons.

"Extreme programming: a new software development method" was presented in the Sixth Software Quality and Innovation Spanish Conference (2001) organised by the Software Quality Group of ATI. In this paper, a brief description of the main characteristics and principles of XP is included. This contribution helps the reader to know the fundamentals of Extreme Programming.

"Programming extremism", by M.McCormick, is a keen analysis of the implications of XP in the field of software processes. His observations about the two antagonist communities in software engineering (and the need of a third party of pragmatists) and how the beliefs of each one interfere in their objective perception of which is the best solution for each type of project and organisation are really interesting. The necessary reference to CMM (and other "formal" models of software process improvement and evaluation) is included.

"The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment" (L.Crispin, T.House and C.Wade) presents details of one of the more robust contributions of XP: automated testing. Instead of the usual unconcerning in testing that can be observed in the traditional software organisations, XP stresses the importance of a proper and efficient testing practice. As well as a brief review and discussion of XP testing principles (in a Q&A format), this paper presents details about a practical experience using JUnit and other "tools". This paper was presented in the XP 2001 conference (thanks to both the negotiation of Michele Marchesi, co-chairman of the Conference, and the collaboration of the authors).

"Qualitative Studies of XP in a Medium Sized Business" is another paper from XP2001 (thanks again to Marchesi and the authors). The paper examines the benefits of a flexible management approach to XP methodology. Presentation and discussion of results of an empirical study using qualitative research techniques (questionnaires, direct observation, etc.) are included as part of a good review of situations that emerge when people try to apply XP in a real organisation.

"XP and software engineering" presents my own analysis of XP. From the perspective of somebody who has to approach extreme programming from the "outer world", this paper is focused on several important improvement proposals. Of course, the doubts of the author about the success of some principles of XP related to classical software engineering practices (e.g. configuration management vs common property of code) are also included.

One of the main advantages of XP is the agility of the proposed software process and its direct application to small projects with a high rate of requirements volatility. But, is XP suitable for larger or more complex projects? To answer this question, M.Lippert, S. Roock, H. Wolf and H. Züllighoven present an extension of the roles of client representatives and the creation of new document types to address the subsequent project situation.

I hope this variety of contributions would satisfy the increasing demand of information about XP of software practitioners (in general sense) and our readers as the persons who we detove our work to.

Luis Fernández Sanz received a degree in informatics engineering from Technical University of Madrid (Spain) in 1989 and a Ph. D. degree in informatics from University of the Basque Country in 1997 (as well as an extraordinary mention for his doctoral thesis). He is currently head of the department of programming and software engineering at Universidad Europea-CEES (Madrid). From 1992, he is the coordinator of the software engineering section of Nováutica. He is author or coauthor of several books about software engineering and software measurement, as well as different papers in international journals and conferences. He is member of the Software Quality Group of ATI and he has acted as chair of the VI Spanish Conference on Software Quality and Innovation organised by ATI. He is a member of ATI and the Computer Society of the IEEE.
<lufern@dpris.esi.uem.es>

Useful references on eXtreme Programming

Note: See also the references included in the papers published in this issue.

Books

K. Beck: Extremme programming. Embrace change, Addison-Wesley, 2000.

K. Beck & M. Fowler: Planning extreme programming, Addison-Wesley, 2000.

R. Jeffries, A. Anderson, C. Hendrickson, K. Beck, R. E. Jeffries: Extreme Programming Installed, Addison-Wesley, 2000

G. Succi, M.Marchesi: Extreme Programming Examined, Addison-Wesley, 2001.

W. C. Wake: Extreme Programming Explored, Addison-Wesley, 2001.

R. Hightower & N. Lesiecki: Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus, John Wiley & Sons, 2001.

K. Auer, R. Miller: Extreme Programming Applied: Playing to Win, Addison-Wesley, 2001.

M. Fowler, K. Beck, J. Brant, W. Opdyke & D. Orberts: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

Conferences

Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, May 26-29, 2002, Alghero, Sardinia, Italy. <http://www.xp2002.org>.

XP Agile Universe, August 4-7, 2002, Chicago, Illinois, USA. http://www.xpuniverse.com.

Web sites

Xprogramming, an Extreme Programming Resource.
<http://www.xprogramming.com/>

Extreme Programming: A gentle introduction
. <http://www.extremeprogramming.org/>

PortlandPatternRepository and WikiWikiWeb:
<http://c2.com/cgi/wiki>

XP Developer:
<http://www.xpdeveloper.com/>

JUnit and other XUnit testing frameworks.
<http://www.junit.org/>

eXtreme Programming.
<http://www.armaties.com/extreme.htm>

Pair programming.
<http://pairprogramming.com/>

XP123 - Exploring Extreme Programming.
http://xp123.com/

Some important recent articles

J. Highsmith, A. Cockburn, Agile Software Development: The Business of Innovation, IEEE Computer, September, 2001, pp. 120-122.

This article include a brief explanation o f characteristics and advantages of agile software development approaches (not only XP). Moreover, it include a manifesto where representative persons related to agile development software methods state their compromise with "individuals, working software, customer collaboration and responding to change" as a clear guide for overcoming current problems in software development. The importance of this article, in my opinion, does not reside in its content but in the reaction that it has provoked in the IEEE Computer Magazine Community. For example:

• A letter from S. Rakitin was published in December (Letters, "Manifesto elicits cynicism", IEEE Computer, December, 2001, pp. 4-7). In it, the author strongly disagreed with the point of view of Highsmith and Cockburn.

• An article by the well-known B. W. Boehm (B. W. Boehm, "Get Ready for Agile Methods, with care", January, pp. 64-69) tried to balance the advantages of agile processes and several important preventive measures that should be taken in consideration in order to avoid "overresponding to change".

• Two recent letters (Letters, "Professional Approach to Software Development" and "Unavoidable statistics", IEEE Computer, March, 2002, pp.6-7) in the issue of March 2002 make interesting contributions to the point of view expressed by Rakitin and Boehm.

As anyone can see, XP is a really "hot spot" for the software development community.

A new method of Software Development: eXtreme Programming

César F. Acebal and Juan M. Cueva Lovelle

What is eXtreme Programming – also known as XP? The aim of this article is to answer that question, and to reveal the nature of this new method of software development to the uninitiated reader. Naturally the length of any technical article does not permit more than a brief introduction to any new method or technique, but we will try to be sufficiently informative so that you will all come away with some idea of the basic underlying principles, and for anyone who might want to delve deeper into the subject, we will provide suitable references.

Keywords: eXtreme Programming, XP, Software Development.

1 Introduction

XP is a new software development discipline which, amid much fanfare, has recently joined the welter of methods, techniques and methodologies that already exist. To be more precise: this is a lightweight method, as opposed to heavyweight methods like Métrica. Before we go on, we'd like to make a clarification: in this article we refer to XP as a "method", contrary to the official IT tendency to apply the term "methodology" (science of methods) to what are no more than methods[1] or even mere graphic notations.

It could be said that XP was born "officially" five years ago in a project developed by *Kent Beck* at *Daimler Chrysler*, after he had worked for several years with *Ward Cunningham* in search of a new approach to the problem of software development which would make things simpler than the existing methods we were used to. For many people, XP is nothing more than common sense. Why then does it arouse such controversy, and why do some people adore it while others heap scorn on it? As Kent Beck suggests in his book [Beck 00], maybe it's because XP carries a series of common sense techniques and principles to extreme lengths. The most important of these techniques are:

- Code is constantly reviewed, by means of *pair programming* (two people per machine)
- Tests are made all the time, not only after every class (*unit tests*), but also by customers who need to check that the project is fulfilling their requirements (*functional tests*)
- Integration tests are always carried out before adding any new class to the project, or after modifying any existing one (*continuous integration*), making use of *testing frameworks*, such as *xUnit*
- We (re)design all the time (*refactoring*), always leaving code in the simplest possible state

---

1. Ricardo Devis Botella. C++. STL, *Plantillas, Excepciones, Roles y Objetos* (*Templates, Exceptions, Roles and Objects*). Paraninfo, 1997. ISBN 84-283-2362-3

- Iterations are radically shorter than is usual in other methods, so that we can benefit from feedback as often as possible.

By way of summary, and to wrap up this section and to get down to the nitty-gritty, I will leave you with this quote from Beck's aforementioned book.

"Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team members change. The problem isn't change, per se, because change is going to happen; the problem, rather, is the inability to cope with change when it comes."

2 The four variables

XP sets out four variables for any software project: *cost*, *time*, *quality* and *scope*.

It also specifies that of these four variables, only three of them can be established by parties outside the project (custom-

César Fernández Acebal received a degree in informatics engineering from Oviedo University. He worked as a teacher in Java and Web programming for students of higher professional education. Afterwards, he worked as technical director in a web site development company. He has combined these positions with a continuous educational activity related to Java, XML, Web development, etc. He is currently an IT architect of B2B 2000, an e-business company. His research interests include object-oriented programming and software engineering and agile software processes. He is a member of ATI, IEEE, Computer Society, ACM, etc. <acebal@ieee.org>

Juan Manuel Cueva Lovelle is a mining engineer from Oviedo Mining Engineers Technical School in 1983 (Oviedo University). He has the Ph. D. from Technical University of Madrid in 1990. From 1985 he is Professor at the Languages and Computers Systems Area in Oviedo University. ACM and IEEE voting member. His research interests include Object-Oriented technology, Language Processors, Human-Computer Interface, Object-Oriented Databases, Web Engineering, Object-Oriented Languages Design, Object-Oriented Programming Methodology, XML, WAP, Modelling Software with UML and Geographical Information Systems. <cueva@lsi.uniovi.es>

ers and project managers), while the value of the free variable will be established by the development team in accordance with the other three values. What is new about this? It's that normally customers and project managers considered it their job to pre-establish the value of *all* the variables: *"I want these requirements fulfilled by the first of next month, and you have this team to work with. Oh, and you know that quality is the number one priority!"*

Of course when this happens – and unfortunately it happens quite often – quality is the first thing to go out of the window. And this happens for a simple reason which is frequently ignored: no one is able to work well when they are put under a lot of pressure.

XP makes the four variables visible to everyone – programmers, customers and project managers –, so that the initial values can be juggled until the fourth value satisfies everybody (naturally, with the possibility of choosing different variables to control).

Also the four variables do not in fact bear such a close relation with one another as people often like to think. There is a well known saying that "nine women cannot make a baby in one month" which is applicable here. XP puts special stress on small development teams (ten or twelve people at most) which naturally can be increased if necessary, but not before, or the result will generally be the opposite of what was intended. However, a number of project managers seem to be unaware of this when they declare, puffed up with pride, that their project involves 150 people, as if it were a mark of prestige, something to add to their CV. It is good, however, to increase the cost of the project in matters such as faster machines, more specialists in certain areas or better offices for the development team.

With *quality* too, another strange phenomenon occurs: often, *increasing the quality means the project can be completed in less time*. The fact is that as soon as the development team gets used to doing intensive tests (and we will be coming to this point soon, as it's the corner stone of XP) and coding standards are being followed, gradually the project will start to progress much faster than it did before. The project's quality will still remain 100% assured – thanks to the tests – which in turn will instil greater confidence in the code and, therefore, greater ease in coping with change, without stress, and that will make people programme much faster… and so on.

The other face of the coin is the temptation to sacrifice the internal quality of the project – that which is perceived by the programmers – to reduce the delivery time of the project, trusting that the external quality – that which the customers perceive – will not be affected too greatly. However, this is a very short term bet, which tends to be an invitation for disaster, since it ignores the basic fact that *everyone works better when they are allowed to do a quality job*. Ignoring this will cause the team to get demoralised and, in the long term, the project will slow down, and much more *time* will be lost than ever could have been hoped to be saved by cutting down on quality.

With regard to the project's *scope*, it is a good idea to let this be the free variable, so that once the other three variables have been established, the development team should decide on the scope by means of:
- The estimation of the tasks to perform to satisfy the customer's requirements.
- The implementation of the most important requirements first, so that at any given time the project has as much functionality as possible.

3 The cost of change

Although we cannot go into any great depth on this subject here, we believe it is important to at least mention one of the most important and innovative suppositions that XP makes in contrast to most known methods. We are referring to the cost of change. It has always been considered a universal truth that the cost of change in the development of a project increased exponentially in time, as shown in figure 1.

XP claims that this curve is no longer valid, and that with a combination of good programming practices and technology it is possible to reverse the curve, as we show in figure 2.

Naturally, not everyone agrees with this supposition (and in Ron Jeffries web site [Jeffries] you can read several opinions to this effect). But in any event it is clear that if we decide to use XP as a software development process we should accept that this curve is valid.

The basic idea here is that instead of changing for change's sake, we will design as simply as possible, to do only what is absolutely necessary at any given moment, since the very simplicity of the code, together with our knowledge of refactoring [Fowler 99] and, above all, the testing and continuous integration, all mean that changes can be carried out as often as necessary.

4 Practices

But let's get down to brass tacks. What does XP really entail? What exactly are these practices we have been referring to, which are able to bring about this change of mentality when it comes to developing software? Trusting that you, the reader, will excuse the enforced brevity of our explanation we will now give a brief description of these practices.
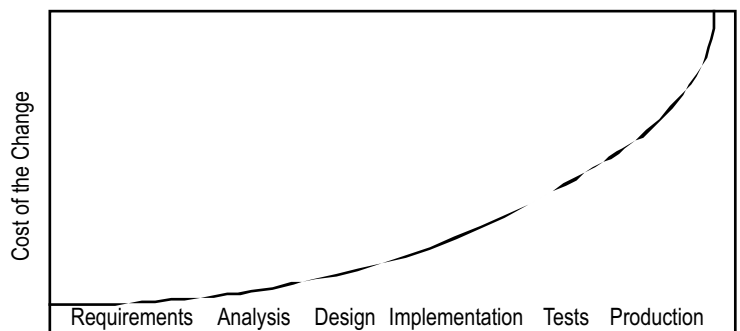

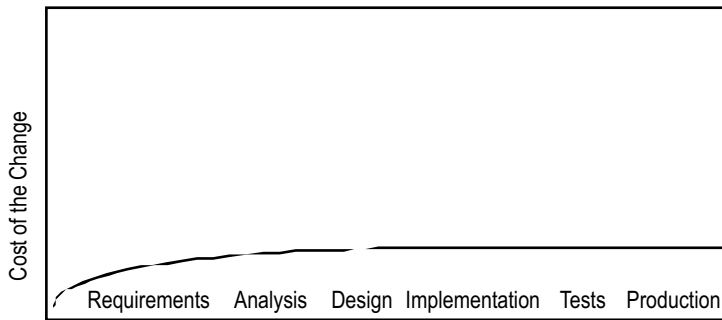**Figure 1:** Cost of change in "traditional" software engineering

Figure 2: Cost of change in XP

Planning

XP sees planning as a permanent dialogue between the business and technical parties involved in the project, in which the former will decide the scope – what is really essential for the project –, the priority – what should be done first –, the composition of the releases – what should be included in each one – and the deadline for these releases.

