Computer Supported Cooperative Work **13:** 91–117, 2004.
© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice

ADRIAN MACKENZIE[1] & SIMON MONK[2]
[1]*Computing Department, Faculty of Applied Sciences, Lancaster University, Lancaster LA1 4YR, UK (Phone: +44(1524)594193; Fax: +44(1524)594273; E-mail: a.mackenzie@lancaster.ac.uk);*
[2]*SabiliLtd, Manchester, UK*

Abstract. This paper discusses Extreme Programming (XP), a relatively new and increasingly popular 'user-centred' software design approach. Extreme Programming proposes that collaborative software development should be centred on the practices of programming. That proposal contrasts strongly with more heavily instrumented, formalised and centrally managed software engineering methodologies. The paper maps the interactions of an Extreme Programming team involved in building a commercial organisational knowledge management system. Using ethnographic techniques, it analyses how this particular style of software development developed in a given locality, and how it uniquely hybridised documents, conversations, software tools and office layout in that locality. It examines some of the many artifices, devices, techniques and talk that come together as a complicated contemporary software system is produced. It argues that XP's emphasis on *programming* as the core activity and governing metaphor can only be understood in relation to competing overtly formal software engineering approaches and the organisational framing of software development. XP, it suggests, gains traction by re-embodying the habits of programming as a collective practice.

Key words: co-ordination work, ethnography, extreme programming, software development techniques, user-centred design

1. Introduction: Studying an XP project

UK cinemas currently show an advertisement for SAAB cars in which a golfer tees off over the edge of an ice crevasse, a skier slaloms down what looks like a rock face, a high-diver dives into a frozen pool, etc. The car itself, the ad implies not without a touch of irony, also copes with similar extreme conditions. (A second advertisement, usually shown a few minutes later, makes this irony explicit when it describes how the frightened, freezing golfer lost several hundred golf-balls during the shooting of the ad.) The ad plays on the wider popularity of *extreme* sports. "Extreme environments" in Antarctica or on K2, "extreme sports" such as "aggressive inline skating", snow-boarding, "vert biking", extreme bouldering, "extreme ironing", and now extreme programming or *XP* (not to be confused with XP Windows, or the Athlon XP chip, etc.): why would computer programming and software development need to become "extreme"? Extreme sport involves calculated risk-taking with minimal technical backup. Extreme sports tend to do without much technical equipment. For instance, extreme bouldering involves climbing large rocks with minimal assistance from ropes, carabiners, harnesses, pitons and cleats. In each case, a controlled exposure to risk, especially of bodily injury, occurs. Could software development become extreme in this sense? How could it strip down to a bare minimum of equipment?

Addressing these questions, this study of an Extreme Programming project in progress undertakes two tasks. Firstly, it evaluates some key aspects of XP in a real world setting. It asks about the process of stripping work down to a minimum: why cut back technical equipment to a minimum? Complementary to this question, it asks: what kind of work goes into implementing XP at a specific site? How does it gain traction amongst a group of developers so that their interactions and work process can be effectively called "extreme programming"? What kinds of work do they have to do to keep XP working as a process? Does that maintenance work, and the skills it draws on, appear in the official accounts of XP? Here what actually happens in an XP project – the shuffling through packs of cards, sitting around a table together, announcing "flag's up", arguing about how code executes and using certain kinds of specific software devices – will be the focus of attention.

Secondly, the paper asks a broader evaluative question about contemporary practices of software engineering: how do software projects, software engineering and development cope with a world or a "new economy" increasingly imagined in terms of human and code mobility (Kelly, 1998)? Some conventional software engineering methods heavily instrument the process of software development, and in effect, attempt to batten down against a stormy outside world. XP takes a different tack, and this tack might have implications for collaborative work practices more generally.

In a different but not unrelated context, Star and Ruhleder argued that:

> Experience with groupware suggests that highly structured applications for collaboration will fail to become integrated into local work practices. ... Rather, experimentation over time results in the emergence of a complex constellation of locally-tailored applications and repositories, combined with pockets of local knowledge and expertise. They begin to interweave themselves with elements of the formal infrastructure to create a unique and evolving hybrid. This evolution is facilitated by those elements of the formal structure which support the redefinition of local roles and the emergence of communities of practice around the intersection of specific technologies and types of problems. (Star and Ruhleder, 1996, p. 132)

Along these lines, I am suggesting that we regard XP as a "unique and evolving hybrid". It introduces some formal structures and processes which redefine local roles, generate locally tailored applications, repositories, knowledges and expertises. The core of my argument is the XP gains traction to the extent that it manages to 're-embody' individual programming practices as a process of collaboration and organisation of software development. The term 're-embody' is slightly awkward, but it avoids talking about XP as an abstract or general methodology.