The technical people, for their part, are responsible for estimating the time needed to implement the functionalities which the customer requires, for reporting on the consequences of decisions taken, for organising the work culture and, finally, for carrying out a detailed planning of each release.

Small releases

The system first gets into production just a few months at most before it is completely finished. Successive releases will be more frequent –at intervals of between a day and a month–. The customer and the development team will benefit from the feedback produced by a working system and this will be reflected in successive releases.

Simple design

Instead of being hell bent on producing a design which requires the gift of clairvoyance to develop, what XP advocates is, at any given moment, that we should design for the needs of the present.

Testing

Any feature of a programme for which there is not an automated test simply does not exist. This is undoubtedly the cornerstone on which XP is built. Other principles are liable to be adapted to the characteristics of the project, the organisation, the development team… But on this one point there is no argument: if we aren't doing tests, we aren't doing XP. We should be using some automated testing framework to do this, such as JUnit [JUnit] or any of its versions for different languages.

Not only that, but we will write the tests even before we write the class which is to be tested. This is an aid to following the principle of *programming by intention*, that is, writing code as if the most expensive methods had already been written, and so we only had to send the corresponding message, in such a way that the code will be a true reflection of its intention and will document itself. On JUnit's web site, mentioned in the previous paragraph, you can find interesting articles which explain how these tests should be written.

Refactoring

This responds to the principle of simplicity and basically consists of leaving the existing code in the simplest possible state, so that no functionality is lost – or gained – and all tests continue to be carried out correctly. This will make us feel more comfortable with the code already written and therefore less reluctant to modify it when some feature has to be added or changed. In the case of legacy systems, or projects taken over after they have already been started, we would be bound to need to devote several weeks just to refactoring the code – which tends to be a source of tension with the project managers involved when they are told that the project is going to be held up for several days "just" to modify existing code, which works, without adding any new functionality to it.

Pair programming

All code will be developed in pairs – two people sharing a single monitor and keyboard. The person writing the code should be thinking about the best way to implement a particular method, while his colleague will do the same, but from a more strategic viewpoint:

• Are we going about this in the right way?
• What could go wrong here? What should we be checking in the tests?
• Is there any way to simplify the system?

Of course, the roles are interchangeable, so that at any moment the person observing could take over the keyboard to demonstrate some idea or simply to relieve his colleague. Similarly the composition of the pairs could change whenever one of them were required by some other member of the team to lend a hand with their code.

Collective property of the code

Anyone can modify any part of the code, at any time. In fact, any one who spots an opportunity to simplify, by refactoring, any class or any method, regardless of whether they have written them or not, should not hesitate to do so. This is not a problem in XP, thanks to the use of coding standards and the assurance that testing gives us that everything is going to carry on working well after a modification.

Continuous integration

Every few hours – or at the very least at the end of a day's programming – the complete system is integrated. For this purpose there is what is known as an integration machine, which a pair of programmers will go to whenever they have a class which has passed a unit test. If after adding the new class together with its unit tests the complete system continues to function correctly – i.e. passes all the tests –, the programmers will consider this task as completed. Otherwise they will be responsible for returning the system to a state in which all tests function at 100%. If after a certain time they are unable to
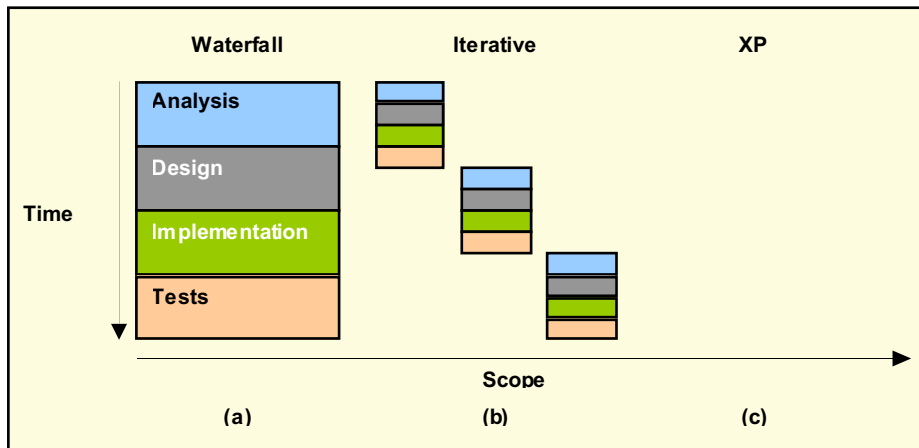
Figure 3: Comparison of long development cycles: (a) the waterfall model (b) shorter iterative cycles, for example, the spiral model and (c) the mix of all these activities which XP employs throughout the whole software development process

discover what it is wrong, they will bin the code and start over again.

40 hours a weeks

If we really want to offer quality, and not merely a system that works – which we all know, in IT, is trivial[2] – we will want each member of our team to get up each morning rested and to go home at 6 in the evening tired but with the satisfaction of a job well done, and that when Friday comes around he or she can look forward to two days' rest to devote to things which have nothing whatsoever to do with work. Naturally it doesn't have to be 40 hours – it could be anything between 35 and 45, but one thing is certain: nobody is capable of producing quality work 60 hours a week.

Customer on site

Another controversial XP rule: at least one real customer should be permanently available to the development team to answer any question the programmers may have for them, to establish priorities… If the customer argues that their time is too valuable, we should realise that the project we have been given is so trivial that they do not consider it worthy of their attention, and that they don't mind if it is based on suppositions made by programmers who know little or nothing of the customer's real business.

Coding standards

These are essential to the success of collective property of the code. This would be unthinkable without a coding based on standards which allow everyone to feel comfortable with code written by any other member of the team.

---

2. Ricardo Devis Botella. *Curso de Experto Universitario en Integración de Aplicaciones Internet mediante Java y XML*. (University Expert Course in the Integration of Internet Applications via Java and XML) University of Oviedo, 2000.

5 Planning

While XP is a code centred method it is not just that. It is also above all a software project management method, in spite of the criticism levelled by many people, perhaps after a too hasty reading of an article such as this one. But for anyone who has taken the trouble to read any of the books explaining the process, it will be clear that planning makes up a fundamental part of XP. The thing is that, given that software development, like almost everything in this life, is a chaotic process, XP does not attempt to find a non-existent determinism but rather provides the means necessary to cope with that complexity, and accepts it, without trying to force it into constraints of heavyweight or bureaucratic methods. We wholeheartedly recommended you to read Antonio Escohotado's gentle introduction to chaos theory [Escohotado 99], which we believe has a lot to do with the idea behind XP. In short, lightweight methods – and XP numbers among them – are adaptive rather than predictive [Fowler].

The life cycle

If, as has been demonstrated, long development cycles of traditional methods are unable to cope with change, perhaps what we should do is make development cycles shorter. This is another of XP's central ideas. Let's take a look at a chart which compares the waterfall model, the spiral model and XP:

6 Conclusions

As we said at the beginning article, XP, just one year after the publication of the first book on the subject, has caused a great furore among the software engineering community. The results of the survey below, commissioned by IBM, evidences the fact that opinions on the subject are divided [IBM 00]:

Pair programming comes in for some especially strong criticism – above all from project managers, though it is an opinion which is doubtless shared by many programmers with an overdeveloped sense of ownership regarding code ("I did this, and what's more I am so good at programming and I have such a command of the language's idioms that only I can understand it"), but a lot is also said about the myth of the 40 hour week, that "all this business about tests is all very well if you have plenty of time, but they are an unaffordable luxury under current market conditions"… and many other vigorous criticisms in a similar vein.

There are also people who say, (and this criticism is perhaps more founded than the previous ones) that XP only works with good people, that is, people like Kent Beck, who are able to make a design which is good, simple and, at the same time – and maybe precisely for that reason – easily extendable, right from the outset.[3]
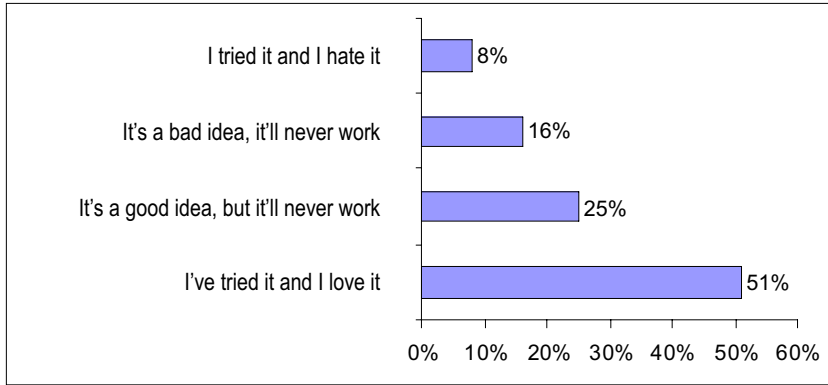
Figure 4: IBM survey (October 2000): What do you think about EXtreme Programming?

One of the things we are trying to say is that XP should not to be misinterpreted due to the inevitable superficiality of articles such as the one you are reading. At the end of the day the creator of this method is not some upstart, but one of the pioneers in the use of software templates, creator of CRC files, author of the *HotDraw* drawing editor framework, and the *xUnit*.testing framework. Were it for no other reason, it would be worth at least taking a look at this new and exciting software development method.

However, none of the practices advocated by XP are an invention of the method; all of them existed before, and what XP has done is to put them all together and prove that they work.

In any event, Beck's first book is a breath of fresh air which should be compulsory reading for any *software engineer* or *software architect*, to use the term preferred by our friend Ricardo Devis, whatever conclusions you may finally draw about XP. At the very least, it's great fun to read.

---

3. Raúl Izquierdo Castanedo. *Comunicación privada*. (*Private communication*)

References

[Beck 99]
    Kent Beck. Embracing Change with eXtreme Programming. Computer (magazine of the IEEE Computer Society).Vol. 32, No. 10. October 1999, pp. 70–77
[Beck 00]
    Kent Beck. eXtreme Programming Explained: Embrace Change. Addison Wesley Longman, 2000. ISBN 201-61641-6
[Beck/Fowler]
    Kent Beck, Martin Fowler. Planning eXtreme Programming. Addison-Wesley. ISBN 0201710919
[Escohotado 99]
    Antonio Escohotado. Caos y orden (Chaos and order). Espasa Calpe, 1999. ISBN 84-239-9751-0. An interesting introduction to chaos theory, which in our view describes the attitude you need to approach software development from an XP point of view.
[Fowler]
    Martín Fowler. The New Methodology. http://www.martinfowler.com/ articles/newMethodology.html
[Fowler 99]
    Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999. ISBN 0201485672
[Jeffries et al. 00]
    Ronald E. Jeffries et al. EXtreme Programming Installed. Addison-Wesley, 2000. ISBN 0201708426
[IBM 00]
    http://www-106.ibm.com/developerworks/java/library/java-poll-results/xp.html. Java poll results: What are your thoughts on EXtreme Programming? IBM survey, October 2000
[IEEE CS]
    http://www.computer.org/seweb/Dynabook/Index.htm. eXtreme Programming. Pros and Cons. What questions remain? The first "dynabook" from the IEEE Computer Society was devoted to XP, with a series of related articles.
[JUnit]
    http://www.junit.org. JUnit, an automated testing framework for Java, adapted from the framework of the same name for Smalltalk, and available in many other languages. These other versions, under the generic name xUnit, are available at the Ron Jeffries' web site [Jeffries], in the software section.
[Jeffries]
    http://www.xprogramming.com. One of the most complete XP portals, by Ron Jeffries.

Programming Extremism

Michael McCormick

The author reviews antecedents and experiences of the "agile" methodology of software development called eXtreme Programming, comparing it to other methodologies and pointing to its advantages and disadvantages from a pragmatic standpoint, depending on the kind of project it applies to. He draws the conclusion that it is necessary to stay away from "religious" positions about existing methodologies.

Keywords: eXtreme Programming, XP, methodologies, Software Engineering

The battle lines are drawn. Hostilities have broken out between armed camps of the software development community. This time the rallying cry is, "XP!"

At recent OOPSLA conferences advocates of Extreme Programming (XP) made themselves conspicuous with their red "I XP – Do You?" badges. Some well-known authors and consultants in the OO community reinvented themselves as preachers of XP; others muttered under their breath about the end of the world.

If you've been in a dark cave (or cubicle) for more than a year, and you somehow haven't heard about XP, it's a lightweight OO development process. Like all good religions, XP is built around a codifiable belief system and a collection of practical techniques. These techniques include small teams, pair programming, JAD with business stories, very short development iterations, automated testing, and discovered design. This

Extreme Programming, a lightweight OO development process, is the latest eruption between programmers and software engineers.

is not a general introduction to the EXP methodology. Those interested should refer to [Beck 99], [Internet].

What XP uncovered (again) is an ancient, sociological San Andreas fault that runs under the software community – programming versus software engineering (a.k.a. the scruffy hackers versus the tweedy computer scientists). XP is only the latest eruption between opposing continents.