It emphasises, moreover, that much of the appeal of XP is predicated on quite mundane practices. These practices affect the notoriously complicated workplace relations between programmers and those who regard managing programmers as something like ‘herding cats’.

1.1. WHY ANOTHER ETHNOGRAPHY FOCUSED ON SOFTWARE DEVELOPMENT?

In the worlds of software development, extreme programming has recently begun to make a name for itself. XP conferences, websites, how-to books and articles, list-groups and local user-groups have established themselves over the last few years and continue to expand rapidly at the time of writing (see http://www.xprogramming.com/index.htm). Most articles and books either provide a formal, somewhat idealised account of XP as a software development process, or closely reflect the personal experience of participants or proponents (e.g Beck, 2000; Wake, 2001). Furthermore, most of these documents are tutelary: they contain injunctions, descriptions and discussions in lesser or greater detail of how to do XP. The current interest in XP needs to be set against a background of several decades of work on software development methodologies and software engineering.

The name XP, Extreme Programming, suggests a twist here. The very name suggests that XP is not a software *engineering* methodology but something focused on *programming*. Now from the standpoint of software engineering, programming has long been a problem-laden activity. As Paul Quintas has observed, ‘software engineering is an oppositional concept which is contingent on the identification of conventional development processes that are regarded as unsatisfactory’ (Quintas, 1996, p. 90). Programming stands right in the middle of these unsatisfactory processes because it too often relies on *ad hoc* craft practices. Beginning with seminal texts such as Frederick Brooks *The Mythical Man Month* (Brooks, 1975), various attempts have been made to remedy programming practices. Various organisational, management, documenting and designing techniques were suggested by Brooks. During the 1980s, CASE tools and other formal modelling techniques were used. In the 1990s, object-oriented languages and software engineering methodologies such as the Rational Process and Unified Modelling Language (UML) continued to address the persistent problems of programming. In evaluating XP, we should be aware that by promoting *programming* as a core activity and using it as guiding metaphor, XP explicitly lays down a challenge to the project of software engineering as a discipline (in both senses of the word).

1.2. WHAT WILL IT LOOK AT?

When this study began, software development on *Deskartes Universal* was well underway at KMS plc, a software production house located in Manchester Science Park, UK. *A* small somewhat fluctuating team of software developers, testers and managers (6–12 people) were working on ‘Universal’, as the developers called it, a *knowledge management* system. The fieldwork phase of the study involved visiting KMS 2–3 times a week for 3 months, watching pair-programming, team meetings, reading code repository logs, code and design documents. Many interactions were audio-taped and transcribed.

Described in basic terms, KMS’ *Universal* system aims to help organisations access the knowledge embedded in documents by organising documents within a heavily keyworded, cross-referenced repository which can process and “learn” from natural language search queries. In addition, the system sets up ways of capturing “expert” replies to common queries. Once captured in *Universal*, knowledge becomes accessible through natural language queries typed into forms in a web browser. At the centre of *Universal* lies neural network technology that processes documents and queries. It brings up relevant information more directly than most standard search engines. According to KMS promotional materials (brochures and website), *Universal* could be important to call-centres, help-desks or wherever in organisations the same questions are asked repeatedly in endlessly slightly different variations. In implementation terms, the important attributes of this system for our purposes include its heavy reliance on Java, html and some Javascript as programming/scripting languages, its use of a distributed component-based architecture (Enterprise Javabeans running on an application server) and its reliance on standard web-browser software as a user interface. As we will see, these features of the development environment impinge fairly strongly on the day-to-day work of the development team. At the same time, like many other software development projects at the time of writing, by using Java and the Enterprise Javabeans framework, KMS was hoping to sidestep many of the configuration management issues that have long dogged software development teams as they sought to keep their software running on a range of different platforms (Grint, 1997).

This was the first time XP had been tried at KMS. For that reason, XP at KMS was very definitely a ‘unique, evolving hybrid’ rather than a fully developed organisational application of a method. Significantly, however, the leader of the developer team, PL, mentioned early on that code quality in a previous outsourced implementation of *Universal* had been so poor that a new *Universal* team had to begin again almost from scratch. (A long printout of some of the worst code from the outsourced project had been stuck to the wall in the developer’s area.) Symptomatically, the outsourced work had explicitly relied on a fairly up-to-date modelling technique, the Unified Modelling Language, UML. KMS had produced a full set of specifications in the form of UML diagrams (showing the architecture of the software and the planned sequence of operations). Yet programmers working on another continent (India) had interpreted those diagrams differently. They had, as PL pointed out, ‘reverse-engineered’ the diagrams to produce a system that met some of KMS expectations but not others. In at least a general sense, the move to XP, driven by the interest and keenness of two main individuals (PL and

SM), and helped by the relatively small team-size, was yet another response to the less-than-ideal behaviour of programmers.