Which brings us back to the OOSLA conferences. Imagine you were an anthropologist moving invisibly among the social cliques at OOPSLA. If astute (and politically incorrect) enough to risk some broad stereotypes, you might discern two opposing belief systems (see the table).

If it weren't for Microsoft bashing, what would ever bring these tribes together? They resemble Republicans and Democrats battling ideologies caught up in the divisive dualism of either-or positions on hot-button issues (while the rest of the country rolls its eyes and stays home from the polls).

But, just maybe, the software development world isn't black-and-white. Maybe it's a fuzzy grayscale. For example, there have been projects where "hack (er, prototype) until it works" RAD approaches – some more "extreme" than XP – were successful, even tactically appropriate. I've seen a few. However, in some cases I later saw those same RAD teams try applying their techniques to other kinds of efforts, only to fail disastrously and wonder why.

At the other extreme, I've also watched helplessly while a high ceremony heavyweight process brought an organization of talented, formerly productive software engineers to a dead stop. Crimes were committed in the name of SEI CMM and ISO 9001. Yet, eventually management had to let go their dreams of Malcolm Baldridge awards and let their people get some real work done.

On the other hand, I once had the privilege to observe an organization achieve CMM maturity Level 4[1] certification without the baggage of a productivity-killing, paperwork-clogged high ceremony methodology. Lean, mean … yet mature.

Michael McCormick has worked with information technology in the United States since 1977. He has worked with object-oriented design and development processes since 1991, including many techniques now incorporated in eXtreme Programming. From 1997 to 2000 he was Principal Methodologist in the Object Technology Centre of a Fortune 100 company. Currently he is senior system architect in a large American bank. He is also affiliated with the Software Engineering Institute at Carnegie-Mellon University and the Object Technology User group at Saint Thomas University. In the past year he's been quoted in eWeek, Information Week, and Network World. His article "Programming Extremism" triggered much dialog in America, was cited in an academic paper, and debated at a conference (XP Universe). Michael hopes that constructive XP debate will continue in Europe, and welcomes reader e-mail at m.mccormick@acm.org.

1. On a scale of 1 to 5, as assessed by the SEI. Most (90% by some estimates) development organizations never even reach Level 2.

Observed Belief Patterns in Software Community	
Group "P" Beliefs	Group "S" Beliefs
"Code is easy to change."	"Code is expensive to change."
Likes verbal communication.	Likes written specification.
"The code is the design."	"Code is poor design artifact."
"Good designs emerge."	"Good design comes up front."
"Programmers collaborate."	"Programmers can't communicate."
Codes with peers.	Reviews code for defects.
Informal requirements suffice.	Formal specs and change control.
Loved RAD.	Smugly says, "I warned you!"

An XP evangelist[2] recently accused the Software Engineering Institute of setting back the practice of computer programming. Now certainly CMM has been abused, but this attitude betrays a misunderstanding (and mistrust) of software engineering's goals. The goals are worthy, and (surprise) they can even be implemented with lightweight methodologies where appropriate.

It would be enlightening to conduct a CMM assessment of a team successfully practicing XP. In theory, I see no reason why the XP team should not achieve a maturity level of 2 or better. CMM Level 2 is about managing project requirements and schedules effectively and repeatably. XP claims to do just that, using story cards and a planning game.

On the software engineering side of the fault line, there is an equal amount of misunderstanding and mistrust. Superficially, XP resembles RAD in some respects, and at least as many crimes have been committed in the name of rapid prototyping as in the name of SEI CMM. Yet, as with CMM, in many such cases RAD itself was less to blame for its failures than were the people who misused it. Besides, XP theoretically demands a level of discipline and rigor well above RAD.

It's time to stop the methodology crusades. A one-size-fits-all development process does not exist. Software projects vary wildly in technology, size, complexity, risk, critically, regulatory, and cultural constraints, and many other key variables. Alistair Cockburn[3] has done insightful work mapping out the spectrum of software projects, and the parallel Methodology Space. He argues persuasively that there is a sweet spot where XP will flourish, mainly on smaller, less critical projects.

What's needed is not a single software methodology, but a rich toolkit of process patterns and "methodology components" (deliverables, techniques, process flows, and so forth) along with guidelines for how to plug them together to customize a methodology for any given project. My own work led me away from one-size-fits-all methods, and toward tailorable process frameworks based on proven best practices.

By recognizing each project's unique needs and circumstances, and giving them the flexibility they need to succeed (while balancing their parochial needs against strategic goals of the enterprise to improve reuse, quality, cost, and so forth), we found corporate development guidelines gain more acceptance from development teams. A process rejected by practitioners is doomed to fail. Methodologists, managers, and Software Engineering Process Group police must resist the temptation to blame such rejections on the so-called practitioners, which would be like the Coca-Cola Company blaming consumers for the failure of New Coke.

Even if XP is best suited only to certain projects, it ought to be one of the tools in our bag of tricks. How often (if ever) one actually uses XP (or any other process) becomes a matter of project circumstances, not religious beliefs.

The dream (now misguidedly advocated by some members of the Object Management Group) of a single, standard grand unified process is fool's gold. It would be much more useful to collaborate on a series of process frameworks, or a meta-methodology, based on abstractions of proven process patterns. Under certain project conditions, XP might be an instantiation of the framework. Under other conditions, something like RUP[4] or OPEN[5] might emerge. Robert Martin has even demonstrated that XP can be viewed as a sort of degenerate case of RUP.[6] In the end, we're still left with the underlying social conflict – those scruffy programmers and tweedy software engineers pitted against each other. Even if the XP issue is defused and the current furor subsides, another conflict seems likely to erupt again.

If the opposing groups are like Democrats and Republicans, then maybe what the software development community needs is a Reform Party. Is there a disaffected, apathetic silent majority that is neither scruffy nor tweedy, that isn't violently emotional about methodology? You bet there is. Most of us want tools that work, not religious dogma.

Who speaks for the pragmatists? It's time for a third party in software development politics.

References
[Beck 99]
    Beck, K. The XP Bible Remains Extreme Programming Explained: Embrace Change. Addison-Wesley, Cambridge, Mass., 1999.
[Internet]
    www.extremeprogramming.org; www.XProgramming. com; or /c2.com/cgi/wiki?Extreme-Programming.

---

4. Rational Unified Process, a relatively formal OO development process derived in large measure from Ivar Jacobson's Objectory, but now marketed by Rational Corp. under this some-what misleading name. Recently RUP has moved belatedly toward a flexible framework in an effort to embrace (or co-opt, depending on your point of view) XP (see also footnote 6); www.rational.com/products/rup/index.jsp for more on RUP.

5. The OPEN group, a consortium of companies and individuals, propose an alternative process standard. It rivals RUP, but remedies some of its shortcomings. In its second version, OPEN was already evolving to more of a framework than a fixed process before the advent of RUP or XP; www.open.org.au/.

6. Rational's latest Unified Process is rubbery enough it can be squeezed and stretched to look almost like XP. Robert Martin dubbed his "extreme RUP" dx (read it upside-down); www. objectmentor.com/publications/RUPvsXP.pdf.

---

2. Ron Jeffries, speaking at the Current Object Practices and Experience conference (COPE '99), St. Thomas University, St. Paul MN, 11/16/99.

3. Cockburn's writings on Methodology Space are available from the Humans and Technology Web site (members.aol.com/humansandt/). Especially recommended is his article "One Methodology Per Project."

The Need for Speed:
Automating Acceptance Testing in an eXtreme Programming Environment

Lisa Crispin, Tip House and Carol Wade (Contributor)

In "eXtreme Programming Explained", Kent Beck compares eXtreme Programming to driving a car: the driver needs to steer and make constant corrections to stay on the road. If the XP development team is steering the car, the XP tester is navigating. Someone needs to plot the course, establish the landmarks, keep track of the progress, and perhaps even ask for directions. Acceptance tests must go beyond functionality to determine whether the packages meet goals such as specified performance levels. Automating end-to-end testing from the customer point of view can seem as daunting as driving along the edge of a cliff with no guard rail. At Tensegrent, a software engineering firm in Denver organized around XP practices, the developers and the tester have worked together to design modularized, self-verifying tests that can be quickly developed and easily maintained. This is accomplished through a combination of in-house and vendor-supplied tools.

Keywords: Testing, Automated Testing, Acceptance Testing, Test Scripts, Tester, Test Tools, Web Testing, GUI Testing.

Introduction

The three XP books give detailed explanations of many aspects of the development side of XP. The test engineer coming from a traditional software development environment may not find enough direction on how to effectively automate acceptance tests while keeping up with the fast pace of an XP project. In an XP team, developers are also likely to find themselves automating acceptance tests – an area where they may have little experience. Automating acceptance testing in an XP project may feel like driving down a 12% grade in a VW bug with a speeding semi in the rear-view mirror. Don't worry – like all of XP, it requires courage, but it can – and should – be fun, not scary.

The XP practices we follow at Tensegrent include:
- pair programming
- test first, then code
- do the simplest thing that works (NOT the coolest thing that works!)
- 40-hour week
- refactoring
- coding standards
- small releases
- play the planning game

We apply these same practices to testing – including pair testing.

Lisa Crispin has more than10 years experience in testing and quality assurance, and is currently a Senior Consultant with Bold-Tech Systems (http://www.boldtech.com), working as a tester on Extreme Programming (XP) teams. Her article "Extreme Rules of the Road: How an XP Tester can Steer a Project Toward Success" appeared in the July 2000 issue of STQE Magazine. Her presentation "The Need for Speed: Automating Acceptance Tests in an Extreme Programming Environment" won Best Presentation at Quality Week Europe in 2000. Her papers "Testing in the Fast Lane: Acceptance Test Automation in an Extreme Programming Environment" and "Is Quality Negotiable?" will be published in a collection called Extreme Programming Perspectives from Addison-Wesley. She is co-writing a book Testing for Extreme Programming which will be published by Addison-Wesley in October 2002. Her presentations and seminars in 2001 included "XP Days" in Zurich, Switzerland, XP Universe in Raleigh, and STAR West. Lisa can be contacted at lisa.crispin@att.net.

Tip House is Chief Systems Analyst at the OCLC Online Computer Library Centre Inc., a non-profit organization dedicated to furthering access to world's information, where he develops and supports test automation tools and document management systems for the Web. Although his main interest has always been software development, he also has a long-standing interest in software testing, software measurement, and quality assurance, having presented papers on these subject at development, measurement and testing conferences in the US and Europe. He has achieved Certified Quality Analyst, Certified Software Quality Engineer, and Lead Ticket Auditor certifications, and managed the independent test function at OCLC during their three year successful effort to become registered to the ISO9000 standards. Tip can be contacted at house@oclc.org.

Carol Wade, a technical writer for over twenty years, Ms. Wade has worked primarily in the field of computer software, writing end-user documentation. For nine years, Ms. Wade worked for Los Alamos National Laboratory, where she served as a writer/editor and started a language translation service. Currently, she is the sole technical writer for Health Language, Inc., which has produced the first language engine for healthcare.

Do XP teams really need a dedicated tester? It's hard for a tester to answer this in an unbiased manner. In my experience, even senior developers don't have much testing experience, beyond unit and integration tests and perhaps load tests. They tend to write acceptance tests only for "happy paths" and don't think of the nasty evil steps that might break the system. At Tensegrent, we had one project wrapping up while another one was starting, so a decision was made to do the first two-week iteration of the new project with a developer serving as a part-time tester. By their own admission, without an experienced tester to push them, the developers got 90% of all the stories done by the end of the iteration. To the customer, this looked like nothing at all was done, and they were very unhappy. It took some work to win back the customer's trust.

How is Testing in XP Different?

How does acceptance testing in an XP environment deviate from traditional software testing? First of all, let's look at acceptance testing. Acceptance tests prove that the application works as the customer wishes. Acceptance tests give customers, managers and developers confidence that the whole product is progressing in the right direction. Acceptance tests check each increment in the XP cycle to verify that business value is present. Acceptance tests, the responsibility of the tester and the customer, are end-to-end tests from the customer perspective, not trying to test every possible path through the code (the unit tests take care of that), but demonstrating the business value of the application. Acceptance tests may also include load, stress and performance tests to demonstrate that the stability of the system meets customer requirements.

Should I strap on a helmet and arm the air bags?

Testing in an XP environment feels like a drive through twisting mountain roads at first. When I first read eXtreme Programming Explained, the very idea of testing without any formal written specifications seemed a bit TOO extreme. It's been difficult learning all the different ways I can contribute to the team's success. My roles can be confusing and conflicting – I'm part of the development team, but I need a more objective viewpoint. I'm a customer advocate, making sure the customer gets what she pays for. At the same time, I need to protect the developers from a customer who wants MORE than they paid for.

While XP is definitely a new way to drive, the road isn't as unfamiliar as some might think. For example, many people new to XP think that XP projects produce very little documentation. This hasn't been our experience. For one thing, the acceptance tests themselves become the main documentation of the customer requirements. They can be quite detailed and extensive. As an XP project progresses, many other documents may be produced: installation instructions, UML documents, Javadocs, developer setup documents, the list goes on. The difference between these and the documents in many traditional projects is, the XP project documents are up to date and accurate

Question: How do you write acceptance test cases without documents?

Answer: You don't need documents, because you have a customer there to tell you what she is looking for. Not that this is always easy. In my experience, it is fairly easy to get a customer to come up with tests for the intended functionality of the system. What is more difficult, and requires a tester's skill, is to make sure the customer thinks about areas such as security, error handling, stability, and performance under load.

Other differences between traditional and XP development are more subtle. It's really a matter of degree. XP projects move fast even when compared with the pace at the Web startup where I used to work. It's the fast lane on the Autobahn. A new iteration of the software, implementing new customer "stories", is released every one to three weeks. My goal is always to get acceptance test cases defined within the first day or two of an iteration, as these are the only written "specifications" available. For our projects, the acceptance test definitions have been a joint effort of the team.

From a tester's point of view, the developer to tester ratio in XP looks about as comfortable as driving through the desert in an un-air-conditioned Jeep. According to Kent Beck, there should be one tester for each eight-developer team. At Tensegrent, the ratio gets even higher.

Eeek! Are you SURE protective gear isn't required?

Fear not! XP builds in checks and balances that enable a small percentage of test specialists to do an adequate job of controlling quality.

- Because the developers write so many unit tests, which they must write before they begin coding – the tester doesn't need to verify every possible path through the code.
- The developers are responsible for integration testing and must run every unit test each time they check in code. Integration problems are manifested before acceptance tests are run.
- The customer gives input to the acceptance tests and provides test data.
- The entire development team, not just the tester, is responsible for automating acceptance tests. Developers also help the tester produce reports of test results so that everyone feels confident about the way the project is progressing.

A caveat – if developers aren't diligent in writing and running unit tests and integrating often, you're going to have to hire more testers. A couple of iterations into our first project at Tensegrent, I told my boss I thought we'd have to hire more testers, there was no way I could keep up! The problem was simply that the developers hadn't gotten the hang of "test before code" yet. Once they did a thorough job of unit and integration testing, my job became much more manageable.

The roles of the players on an XP team are quite blurred compared with those in a traditional software development process. Thus our Tensegrent XP ("XP") philosophy is "*specialization is for insects*". Here are some of the tasks I perform as a tester:

- Help the customer write stories
- Help break stories into tasks and estimate time needed to complete them
- Help clarify issues for design

- Team with the customer to write acceptance tests
- Pair with the developers to develop test tools, automated test scripts, and/or test data.

Question: The whole concept of pair programming sounds weird enough. How can a tester pair with a programmer?

Answer: I'm not a Java programmer and our developers don't know the WebART scripting language, but we still pair program. The partner who is not doing the actual typing contributes by thinking strategically, spotting typos and bad habits, and even serving as a sounding board for the coder. This is a fabulous way for developers and testers to understand and work together better. It also gives the tester *much* more insight into the system being coded.

I was reluctant to pair test at first. If the developers wrote the test scripts, would I be able to understand them and maintain them? The developers weren't anxious to pair with me for testing, either. They felt too busy to spare time for acceptance testing. Then we had a project where I needed very complicated test data loaded into a Poet database for testing a security model. By pairing with a developer, I finished in at least half the time it would have taken to do it alone, and did a better job. Now developers take turns on "test support" to produce test scripts and data needed for automation, sometimes also to help define test cases if I'm having trouble understanding a story.

Once you've mustered the courage to switch to the XP fast lane, it feels fun and safe.

How do I Educate Myself About XP?

Just as you wouldn't attempt to drive a Formula One car without preparing yourself with training and practice, the XP team needs good training to start off on the right road and stay on it.

Start by reading the XP books. The first written about XP is *Extreme Programming Explained*, by Kent Beck. The other two are also essential: *Extreme Programming Installed*, by Ron Jeffries, Ann Anderson, and Chet Hendrickson; and *Planning Extreme Programming*, by Kent Beck and Martin Fowler.

You can get an overview and extra insight into XP and similar lightweight disciplines from the many XP-related websites, including:

http://www.xprogramming.com
http://www.extremeprogramming.org
http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
http://www.martinfowler.com

When we at Tensegrent had assembled our first team of eight developers and a tester, we got together and went through *Extreme Programming Explained* and *Extreme Programming Installed* as a group, discussing each XP principle, recording our questions (many of them on testing) and deciding how we thought we would implement each principle. This took several hours but put us all on common ground and made us feel more secure in our understanding of the concepts.

Once your team has read and discussed the XP literature, it's time to get professional training. We hired Bob Martin of ObjectMentor, a consulting company with much XP expertise, for two days of intense training (see www.objectmentor.com for more information). After Bob answered all our questions,

we felt much more confident about areas that had previously been difficult for us to understand, such as the planning game, automated unit testing and acceptance testing.

Don't stop there. Talk to XP experts. Look at the Wiki pages and sign up for the egroups. If no XP user group has been formed in your city, start one.

Automating Acceptance Tests

What can you automate?

According to Ron Jeffries, author of XP Installed, successful acceptance tests are, among other things, customer-owned and automatic. However, customer-owned does not necessarily mean customer-written. In fact, as Kent Beck points out in Extreme Programming Explained, customers typically can't write functional tests by themselves, which is why an XP team has a dedicated tester: to translate the customers ideas into automatic tests.

Even with a dedicated tester, though, the "automatic" criterion has given us some trouble. We automate whenever it makes sense, but like most things, it is a trade-off. When you have to climb a steep dirt road every day, a four-wheel drive vehicle is a necessity, but it's overkill if you're just cruising around the block.

For example, we haven't found a cost-effective way to automate Javascript testing (so, we just avoid using Javascript). And we're also struggling with how to automate non-Web GUI testing in an acceptable timeframe.

It costs time and money to automate tests and to maintain them once you've got 'em. Recently we had a contract for three two-week iterations with four developers and myself to develop some components of a system for a customer. While the system involved a user interface, the design of the UI itself was to be done later, outside of our project. We developed a very basic interface to be able to test the system. The system involved multiple servers, interfaces, monitors and a database. Full test automation would have been a big effort. It didn't make sense to spend the customer's tight resources on scripts that had a short life span. Still, I automated the more tedious parts of the testing so I could get the tests done in time. In addition, I needed scripts for load testing. About 40% of the testing ended up automated. For a longer project, I would prefer to automate more.

Principles of XP Functional Test Automation

To get more automation, you have to make automation pay off in the short term, and this means spending less time developing and maintaining the automated tests. Here are the principles we are using to accomplish this:

- *Drive the test automation design with a "Smoke Test"*, a broad but shallow verification of all the critical functionality.
- *Design the tests like software*, so that the automated tests do not contain any duplicate code and have the fewest possible modules.
- *Separate the test data from the test code*, so that you can deepen test coverage by just adding additional test data.

- *Make the test modules self-verifying* to tell you if they passed or failed of course, but also to incorporate the unit tests for the module.
- *Verify only the function of concern for a particular test*, not every function that may have to be performed to set up the test.
- *Verify the minimum criteria for success.* "Minimum" doesn't mean "insufficient". If it weren't good enough, it wouldn't be the minimum. Demonstrate the business value end-to-end, but don't do more than the customer needs to determine success.
- *Continually refactor the automated tests*, by combining, splitting, or adding modules, or changing module interfaces or behaviour whenever it is necessary to avoid duplication, or to make it easier to add new test cases
- *Pair program the tests*, with another tester or a programmer.
- *Design the software for testability*, such as building hooks into the application to help automate acceptance tests. Push as much functionality as possible to the backend, because it is much easier to automate tests against a backend than through a user interface. I sit in on the developers' iteration planning and quick whiteboard design sessions. If I perceive business logic getting into the front end, for example in Javascript, I challenge the wisdom of such a move.

An XP Automated Test Design

Appendix A gives an example of a lightweight test design illustrating the application of the principles we have been using successfully at Tensegrent. I'm using WebART (see the Tools section below) to create and run the scripts. However, this design approach should work with any method of automation that permits modularization of scripts. The appendix gives details on downloading both the sample scripts and WebART.

Who automates the acceptance tests?

Some sports appear to be individual, when in actuality, they involve a team. Winners of the Tour de France get all the glory, but their victory represents a team effort. Similarly, the XP team may have only one tester, but the entire team contributes to automating acceptance tests. If tools are needed to help with acceptance testing in an XP project, write stories for those tools and include them in the planning game with all the other stories. You'll probably need to budget at least a couple of weeks for creating test tools for a moderately size project.

In the early days of Tensegrent, we initiated a project for the specific purpose of developing automated test tools. This had several advantages, in addition actually producing the tools:

- *Practice with XP* writing stories, playing the planning game, estimating. This gave us confidence in our XP skills that served us future projects.
- *Practice with development technologies*. Developers could experiment with different approaches and get experience with new tools. For example, the developers investigated in advance the advantages of using a dom versus a sax parser on the XML files containing customer test data. Doing this in advance gave us more time to experiment and research

technologies than we might have had later with a client project.
- *Mutual understanding*. The team tasked with producing an acceptance test driver consisted of only four members and me, so I was called on to pair program. This exercise gave me insight into how tough it is to write unit tests, write code and refactor the code. The developers gave a lot of thought to acceptance testing and we had long discussions about what the best practices would be. This is a great foundation for any XP team.

Tools

To keep the XP car humming, XP testers need a good tool-box: one containing tools designed specifically for speed, flexibility and low overhead.

I've asked several XP gurus, including Kent Beck, Ward Cunningham and Bob Martin, the following question: "What commercial tools do you use to automate acceptance testing?" Their answers were uniform: "Grow your own". Our team extensively researched this area. Our experience has been that we are able to use a third-party tool for Web application test automation, but we need homegrown tools for other purposes.

For *unit testing*, we use a framework called jUnit, which is available free from http://www.junit.org. It does an outstanding job with unit tests. Even though I am not a Java programmer, I can run the tests with jUnit's TestRunner and can even understand the test code well enough to add tests of my own. It's possible to do some functional tests with jUnit. Some XP teams use this tool for automating acceptance tests, but it can't test the user interface. We didn't find it to be a good choice for end-to-end acceptance testing.

Tools for Creating Acceptance Tests

Some XP pros such as Ward Cunningham advocate the use of spreadsheets for driving acceptance tests. We want to make it easy for the customer to write the tests, and most are comfortable with entering data in a spreadsheet. Spreadsheets can be exported to text format, so that you and/or your development team can write scripts or programs to read the spreadsheet data and feed it into the objects in the application. In the case of financial applications, the calculations and formulas your customer puts into the spreadsheet communicate to the developers how the code they produce should work.

At Tensegrent, we provide a couple of ways for documenting acceptance test cases. Usually we use a simple spreadsheet format, separating the test case data itself from the description of the test case steps, actions and expected results. We've also experimented with entering test cases in XML format which is used by an in-house test driver. We're continuing to experiment with the XML idea, but the spreadsheet format has worked well. See Appendix B for a sample acceptance test spreadsheet template.

Appendix C shows a *partial* excerpt of a sample XML file used for acceptance test cases. The test case consists of a description of the test, data and expected output, steps with actions to be performed and expected results.

Automated Testing for Web Applications

Test automation is relatively straightforward for Web applications. The challenge is creating the automated scripts quickly enough to keep pace with the rapid iterations in an XP project. This is always toughest in the early iterations. There are times that I feel like the slow old car blocking the fast lane. For that extra burst of speed, I use WebART (http://www.oclc.org/webart), an inexpensive HTTP-based tool with a powerful scripting language. WebART enables me to create modularized test scripts, creating many reusable parts in a short enough timeframe to keep up with the pace of development. Javascript testing presents a bigger obstacle. We test it manually and carefully control our Javascript libraries to minimize changes and thus the required retesting. Meanwhile, we continue to research ways of automating Javascript testing.

Our developers wrote a tool to convert test data provided by the customers in spreadsheet or XML format into a format that can be read by WebART test scripts so that we can automate Web application testing. Even small efforts like this can help you gain that competitive edge in the speedy XP environment.

Automated Testing for GUI Applications

Test automation for non-HTTP GUI applications has been more of an uphill climb. You can travel faster in a helicopter than a mountain bike, but it takes a long time to learn to fly a helicopter; they cost a lot more than a bicycle and you may not find a place to land. Similarly, the commercial GUI automated test tools we've seen require a lot of resources to learn and implement. They're budget breakers for a small shop such as ours. We searched far and wide but could not come up with a WebART equivalent in the GUI test world. JDK 1.3 comes with a robot that lets you automate testing of GUI events with Java, but it's based on the actual position of components on the screen. Scripts based on screen content and location are inflexible and expensive to maintain. We need tests that give the developers confidence to change the application, knowing that the tests will find any problems they introduce. Tests that need updating after each application change could cause us to lose the race.

We felt that the most important criteria for acceptance tests is that they be repeatable, because they have to be run for each integration. We decided to start by developing our own tool, "TestFactor-e", that will help customers and testers run manual tests consistently. It will also record the results. We plan to enhance this tool to feed the test data and actions directly into application backends in order to automate the tests. As we have only been developing web applications, this effort is on the back burner.

No matter what the system being tested, it takes time to get up to speed with automation. I plan to do manual testing in the first iteration. At the start of the second iteration, I can start automating, using the method described in Appendix A. There are times I run into a roadblock which sets me back a day or two. The solution to that is to find someone to pair with me. As the tester in an XP project, you may feel lonely at times, but remember, you aren't ever alone!

Reports

Getting feedback is one of the four XP values. Beck says that concrete feedback about the current state of the system is priceless. If you're on a long road trip, you check for road signs and landmarks that tell you how far along your route you've come. If you realize you're running behind, you skip the next stop for coffee or push the speed a bit. If you're ahead of schedule, you might detour to a more scenic road. The XP team needs a constant flow of information to steer the project, making corrections to stay in the lane. The team's continual small adjustments keep the project on course, on time and on budget. Unit tests give programmers minute-by-minute feedback. Acceptance test results provide feedback about the "Big Picture" for the customer and the development team.

Reports don't need to be fancy, just easy to read at a glance. A graph showing the number of acceptance tests written, the number currently running and the number currently succeeding should be prominently posted on the wall. You can find examples of these in the XP books. Our development team wrote tools to read result logs from both automated tests and manual tests run with "TestFactor-e". These tools produce easy-to-read detail and summary reports in HTML and chart format.

With all this feedback, you'll confidently deliver high-quality software in time to beat your competition. You'll meet the challenges of 21st century software development!

Appendix A: Lightweight Test Design

XP Automated Test Design

The sample scripts used to illustrate the test design are written with a test tool called WebART (http://www.oclc.org/webart/). Any test tool that permits modularization and paramaterization of the scripts should support this design. To download a soft copy of the sample scripts, go to http://www.oclc.org/webart/samples/ and click on the "qwmain Sample Scripts" link.

The Sample Application

Our sample application is a telephone directory lookup website, http://www.qwestdex.com. This is certainly not intended as an endorsement of Qwest and we have no connection with them, it was just a handy public application with characteristics that allow us to illustrate the tests.

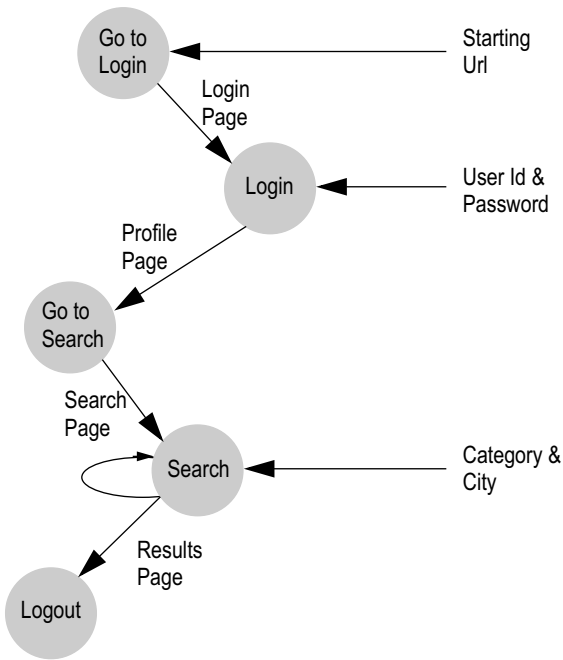| Action	Minimum Passing Criteria
Go to login page	Page contains the login form
Login	Valid login name and password brings up profile page
Search for valid category in specified city	Valid search retrieves table of businesses
Logout	Page contains link to login page and home page

Table 1

Figure 1

The Smoke Test

We will consider the critical functionality to be logging into the site and finding the businesses within a certain city and category. Pretend that this is the most important story in the first iteration. Table 1 shows the basic scenario we want to test.

The Test Design

We know that there will be more functionality to test in subsequent iterations, but we will use the simplest design we can think of to accomplish these tests without duplication. Then we will refactor as necessary to accommodate the additional tests.

The modules will be Go to Login, Login, Go to Search, Search, and Logout. In figure 1 is a diagram showing how the modules are parameterized.

Separating the test data from the code

The items on the right side of the diagram represent test data: the URL of the login page, the user id and password to use to login, and the category and city to search. The test data is segregated into a test case file, which is read in by the test when it executes. In figure 2 is sample content of that file to run a single test case.

Verification

The main modules use a set of primitive validation modules to check for the specific conditions required in a system response and determine a pass or fail condition. The validation modules in turn call *utility* modules to record the results.

This example uses the following three validation modules:
- *vtext* validates that a response contains specified text. for the text string.
- *vlink* validates that a page contains a specific link.

```
smoketest
  [
  :iter1:
  Url <url=http://qwestdex.com>
  UseridPassword <uid=bob&psw=bob>
  CatCity <cat=banks&city=dallas>
  ]
```

Figure 2

- *vform* validates that a page contains a specified HTML form.

Utility Modules

There are also two utility modules which are used by the main modules:
- *trace* – Displays execution tracing information in the WebART execution window, for debugging the tests.
- *log* – Records validation outcomes in a log file.

The "zslog" module in the sample scripts writes test results out in XML format. An in-house tool from Tensegrent called TestFactor-e builds an HTML page from this log file showing the results with colour-coding for pass, not run and fail. See Appendix B for an example.

Creating the Scripts

Creating the first set of scripts is the hard work. Once you have a working set of modules, you can reuse entire modules in some cases or turn them into templates in other cases. Here are the steps I use (preferably as part of a pair) to create test scripts:

1. Capture a session for the scenario I want to test. See "capqwest" in the sample scripts as an example.

2. Copy "qwmain", "zsqwlogin" and the other supporting modules that I already have to new names. Strip out the code that was specific to that application.

3. Paste in the code specific to the scenario I want to test, copying from the captured script into the newly created "templates". Use XP principles here: work in small increments, make sure your scripts work before you go on. For example, first see if you can get the login to work. Then add the search. Then add the logic for switching depending on the pass/fail outcome. Remember to do the simplest thing that works and add complexity only as you need it.

Appendix B: Partial Excerpt of XML Template for Acceptance Test Cases

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<!DOCTYPE at-test SYSTEM "at-test.dtd" [
  <!ELEMENT input ANY >
  <!ELEMENT loan-amount ANY >
  <!ELEMENT interest-rate ANY >
  <!ELEMENT term-of-loan ANY >
  <!ELEMENT output ANY >
```

```
  <!ELEMENT monthly-payment ANY >
]>

<at-test name="calc-monthly-payment" version="1.0" severity="CRITICAL">

 <at-project>mortgage-calc</at-project>

 <at-description>
   Enter loan amount, interest rate, term of loan (in months)
   to calculate monthly payment.
 </at-description>

 <at-data-sets>
   <at-struct id="values">
     <input>
     <loan-amount>1000000000.00</loan-amount>
     <interest-rate>0.5</interest-rate>
     <term-of-loan>1200</term-of-loan>
     </input>
     <output>
     <monthly-payment>A big, fat wad of dough!</monthly-payment>
     </output>
     </at-struct>
 </at-data-sets>
```

```
<at-plan>

  <at-step name="populate-loan-amount">
    <at-action>
      <at-text>Enter "{0}" in the "Loan Amount field".</at-text>
      <at-value dset="values" select="/input[2]/loan-amount"/>
    </at-action>
    <at-expect>
      <at-text>Cursor moved to "Interest Rate" field for input.</at-text>
    </at-expect>
  </at-step>

</at-plan>

</at-test>
```

Appendix C: Sample Acceptance Test Spreadsheet

	A	B	C	D	E
1		Ref #:	Template for acceptance test: Timecard/QA/AcceptanceTest.sdc		
2		Iteration:			
3		Functionality proven by this test case * is / is not * critical	**What does this test?** *Example: Tests to make sure that when a time entry record is input, it is saved to the database and the report generated correctly.*		
4		What to do:	*Examples given in italics*		
5	Step	Command/URL	Action	Input Data	Expected Output
6	1	*Access www/timecard/addEntry in browser*	*Select a project, release, iteration, task, and enter duration, date and comments*	*Row 1, columns Project, Release, Iteration, Task, Duration, Date, Comments*	*Selections displaying on screen*
7	2	*Still on www/timecard/addEntry*	*Click the save button*		*Message that data was saved to database*
8	3	*Access www/timecard/generateReports in browser*	*Select a project, start date and end date*	*Row 1, columns Project, Start Date, End Date*	*Report generated – data matches row 1 columns Project, Release, Iteration, Duration, Cost and Total Cost*
9	4	*Repeat steps 1 – 3 with each row in the test case spreadsheet*			

Qualitative Studies of XP in a Medium Sized Business

Robert Gittins, Sian Hope and Ifor Williams

Qualitative Research Methods are used to discover the effects of applying eXtreme Programming (XP) in a software development business environment. Problems dominating staff development, productivity and efficiency are parts of a complex human dimension uncovered in this approach. The interpretation and development of XP's "Rules and Practices" are reported, as well as the interlaced communication and human issues affecting the implementation of XP in a medium sized business. The paper considers the difficulties of applying XP in a changing software requirements environment, and reports on early deployment successes, failures and discoveries, and describes how management and staff adapted during this period of change. The paper examines the benefits of a flexible management approach to XP methodology, and records the experiences of both management and staff, as initial practices matured and new practices emerged.

Keywords: Extreme Programming, Qualitative methods, Software Methodology.

1 Related Work

Previous qualitative research [Seaman 99], [Sharp et al. 99], [Cockburn/Williams 02], has concentrated on non-judgmental reporting, with the intent of provoking discussion within the culture being studied by providing observations and evidence, collaborators deciding for themselves whether any changes were required. This fieldwork study follows the format of [Gittins/Bass 02], whereby the researcher is immersed for a period in the software developer team; thereby the active researcher becomes instrumental in the development and improvement of XP. [Seaman 99] describes an empirical study that addresses the issue of communication among members of a software development organization. [Sharp et al. 99] use combined *ethnography* and *discourse analysis*, to discover implicit assumptions, values and beliefs in a software management system. [Cockburn/Williams 02] investigate *"The cost benefits of pair programming"*. [Sharp et al. 00] describe a "cross-pollination"' approach, to a deeper understanding of implicit values and beliefs.

XP developed recently from [Beck 00] and [Beck/Fowler 00], and more recently in [Jeffries et al. 00]. [Williams/Kessler 00] study lone and paired programmers, and [Williams et al. 00] the cost effectiveness of pairing.

2 The Study

Secure Trading, the focus of this paper, is a medium sized software company committed to implementing XP, and comprises a team of nine developers. Secure Trading decided to implement XP in a progressive manner, conscious of minimising disruption to the business process. Reference material from other companies, not specifically named in this paper, will only be used in general terms to highlight some typical problems facing established, and highly traditional companies,

Robert Gittins is final year PhD student at the Ada Lovelace Laboratory, University of Wales, BangorGwynedd, Wales, UK. His research explores the interfaces between the disciplinary experts who collaborate to develop approaches to developing commercial software development solutions, especially for distributed systems. Communication between these disciplinary domains, and the cooperative solution of conflicting design problems, are the key areas of his investigation. His research goal is to contribute strategies, methods or algorithms for novel software tools to support the design process. <r.g.gittins@informatics.bangor.ac.uk>

Sian Hope is a senior lecturer at the School of Informatics, University of Wales Bangor. Research interests are focused on contributing to enhancement of the discipline of Software Engineering by researching into practically applicable development methods which are firmly grounded on scientifically sound concepts. Fundamental approaches are sought, which provide a bridge between theory and practice and which are aimed at producing engineering methods, tools and metrics for all phases of software development. Sian can be contacted at sian@informatics.bangor.ac.uk

Ifor Williams. After obtaining a degree in Computer Engineering from the University of Manchester Ifor went on to gain a PhD for research into computer architectures suitable for the efficient execution of object-oriented applications. This work resulted in the design of a machine (MUSHROOM) incorporating support for dynamic binding, object-based virtual memory, efficient garbage collection and targeted as a high-performance Smalltalk platform. Later he spent some time developing software for the medical diagnostics industry using prescribed conventional development processes before joining the rapidly changing.com world where XP proved to be invaluable. He can be contacted at ifor.williams@securetrading.com.

sensitive to their developer environment, and to the cost of disruption that change would incur on staff and production.

Secure Trading, had recently moved to larger offices. When research started, their involvement with XP consisted of some intermittent attempts at "pairing" developers. Their move presented opportunities for improving "pairing" proficiency, and the selective adoption of XP practices.

3 Qualitative Research Work

This research adopts some of the techniques historically developed in the Social Sciences [Gittins 02], *ethnography*, *qualitative interviews* and *discourse analyses*, an understanding of "grounded theory" was particularly important. Grounded theory can provide help in situations where little is known about a topic or problem area, or to generate new ideas in settings that have become static or stale. Developed by Barney Glaser and Anselm Strauss [Glaser/Strauss 67] in the 60s, grounded theory deals with the generation of *theory* from *data*. Researchers start with an area of interest, collect data, and allow relevant ideas to develop. Rigid pre-conceived ideas are seen to prevent the development of research. To capture relevant data, qualitative research techniques are employed [Gittins/Bass 02] that include the immersion of the researcher within the developer environment, qualitative data analyses, guided interviews, and questionnaires.

3.1 Qualitative Data

Qualitative evaluation allows the researcher to study selective issues in detail, without the pre-determined constraints of "categorised" analyses. The researcher is instrumental in the gathering of data from open-ended questions. Direct quotations are the basic source of raw materials, revealing the respondent's depth of concern. This contrasts with the statistical features of quantitative methods, recognised by their encumbrance of predetermined procedures.

3.2 Qualitative Interviews

[Patton] suggests three basic approaches to collecting qualitative data through interviews that are open-ended. The three approaches are distinguished by the extent to which the questions are standardised and predetermined, each approach having strengths and weaknesses, dependant upon the purpose of the interview:

1) *"Informal conversational"* interviews, are a spontaneous flow of questions where the subject may not realise that the questions are being monitored. 2) The *"General interview guide"* approach, adopted extensively for this study, predetermines a set of issues to be explored. 3) The *"Standardised open-ended interview"* pursues the subject through a set of fixed questions that may be used on a number of occasions, with different subjects.

In a series of interviews, data was collected using "Informal conversation" and verbatim transcripts taken from "General guided interviews".

3.3 Questionnaires

In an extensive questionnaire consideration was given to the "Rules and Practices" of XP. Questions targeted the software development process, XP practices, and both managerial and behavioural effectiveness. Behavioural questions were based upon Herzberg's "Hygiene and Motivation Factors" [Herzberg 74]. Ample provision was provided for open comments on each of the topics, and a developer floor plan provided for a respondent to suggest improvements to the work area. Repeating the questionnaire at three monthly intervals will help research and management by matching the maturing XP practices, as they progress, against developer responses.

4 Rules and Practices

4.1 Pair Programming

(See [Beck 00], [Beck/Fowler 00]) XP advances what has been reported for some time [Cockburn/Williams 02], [Williams/Kessler 00], [Williams et al. 00]; Two programmers working together generate an increased volume of superior code, compared with the same two programmers working separately. Secure Trading management, discussed the implementation of "Pairing" with the development team, who unanimously agreed to "buy-in" to the practice. The first questionnaire showed some of the team were unhappy with pairing. 28% of developers preferred to work independently, 57% didn't think they could work with everyone, and 57% stated that pair programmers should spend on average 50% of their time alone. XP practices recommend no more than 25% of a conditional 40-hour week be paired. Two developers summed up the team's early attitude to pair programming: *"I feel that pair programming can be very taxing at times, although I can see the benefits of doing it some of the time."*

"Not everyone makes an ideal pair. It only really works if the pair is reasonably evenly matched. If one person is quiet, and doesn't contribute, their presence is wasted. Also, if a person is really disorganised and doesn't work in a cooperative way, the frustration can (disturb) the other participant!"

Developers estimated that they spent approximately 30% of their time pairing, with partner changes occurring only upon task completion, changes being agreed and established *ad hoc*. Frequent partner swapping, and partner mixing, commands great merit in XP. Pairing practices matured with the introduction of a team "Coach" and later a "Tracker" [Beck 00]. Maintenance tasks were another problem which routinely disrupted pairing. Here control was reviewed and tasks better ordered to minimise this problem. In time, the impact of pairing activity upon developers will translate into evidence, returned in the periodic questionnaire reviews, and in the timeliness and quality of code produced.

4.2 Planning Games

(See [Jeffries et al. 00]) Planning games were introduced soon after pairing practices were established. The "customer" duly chooses between having more stories, requiring more time; against a shorter release, with less scope. Customers are not permitted to estimate story or task duration in XP and

developers are not permitted to choose story and task priority. Where a story is too complex or uncertain to estimate, a "Spike" is created. Spike solutions provide answers to complex and risky stories. Secure Trading succeeded well in developing Planning games, utilising "Spike solutions" by logging a "spike" as a fully referenced story to quickly attack the problem, reducing a complex, inestimable story to a simple, and easily understood, group of stories. Results were very effective; "spike solutions" proved easy to develop and derived estimates for completion proved consistently accurate. It was common practice to have the essential elements of both *iteration* and *release* Planning games combined into one meeting. This practice worked for them in the context of the jobs they were planning.

4.3 Client On-site

(See [Beck 00]) Secure Trading rarely had this luxury. When required the "Client" role was undertaken by a client's representative, co-opted from the Customer services department by staff who had worked closely with the client and were able to accept that responsibility. Developer Manager: *"The inclusion of a representative from Customer services has proven to be hugely beneficial, providing immediate feedback of the system's successes and failures on a day-to-day basis."*

4.4 Communication

(See [Beck 00]) A great deal of attention is necessary in providing an XP environment in keeping with the practices to support XP. Key factors in communication are: the use of white boards, positioning and sharing of desk facilities to facilitate pair programmers, "stand-up" meetings, developers "buying-in" to the concepts of the *"rules and practices"* of XP, and "collective code ownership". Interviews and questionnaires revealed many areas of concern among developers. For example, 86% of developers disagreed that meetings were well organized; *"Agreements at meetings are not set in concrete"* and, *"Confidence is lost with meeting procedures, when agreed action or tasks are later allowed to be interpreted freely by different parties."* Management were quick to address these concerns by concentrating on the development of XP story card practices. Developers were encouraged to agree, and finalise with the client, the task description and duration estimates at timely Planning Game meetings. Story cards were fully referenced and signed by the accepting developer, thereby becoming the responsibility of the initiating developer until completion. Only the responsible creator of a Card was authorized to amend it.

The use and placement of White boards is said to be an essential supporting means of good communication in XP practices [Beck 00]. Mobile whiteboards were introduced by Secure Trading soon after pair programming practices gained momentum and used to record the story details agreed at Planning Game meetings. At one point, story cards were physically stuck to the boards in prioritised order with adjacent notes written on the board. This proved unpopular and developed into cards being retained but not stuck on the white board. Stories were written on the boards. Referenced stories

contained ownership, estimation, as well as iteration and priority, which were displayed in columned format. On completion, the owner added the actual task duration. The information served to improve personal proficiency in estimation and in providing feedback towards establishing project "velocity" data, for future Planning Game meetings.

Stand-up meetings promote communication throughout the team. Secure Trading introduced this practice from day one. At ten o'clock every morning, a meeting allowed everyone to briefly state (standing promotes brevity) their work for the day, and discuss problems arising from the previous days activity. Anyone was free to comment, offer advice or volunteer co-operation. The benefits of adopting stand-up meetings were far-reaching and seen by developers and management as an effective way to broadcast activities, share knowledge and encourage collaboration amongst and between team members and management. Secure Trading meetings tended to degrade when reports migrated to topics of yesterday's activity, rather than those planned for the day. This activity persists and may remain or need to be resolved and modified as their particular brand of XP develops.

4.5 Simple Design

Beck [Beck 00] summarises simple design in *"Say everything once and only once."* However a comment by one developer interviewed revealed a common concern, *"Sometimes, it is a bit too simplistic, and issues seem to be avoided"*. XP states that it is important to produce a simple system quickly, and that "Small Releases" are necessary to gain feedback from the client. Secure Trading didn't see themselves in a position to implement this practice so early in their XP programme. XP allows companies to cherry-pick those practices they regard suitable for implementation, in the order they see fit.

4.6 Tests

Unit tests are written in XP before main code and give an early and clear understanding of what the program must do. This provides a more realistic scenario, as opposed to "after-the-code testing," that could, for many reasons, neatly match completed code. Time is saved both at the start of coding, and again at the end of development. Latent resistance to early unit testing became manifest, when the perceived closeness of a deadline loomed. This activity is perhaps the hardest to implement and requires commitment from developers. An early questionnaire revealed that 71% of Secure Trading developers regarded unit-testing practices in general to be "very poor". Developer Manager on early introduction of unit testing: *"If you already have a large complex system, it is difficult to determine to what extent testing infrastructure is to be retrospectively applied. This is the most difficult aspect in our experience. Starting from scratch it is much easier to make stories and code testable."*

4.7 Refactoring

(See [Fowler 99]). *"The process of improving the code's structure while preserving its function."* The use and reuse of old code is deemed costly, often because developers are afraid

they will break the software. XP indicates that refactoring throughout the project life cycle saves time and improves quality. Refactoring reinforces simplicity by its action in keeping code clean and reducing complexity. Secure Trading had not developed refactoring activities in line with XP at that time. Many developers expressed concern with refactoring, more commonly reported by traditional companies: *"… with more people, we could spend more time refactoring and improving the quality of our existing code base."* The questionnaire revealed that 45% of developers considered refactoring sporadic or very poor.

4.8 Collective Code Ownership

(See [Beck 00], [Beck/Fowler 00]). This concept states *"Every programmer improves any code anywhere in the system at any time if they see the opportunity."* Collective code ownership has many merits: It prevents complex code entering the system, developed from the practice that anyone can look at code and simplify it. It may sound contentious, but XP Test procedures should prevent poor code entering the system. Collective Code Ownership also spreads knowledge of the system around the team. Secure Trading experienced growing pains in developing this principle, revealed by the comments of two developers: *"I have conflicting interests in collective code ownership. I think it is very good when it works, but there are times when some code I have written seems to just get worse when others have been working on it."*

"I like the idea of collective code ownership, but in practice I feel that I own, am responsible for, some bits of code." From the traditional perspective of individual ownership, it will be important to record how attitudes change, as XP practices mature.

4.9 Metaphor

A metaphor in XP is a simple shared story to encompass and explain what the application is "like", communicating a mental image, so that everyone involved can grasp the essence of the project in a term universally understood. This may seem to be a relatively easy, or lightweight, activity to adopt. However, the value of this practice was not immediately evident to developers, early difficulties developing and applying suitable metaphors were experienced and this practice was reluctantly abandoned for future consideration.

5 Companies Starting from Scratch

Long established and traditional companies, considering adopting XP, have, unlike Secure Trading, many more difficulties to overcome. They mostly comprise traditional teams of developers, who are comfortably established, working in small offices, in prohibitively cloistered environments. Management is often aware that legacy software in circulation is in the "ownership" of one or two heroic developers, at the cutting edge of their business. Some teams were reported as badly under-performing and in some circumstances management had resorted to consultants to resolve their problems with no significant success reported. Often with great reluctance, management

allowed the research team to visit developer offices. Tension was evidently high. In these companies, "Risks" [Beck 00] are high, quality is compromised, communication difficult, and control largely ineffective. There are other considerations when starting from scratch; The Secure Trading developer manager reflecting upon attempts at implementing XP in his early projects stated: *"One of the key "discoveries" has been the relative ease to which XP has been employed on an all-new project, and the difficulty in applying XP retrospectively on an established system."*

6 Conclusions

A combination of qualitative and quantitative methods has helped identify uncertainties in applying XP practices in a medium sized software development company. How particularly one company interpreted and developed their *brand* of XP, moulded from their successes and failures. Successes in such areas as the use and development of "spike solutions", and Customer role-play within "Planning Game" activity, and from failures, as in developer reluctance to "buying-in" to "collective code ownership", and the difficulties of implementing the practice of "simple design", and in the use of "metaphors". Partial success was seen in "Pair programming", that having posed early problems, showed improvement in maturity. Future work will monitor the complex factors in the development of XP within small and growing companies at various levels of maturity. By acknowledging the characteristic unsharp boundaries of qualitative data sets, future work will investigate the use of fuzzy logic for data analyses.

References
[Beck 00]
    K. Beck: "Extreme Programming Explained: Embrace change". Addison Wesley. 2000
[Beck/Fowler 00]
    K. Beck and M. Fowler: "Planning Extreme Programming". Addison Wesley 2000.
[Cockburn/Williams 02]
    A. Cockburn and L. Williams: "The cost benefits of pairprogramming".
    http://members.aol.com/humansandt/papers/
    pairprogrammingcostbene/pairprogrammingcostbene.htm.
[Fowler 99]
    M. Fowler: "Refactoring: Improving the design of existing code", Addison Wesley. July 1999.
[Gittins 02]
    R. G. Gittins: "Qualitative Research: An investigation into methods and concepts in qualitative research". Technical Paper: via http://www.sesi.informatics.bangor.ac.uk/english/home/research/technical-reports/sesi-020.htm
[Gittins/Bass 02]
    R. G. Gittins and M. J. Bass: "Qualitative Research Fieldwork: An empirical study of software development in a small company, using guided interview techniques", TechnicalPaper: via http://www.sesi.informatics.bangor.ac.uk/english/home/research/technical-reports/sesi-021.htm

[Glaser/Strauss 67]
B. G. Glaser and A. L. Strauss: "The discovery of grounded theory: strategies of qualitative research" Chicago: Aldine Publications. 1967

[Herzberg 74]
F. Herzberg: "Work and the Nature of Man", Granada Publications Ltd.1974

[Jeffries et al. 00]
R. Jeffries, A. Anderson and C. Hendrickson: "Extreme Programming Installed". Addison Wesley 2000.

[Patton]
M. Q. Patton: "Qualitative Evaluation and Research Methods" (2nd Edit.). SAGE Publications

[Seaman 99]
C. B. Seaman: "Qualitative methods in empirical studies of software engineering", IEEE Trnsctns on Software Engineering, Vol.25 (4):557–572 Jul/Aug 99.

[Sharp et al. 99]
H. Sharp, M. Woodman, F. Hovenden and H. Robinson: "The role of 'culture' in successful software process improvement." EUROMICRO:1999 Vol.2, p17.

[Sharp et al. 00]
IEEE Computer Society. H. Sharp, H. Robinson and M. Woodman: "Software Engineering: Community and Culture". IEEE Software, Vol. 17, No.1, Jan /Feb2000

[Williams/Kessler 00]
L. A. Williams and R. R. Kessler: "All I Really Wanted to Know About Pair Programming I Learned in Kindergarten". Communications of the ACM. May 2000 Vol.43, No5.

[Williams et al. 00]
L. A. Williams, R. R. Kessler, W. Cunningham and R. Jeffries: "Strengthening the Case for PairProgramming". IEEE Software, Vol. 17, No. 4: July/August,2000,pp19–25.

XP and Software Engineering: an opinion

Luis Fernández Sanz

In this article, the author makes some reflections on certain specific aspects of eXtreme Programming as described in Kent Beck's book "eXtreme Programming explained. Embrace change". The analysis presented here is in relation to principles and techniques of software engineering.

Keywords: eXtreme Programming, XP, Software Engineering

1 First contact

The first time I came across the term EXtreme Programming (also known by the initials XP), my mind was immediately overrun with images of those well known extreme sports: people who love danger and who spend their time skiing down impossible mountainsides, making bungee jumps, etc.[1] Of course the expression was deliberately coined by Kent Beck to benefit from the fashion for this kind of sport in order to make a bigger splash on the IT scene. Perhaps the idea was also to suggest a world of "winners", people who flirted with danger, who were always "cool" (another in word) whatever they did and who turned their back on convention. Regrettably, however, I discovered it was something altogether simpler and less glamorous than the risk and adventure of extreme sports[2] but, fortunately, I did recognise the tremendous appeal of a new approach to software development.

Curiosity naturally led me to seek out references on XP where I could learn some more about it. Coincidentally, and at the same time, I began to hear opinions from experienced people; people who could never be accused of not being broad minded. While some of them were drawn to this idea (although somewhat sceptical about its potential for becoming a common practice), others pointed out in no uncertain terms the "madness" of expecting to achieve quality in software developed using XP practices. I have been in this field long enough to experience several passing fads which claimed to be the cure of all the ills besetting software development (of which there are plenty), and high hopes were placed in all of them (mainly by their mentors and supporters, sometimes with obvious commercial interests at heart) in terms of their scope and life span. There were many such trends and most passed away, although it's only fair to admit that some did make a contribution to current good practices. Which is why something told me that I could be looking at just another passing fad wrapped up

Luis Fernández Sanz received a degree in informatics engineering from Technical University of Madrid (Spain) in 1989 and a Ph. D. degree in informatics from University of the Basque Country in 1997 (as well as an extraordinary mention for his doctoral thesis). He is currently head of the department of programming and software engineering at Universidad Europea-CEES (Madrid). From 1992, he is the coordinator of the software engineering section of Novática. He is author or coauthor of several books about software engineering and software measurement, as well as different papers in international journals and conferences. He is member of the Software Quality Group of ATI and he has acted as chair of the VI Spanish Conference on Software Quality and Innovation organised by ATI. He is a member of ATI and the Computer Society of the IEEE. <lufern@dpris.esi.uem.es>

in impressive sounding terms. Even when a colleague, the type who like to loudly proclaim any nonsense they have just learnt (preferably something trendy) and who need constant attention to compensate for their inferiority complex, piped up that he knew what XP was and waxed lyrical about how wonderful it all was[3], subjectivity got the better of me.

But there is one thing I just cannot help doing: I can't help trying to find out for myself about the things which arouse my curiosity, and I will not take anyone else's opinion as gospel. In my quest for information about XP it was not hard for me to find various web sites (see the brief list at the end of this article) which contained a copious selection of documentation, links and resources on the subject of eXtreme Programming. Of course, I had Beck's classic book [Beck 00] in which he explains the essence of XP. I even discovered the existence of monographic international congresses on XP with several editions already held. However something was worrying me. I could see the same signs that I had seen before in some of the previous fads: a lot of words (albeit reasonable and attractive ones), a certain implicit assumption that XP is a dogma of faith (nobody doubted that its principles were properly justified) and, especially, a shortage of really reliable data and real life

---

1. This "confession" may not seem so surprising when you read McCormick's article [McCormick 01] which you can find elsewhere in this edition.
2. The truth is that, in terms of physical appearance, there is seldom any possible comparison between IT people and the extreme sports community.

---

3. It goes without saying that he planned to reap all the benefits of XP straight away as his idea was to implement it immediately (as I write, I have had no reports that he has done anything, just like on so many other occasions).

experiences. In fact, a simple search involving direct consultations with famous "extremes" in the international arena always resulted in the same response: at best some academic experiments somewhat divorced for any professional reality, some experience, but few projects (though there were some) with specific data and much less any formal experiments. However, my intention is not to put XP practitioners down in any way with this: it is sadly only too common a state of affairs in the software world in any field[4]. However Chrysler's famous C3 project in which Beck applied XP practices for the first time is much talked about.

My main problem when it comes to expressing an opinion about XP is that I don't have any first hand references: I have had no opportunity to apply XP in a project nor have I managed to find colleagues in Spain who are practising it in a professional environment. It's true there are companies like IBM who have given support to specific aspects of XP (for example, with JUnit) and who have documentation on it. Other major companies like Sun give it some kind of exposure in their technical documents, almost certainly because their marketing people don't want to miss out on the advertising draw it has (at least in terms of the name and the frequency with which it crops up in publications).

Anyway, after a careful reading of Beck's book [Beck 00] and with the inspiration of various documents taken from web sites on XP and the occasional interesting article like McCormick's [McCormick 01], I would like to share some of my reflections with the readers.

2 Reflections on XP from a Software Engineering viewpoint

One of the bases of XP is the technical promise which Beck makes at the beginning of his book [Beck 00]. The problem lies in the changes which have to be made in a software development. XP aims to minimise changes by concentrating only on those changes which simplify code (which should be as simple as possible[5]) or which will facilitate the rest of the code. Consequently, it is important to design without having to complicate the design in an attempt to cater for possible future expansions or increased functionality or performance. From my point of view, this concern about change is laudable, though I am not so sure that it is always a good idea to ignore future extensions for the sake of simplifying the design as much as possible. In this respect, the reasoning behind the approach to projects is based on three variables: the scope of the project, the quality and the cost. Beck's rationale lays emphasis on the maximum reduction of the scope of the project (that is, concentrating only on functionality and the features which are strictly necessary) to gain advantages in the other two variables.

Once the ground rules have been established for action, a great deal of what the developers do should be aimed at simplifying the design (without making any extra effort with an eye to the future), performing automated tests and accumulating a lot of practice in the modification of designs so as to lose the fear of making changes. In fact, a phrase has been coined which uses an analogy to sum up XP's philosophy: "Driving is not about getting the car going in the right direction. Driving is about constantly paying attention". In this case, the driver is the customer with whom we should have a constant interaction. To achieve this the customers' directors have to be convinced that, if they can't free a person to attend to the developers constantly, maybe their bet on the system under construction is not worth making.

From my point of view, it is especially satisfying that software tests (always the most hated and neglected part of a development) are being given more importance and, it is also worth underlining that productivity in this area (and in others such as coding) depend on the introduction of automated environments. In my experience, generally speaking there tends to be practically no use of support tools in automated tests by organisations developing conventional software[6]. However, I am not sure if XP's strategy concerning changes and design will always be appropriate. What I do like is the fact that emphasis is put on simplicity achieved through good design, since I believe that all too often developers opt for code which "more or less works" and take little time to consider what might be the simplest and most understandable code to implement a functionality.

XP also promotes certain interesting values of human team and project management (as well as some technical aspects): communication, simplicity, feedback with the customer and courage when tackling the problems and challenges thrown up by design. But, above all, what it proposes is a life cycle or process which is "lightweight" (as McCormick says [McCormick 01]) or agile. This fast process, which is a successor to the RAD concept and evolutionary prototyping, involves very fast iterations (releasing versions, builds or whatever we want to call them) run at very frequent intervals: one a day or even every few hours. This requires the tests to be very agile and highly organised, which is only possible if they can benefit from efficient and well organised automation.

Each XP iteration involves:
- A coding phase: code is the essential building block and the primordial aim of XP as a concise and precise communication vehicle, regardless of whether we work with visual environments or text editors or code generators.
- An indispensable automated test phase. Personally I find it very satisfying to see how in XP tests are not a torture but rather something which is more fun than programming.

---

4. At the end of the day, I am influenced by my work on software measurement which led up to my doctoral thesis (as well as an extraordinary mention for my doctoral thesis), and to the publishing of the first book in Spain on this subject, with the collaboration of the outstanding researcher and professor Javier Dolado (my thesis director), together with other eminent researchers from Spain and Great Britain, [Dolado/Fernández 00].

5. Although as Einstein once said, "it should be as simple as possible but no simpler".

6. In several surveys when giving training courses on software tests, a significant percentage of the attendees admitted a very poor level of automation and support environments, and also of test organisation and procedure.

They also help to lengthen software's life span since they are an aid to efficient change management. The danger lies in lowering the stringency level of the tests without establishing a clear tolerable error rate. One direct consequence of this is the incorporation of test measurements: of code coverage and the like. This is a cause of some satisfaction for me given the current dearth of software development organisations which implement software measurement.

- A phase of active listening with the customer. As I said earlier, this requires the almost full time dedication of one of the customer's employees to manage and check customer requirements.
- A design phase to simplify and correct anything which isn't appropriate.

While this simple review does not aim to match the detailed descriptions to be found in other articles in this issue, perhaps it may serve to comment on certain features of XP in relation to software engineering philosophy. However, there are a number of general considerations concerning XP's applicability which I would like to include.

3 Some of my general reflections about XP

In Beck's book he advocates multi-disciplinarity in IT personal. He proposes that we should forget distinctions between analysts, programmers and test personnel since XP's application requires development personnel to play various roles and be equipped to carry them out efficiently: everyone should take part in analysis, design, programming and tests. From my point of view, this idea is very attractive to say the least. The rigid division of work often leads to a loss of effectiveness and efficiency in software development, regardless of any efforts made to improve teamwork.

However, there is currently a very serious lack of professionals in information technologies, and for software development in particular, in spite of the effect of the economic downturn on employment. In fact, there are studies by organisations such as Forrester Research [Forrester 01] which are already predicting a recovery of the IT business for 2003, based on forecasts of expenses made by corporate managers.

It would, however, be easier to get trained personnel to turn their hand to the agile processes of XP if a serious attempt were made to introduce the qualification or profession of software engineer [Dolado 00], instead of just having a qualification in computer science or, of course, just revamping other qualifications centred exclusively on the knowledge of programming languages without a solid grounding in analysis, design and software testing. By this I do not mean that those who don't have a solid training in software engineering cannot contribute their intelligence towards the correct application of the XP philosophy. Of course, it is possible to train and coach people in the use of XP but I find it hard to believe that this training could be successful with people who lack a solid grounding in software development.

Another of the fears I harbour regarding XP is that it will provide a great excuse for those who like to work in the development process in a state of pure chaos. XP, as its own exponents say, does not consist of relinquishing the notion of a controlled process but rather it is merely the adoption of a strategy of simplification and agility in development work. In spite of the fact that, on occasions, work on models to improve processes have fallen into the error of establishing heavyweight and somewhat rigid processes, the intellectual contributions made by CMM [Paulk et al. 93], SPICE [ISO 98], etc. have been fundamental in bringing about a clear awareness that it is necessary to organise the way we work, and that chaos can only end in tears. It is vital that we do not lose what we have achieved: the awareness (although the we may sometimes fail to put it into practice) that a development process which is well designed and well suited to the problem is fundamental if we want to prevent our projects from failing.

Finally, although XP lays stress on effective communication, I can see a problem in the fact that it always insists on face to face conversation. While a constant and face to face exchange of opinions on a project is a rare commodity in most conventional projects, it is nonetheless true that documentation is also an important asset in the control of the project. In XP I understand that its orientation towards small projects with volatile requirements encourages agility in personal verbal communication. However, what happens in the case of personnel turnover? Or with problems of absences due to sickness or for other reasons? It is true that XP's idea of collective ownership of code (everyone can know and change any part of an application) means greater ease in handling personnel turnover but I believe that we should never pass up the chance to have good documentation so as to be able to understand and maintain an application. Later we can try to remember the content and format of the documentation we think will be suitable for a project or an application (or we can even use automated documentation tools). But, at the end of the day, documentation is necessary, as well as code (the real instrument of communication for exponents of XP).

In this article, I hope I have been able to convey, to the limits of my knowledge of it (and having never attempted to apply it in real projects), the idea that I have of XP. Naturally I am open to any comments on my opinions.

References

[Beck 00]
K. Beck, EXtreme Programming explained. Embrace change, Addison-Wesley, 2000.

[Dolado 00]
J. J. Dolado, "El cuerpo de conocimiento de la Ingeniería del software" (Body of software engineering knowledge), Novática, no. 148, November-December, 2000, pp56–59.

[Dolado/Fernández 00]
J. J. Dolado and L. Fernández (eds.), "Medición para la gestión en la ingeniería del software" (Measurement for software engineering management), Ra-Ma, 2000.

[Forrester 01]
Forrester Research, "End Predicted for IT Sector Slump", 2001.

[ISO 98]
ISO, ISO/IEC TR 15504-1998. Information technology. Software process assessment. Parts 1–9, International Organization for Standardization, 1998.

[McCormick 01]
M. McCormick, "Programming extremism", Communications of the ACM, Vol. 44, no. 6 June, 2001, pp199–201.

[Paulk et al. 93]
  M. Paulk et al., Capability maturity model for software. Version 1.1. Technical Report CMU/SEI-93-TR024, Software Engineering Institute, February, 1993.

Some web resources on XP

http://www.xprogramming.com/
http://www.extremeprogramming.org/

http://c2.com/cgi/wiki
http://www.xpdeveloper.com/
http://www.junit.org/
http://www.armaties.com/extreme.htm
http://www.xp2001.org/:
  the XP2002 conference is currently accepting submissions
http://pairprogramming.com/
http://xp123.com/

XP in Complex Project Settings: Some Extensions

Martin Lippert, Stefan Roock, Henning Wolf and Heinz Züllighoven

XP has one weakness when it comes to complex application domains or difficult situations at the customer's organization: the customer role does not reflect the different interests, skills and forces with which we are confronted in development projects. We propose splitting the customer role into a user and a client role. The user role is concerned with domain knowledge; the client role defines the strategic or business goals of a development project and controls its financial resources. It is the developers' task to integrate users and clients into a project that builds a system according to the users' requirements, while at the same time attain the goals set by the client. We present document types from the Tools & Materials approach [Lilienthal/Züllighoven 97] which help developers to integrate users and clients into a software project. All document types have been used successfully in a number of industrial projects together with the well-known XP practices.

Keywords: XP, Management, Participation, User, Client, Roles

1 Context and Motivation

It was reported that one of the major problems of the C3 project was the mismatch between the *goal donor* and the *gold owner* [Jeffries 00], [Fowler 00]. While the goal donor – the customer in the XP team – was satisfied with the project's results, the gold owner – the management of the customer's organization – was not. It is our thesis that XP, in its current form, fails to address the actual situation at the client's organization in a suitable way. The main stakeholder, i.e. the users

and their management, are merged into a single role: the customer. This one role cannot address the different forces in a development project. The users of the future system know their application domain in terms of tasks and concepts, but they rarely have an idea of what can be implemented using current technologies. Moreover, it is often misleading to view the users of the future system as the goal donor. They are unfamiliar with the strategic and business goals related to a project and, more important, they do not control the money.

Therefore we make a distinction between the role of the user and the role of the client. The users have all the domain knowledge and therefore are the primary source for the application

Martin Lippert is a research assistant at the University of Hamburg and a professional software architect and consultant at APCON Workplace Solutions. He has several years' experience with XP techniques and XP project coaching for various domains and has given a number of talks, tutorials and demonstrations (e.g. ICSE, XP, OOPSLA, ECOOP, HICSS, ICSTest and OOP). He is a member of the XP 2002 program committee. Among his publications are articles for "Extreme Programming Examined" and "Extreme Programming Perspectives" and he co-authored the book "Extreme Programming in Action", which is due to be published by Wiley in July 2002. <lippert@jwam.org>

Stefan Roock is software architect and consultant at APCON Workplace Solutions. He has solid project experiences with object-oriented technologies, architectures and frameworks as well as with XP. Among his current interests are evolution of frameworks and migration of applications, eXtreme Programming, large refactorings and suitable organizational structures for cooperating XP teams. Stefan Roock has given a number of talks, tutorials and demonstrations (e.g. XP Conference, ECOOP, OOP and ICSTest) and co-organizes the XP 2002 workshop on testing techniques. Among his publications are articles for "Extreme Programming Examined" and "Extreme Programming Perspectives" and he is co-author of the book "Extreme Programming in Action", which is due to be

published by Wiley in July 2002. He can be contacted at roock@jwam.org.

Henning Wolf is software architect at APCON Workplace Solutions in Hamburg. He is one of the original architects of the Java framework JWAM, supporting the tools & materials approach. His current interests are eXtreme Programming, architectures for multi-channelling applications and object-oriented technologies. Besides publications on various software engineering topics he is co-author of the book "Extreme Programming in Action", which is due to be published by Wiley in July 2002. He can be contacted at henning.wolf@itelligence.de.

Heinz Züllighoven, graduated in Mathematics and German Language and Literature, holds a PhD in Computer Science. He is professor at the Computer Science Department of the University of Hamburg and CEO of APCON Workplace Solutions Ltd. He is consulting industrial software development projects in the area of object-oriented design, among which are several major banks. Heinz Züllighoven is one of the leading authors of the object-oriented Tools & Materials Approach. A Tools & Materials construction handbook will be published by Morgan Kaufmann end of 2002. Among his current research interests are object-oriented development strategies and the architecture of large industrial interactive software systems. <zuellighoven@jwam.org>

requirements. The client sets the goals of the development project from a business point of view. The client will only pay for a development project if these goals are met to a certain degree.

We begin with a discussion of the roles in an XP project as defined by Kent Beck. We then split up the customer role into the user and the client role. These two roles change the situation of XP projects. While the user can be seen in a similar way to the XP customer, the client role requires more attention. We address the new project situation by using two document types geared to the client role: base lines and projects stages. We show when and how to use these document types and discuss their relation to story cards and the Unified Process (UP).

2   Roles in XP

XP defines the following roles for a software development process [Beck 99]:

- *Programmer:* The programmer writes source code for the software system under development. This role is at the technical heart of every XP project because it is responsible for the main outcome of the project: the application system.
- *Customer:* The customer writes user stories which tell the programmer what to program. "The programmer knows how to program. The customer knows what to program" ([Beck 99], pp. 142f).
- *Tester:* The tester is responsible for helping customers select and write functional tests. On the other side, the tester runs all the tests again and again in order to create an updated picture of the project state.
- *Tracker:* The tracker keeps track of all the numbers in a project. This role is familiar with the estimation reliability of the team. Whoever plays this role knows the facts and records of the project and should be able to tell the team if they will finish the next iteration as planned or not.
- *Coach:* The coach is responsible for the development process as a whole. The coach notices when the team is getting "off track" and puts it "back on track". To do this, the coach must have a very profound knowledge and experience of XP.
- *Consultant:* Whenever the XP team needs additional special knowledge they "hire" a consultant in possession of this knowledge. The consultant transfers this knowledge to the team members, enabling the team to solve the problem on their own.
- *Big Boss:* The big boss is the manager of the XP project and provides the resources for it. The big boss needs to have the general picture of the project, be familiar with the current project state and know if any interventions are needed to ensure the project's success.

While XP addresses management of the software development aspects with the Big Boss role, it neglects the equivalent of this role on the customer side. XP merges all customer roles into the customer role. We suggest splitting up the customer role into two roles: *user* and *client*.

3   The New User and Client Roles

The *user* is the domain expert which the XP team has to support with the software system under development. The user

is therefore the first source of information when it comes to functional requirements.

The *client* role is not concerned with detailed domain knowledge or functional requirements. The client focuses on business needs, like reducing the organizational overhead of a department by 100,000 USD a year. Given this strategic background, the client defines the goals of the software development project ("Reduce the organizational overhead of the loan department by 100,000 USD per year") and supplies the money for the project. The client is thus the so-called *goal donor* and the *gold owner*.

It is often not easy to reconcile the needs of users and clients at the same time. What the users want may not be compatible with the goals of the client. What we need, then, are dedicated instruments to deal with both roles.

4   Story Cards and the Planning Game

We use story cards for the planning game, but we use them in a different way than in the "original" XP, and our planning game differs in some aspects, too. In our projects, users or clients rarely write story cards themselves. They do not normally have the skills or the required "process knowledge" to do so. Typically, we as developers write story cards based on interviews with users and observations of their actual work situation. These story cards are reviewed by the *users* and the *client*. The users must assess whether the implementation of the story cards will support them. They thus review the developers' understanding of the application domain. The client decides which story cards to implement in the next development iteration, and with which priority. To avoid severe mismatches between the interests of the users and client both parties are involved in the planning game. This means that users can articulate their interests and discuss with the client the priorities of the story cards.

Our experience here is clear: users and client will normally reach a compromise on their mutual interests. But whatever the outcome of the planning game is, the decision about what is to be implemented next is made not by developers but by the client.

If a project is complex, there will be an abundance of story cards. In this case it is difficult for users, clients and developers to get the overall picture from the story cards. For this type of project, we use two additional document types: *project stages* and *base lines*. These are described in the next section.

Subgoal	Realization	When
Prototype with Web frontend is running	Presentation of prototype for users	31/3/00
Prototype supports both Web and GUI frontend.	Presentation of extended prototype for users and client	16/5/00
First running system installed	Pilot Web users use Web frontend.	30/8/00
...	...	...

Figure 1: Example project stages

Who	does what with whom/ what	What for	How to check
Roock	Preparation of interview guideline	Interviews	E-mail interview guideline to team
Wolf, Lippert,	Interview users at pilot customer	First understanding of application domain	Interview protocols on the project server
...	...	...	...
Roock	Implement GUI prototype	Get feedback on the general handling from the users	Prototype acceptance tests are OK; executable prototype is on project server

Figure 2: Examples of base lines

5 Project Stages and Base Lines

In projects with complex domains or large application systems, story cards may not be sufficient as a discussion basis for the planning game. In such cases, we need additional techniques to get the overall picture – especially for the contingencies between the story cards. If one story cannot be developed in the estimated period of time, it may be necessary to reschedule dependent stories. We may also need to divide the bulk of story cards in handy portions and make our planning more transparent to the users and the client. We have therefore enhanced the planning game by selected document types of the Tools & Material approach [Roock et al. 98]: *base lines* and *project stages*.

We use project stages and base lines for project management and scheduling. A project stage defines which consistent and comprehensive components of the system should be available at what time covering which subgoal of the overall project. Project stages are an important document type for communicating with users and clients. We use them to make development progress more transparent by discussing the development plan and rescheduling it to meet users' and client's needs. Figure 2 shows an example of three project stages (taken from the JWAM framework development project). We specify at what time we wish to reach which goal and what we have to do to attain this goal. Typically, the project stages are scheduled backwards from the estimated project end to its beginning, most important external events and deadlines (vacations, train-

ing programs, exhibitions, project reviews and marketing presentations) being fixed when projects are established.

Unlike the increments produced during an XP iteration, the result of a project stage is not necessarily an installed system. We always try to develop a system that can be installed and used as the result of every project stage, but we know that this is not always feasible. In large projects or complex application domains, developers need time to understand the application domain. During this period, developers may implement prototypes but rarely operative systems. We thus often have prototypes as the result of early project stages. Another example here is the stepwise replacement of legacy systems. It is often appropriate to integrate the new solution with the legacy system for reasons of risk management. Project stages then produce systems that can and will be used by users. But the project team may also decide not to integrate the new solution with the legacy system, perhaps because of the considerable effort required for legacy integration. In such cases, the project team will also produce installable increments, but it is clear that the increments will not be used in practice. Users are often reluctant to use new systems until they offer at least the functionality of the old system.

Base lines are used to plan one project stage in detail. They do not focus on dates but rather define what has to be done, who will do it and who will control the outcome in what way. Unlike project stages, base lines are scheduled from the beginning to the end of the stage.

In the base-lines table (for example, in Figure 2), we specify, who is responsible for what base line and what it is good for. The last column contains a remark on how to check the result of the base line. The base-lines table helps us to identify dependencies between different steps of the framework development (see "What-for" column). The last three columns are the most important ones for us. The first column is not that important because everybody can, in principle, do everything (as with story cards). However, it is important for us to know how to check the results in order to get a good impression of the project's progress. The second and third columns contain indicators for potential re-scheduling between the base lines and also helps us to sort the story cards that are on a finer-grained level.

The rows of the base-line table are often similar to story cards, but base lines also include tasks to be done without story cards. Examples are: organize a meeting, interview a user, etc.

The way project stages and base lines are actually used depends on the type of development project in hand. For small to medium-size projects, we often use project stages, but no explicit base lines. In these cases, we simply use the story cards of the current project stage, complementing them by additional task cards. If the project is more complex (more developers, developers at different sites, etc.), we use explicit base lines in addition to story cards. If the project is long-term we do not define base lines for all project stages up front, but rather identify base lines for the
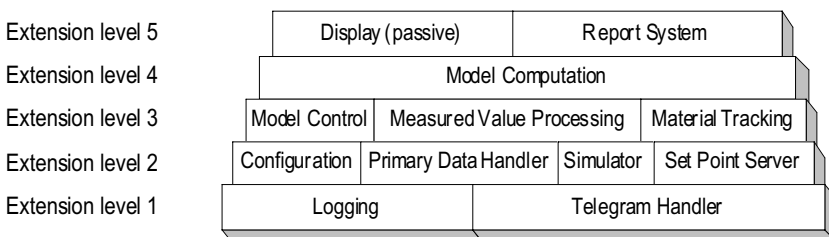

Extension level 5	Display (passive)	Report System	
Extension level 4	Model Computation		
Extension level 3	Model Control	Measured Value Processing	Material Tracking
Extension level 2	Configuration	Primary Data Handler	Simulator	Set Point Server
Extension level 1	Logging	Telegram Handler	

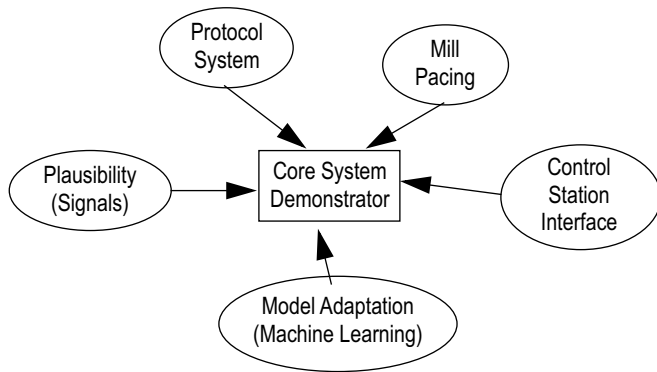**Figure 3:** Example core system with extension levels

Figure 4: Example core system with specialized systems

current and the next project stage. Since a project stage should not be longer than three months, we work on a detailed planning horizon of from three to six months.

It is often a good idea to sketch the entire system as guideline for the project stages. We describe the concept of core system and specialized systems in the next section in order to provide an application-oriented view of the system architecture.

6 System Architecture

In line with the project stages, we divide the software system into a *core system* with *extension levels* [Krabbel et al. 96]. The core system is an operative part of the overall software system which addresses important domain-related needs. It is developed first and put into operation. Since the core system is usually still quite complex, it is subdivided into extension levels which are built successively. An example of a core system with extension levels is shown in Figure 3 (taken from the domain of hot rolling mills). The upper extension levels use the functionality of the lower extension levels. This way, we get an application-oriented structure that is useful for planning and scheduling. It is obvious that the lowest extension level must be created first, followed by the next-higher one, and so on.

Specialized systems are separated from the core system. They add well-defined functionality. An example of a core system with specialized systems is shown in Figure 4 (again taken from the domain of hot rolling mills). The specialized systems are drawn as circles.

Since specialized systems only depend on the core and not vice versa, we can deliver an operative and useful core system very early on and get feedback from the users. In parallel, different software teams can build specialized systems. Adhering to the one-way dependency of specialized systems, we achieve a maximum of independence among the special systems. They can be created in any order or even in parallel. Obviously, the core system has to provide the basic functionality for the whole system because it is the only way for the specialized systems to exchange information. The core system will usually provide a set of basic communication mechanisms allowing information transfer between different parts of the overall system.

The concept of core system and specialized systems can easily be used in the planning game. Users and client get an

impression of the whole system and can negotiate on the different values and priorities (users' needs, client's goals, technical constraints) in order to reach a compromise on the project's development schedule.

In addition, project stages are used to control the project's progress and timelines relating to the overall plan.

7 Conclusion

We have discussed the roles in a XP project as defined by Kent Beck. Based on our experience, we split the XP customer role into two roles: *user* and *client*. The user is the source of application knowledge, while the client defines the project goals and supplies the money for the project. Both parties must be integrated into the development project. We have shown how this can be done with the help of modified story cards, projects stages, base lines and an adapted planning game.

We do not suggest using all the presented new instruments for every project. They should be used as part of an inventory or toolbox, together with the familiar techniques defined by XP. We then use the instruments required for the project in hand. If the project situation is not complex, we will not burden the project with the additional roles and document types. But if the application domain or the project is highly complex, the sketched extensions to XP will be worth while.

Selection of the proper instruments from the toolbox may be difficult for the project team because we are not yet able to provide detailed guidelines. Evaluating project experience to provide such guidelines for tool selection will be one of our future tasks.

References

[Beck 99]
    Kent Beck: eXtreme Programming Explained – Embrace Change. Addison-Wesley. 1999.
[Fowler 00]
    Martin Fowler: The XP2000 conference.
    <http://www.martinfowler.com/articles/xp2000.html>. 2000.
[Jeffries 00]
    Ron Jeffries: Extreme Programming – An Open Approach to Enterprise Development.
    <http://www.xprogramming.com/xpmag/>. 2000.
[JWAM]
    The JWAM framework. <http://www.jwam.org/>
[Krabbel et al. 96]
    A. Krabbel, S. Ratuski, I. Wetzel: Requirements Analysis of Joint Tasks in Hospitals, Information systems Research seminar. In Scandinavia: IRIS 19; proceedings, Lökeberg, Sweden, 10–13 August, 1996. Bo Dahlbom et al. (eds.). – Gothenburg: Studies in Informatics, Report 8, 1996. S. 733–750, 1996
[Lilienthal/Züllighoven 97]
    C. Lilienthal and H. Züllighoven: Application-Oriented Usage Quality, The Tools and Materials Approach, Interactions Magazine, CACM, October 1997
[Roock et al. 98]
    Roock, S., Wolf, H., Züllighoven, H., Frameworking, In: Niels Jakob Buch, Jan Damsgaard, Lars Bo Eriksen, Jakob H. Iversen, Peter Axel Nielsen (Eds.): IRIS 21 "Information Systems Research in Collaboration with Industry", Proceedings of the 21st Information Systems Research Seminar in Scandinavia, 8–11 August 1998 at Saeby Soebad, Denmark, pp. 743–758, 1998