One effect of the move to XP was immediately obvious in the layout of the developer's suite at KMS. Most of the developers sat facing inwards around tables pushed into one large rectangle in the centre of the room. As usual, computer monitors, laptops, PalmPilots, telephones, Coke bottles, snack food, network cables and the odd book (Dilbert cartoons, or Martin Fowler's *Refactoring*) lay scattered over the tables. Somewhat unusually, there were no cubicles or partitions, and developers sat alongside each other so that two people looked onto one keyboard and screen (in order to carry out *pair-programming*; see discussion below). Occasionally, usually after saying "flag's up", a developer went around to the end of the table to use a computer placed there. That computer carried the words "Build Machine" written with a marker pen on a piece of paper stuck to it. At other times, they went to a noticeboard covered with cardboard cards of three different colours – blue, orange and pink – and added or took down cards. Sometimes, a developer also sat outside this large communal table at a work station off to one side. Against a back wall, large bookcases held several hundred computer books, manuals and periodicals. In one corner of the room near the entrance, in a glass-walled office, S, the project manager worked. Further away, just outside the entrance to the developer's area, the software testers work area was situated. Also near the entrance, a small meeting room almost filled by a large table, chairs and a whiteboard was frequently occupied by developer's "stand-up meetings" or planning meetings.

Some of these features (the snack food, the mobile phones and handhelds, the thick semi-popular computer press "how-to" books) are found in any software development team. Many others – the "build machine," at the end of the rectangular table, the noticeboard with coloured cards, the chairs pushed together, and the large central cluster of tables – are specific features of XP implementing recommendations of official XP manuals. Others again – the separate testers area – are local KMS adaptations and hybrids. How is the environment at KMS a 'complex constellation of locally tailored applications, pockets of knowledge and repositories? What is the relation between this 'constellation' and the emphasis that XP puts on programming?

2. When the cards are down: Extreme Programming as a game of software development

2.1. "PROGRAMMING ITSELF" (WITH JUST A FEW SMALL THINGS ADDED)

More than one copy of books written by Kent Beck (2000) can be found lying around KMS, and different members of the team told me that they have read them. PL, the project coach, recommended reading them to me at the end of my first visit. As (Boden, 1994) points out, we have to acknowledge that actors in organisations account for their own actions partly in terms of theories of organisation and management found in such books, so let's start with the "official" account of XP written by Kent Beck.

Beck proposes that the best way of understanding XP lies in the metaphor of programming. In a slightly paradoxical statement, he writes:

> XP uses the metaphor of programming for its activities-that is, everything you do looks in some way like programming: Programming in XP is like programming with a few small things added, like automated testing. However, like all the rest of XP, XP development is deceptively simple. All the pieces are simple enough to explain, but executing them is hard. (Beck, 2000, p. 97)

Why does he say "XP uses the metaphor of programming"? How can software projects be re-organised by taking the programming as a metaphor? Programming as a metaphor for programming? To decode this statement, we need to be aware that software engineering has introduced many activities that do not "look like programming". For instance, structured software design methods "normally involves producing graphical system models and results in large amounts of documentation" (Sommerville, 2001, p. 58), which then has to be maintained by someone. The irony here, and perhaps this irony lies behind the name 'XP', is that XP bases its organisational and management practices on the metaphor of bare programming. This is the very activity that conventional software engineering sees as the root of problems, essentially because it can be an arcane and somewhat uncontrolled craft activity.

Software engineering tries to treat the problem through the documenting and modelling processes. Yet design documentation, as other ethnographies of engineering have shown, creates trouble for engineering projects in general because it does not look like software development:

> [W]ithin the particular work-a-day world of engineers efforts are made to upgrade the production of documentation in order to ensure that it is built into the engineer's work. Upgrading the work is a general device that is used to ensure that "dirty work" gets done and within software engineering there are a number of such upgrading devices that are used in this respect. (Sharrock and Button, 1996, p. 382)

We could see the popularity of software engineering methods such as Rational Unified Process as just such 'upgrading devices' Making "use cases" as in the Rational Unified Process, or drawing class diagrams using UML (Unified Modelling Language) for instance, might not look much like programming to programmers focused on writing lines of program code, compiling them, running them, and then debugging code. KMS' previous attempt to build *Universal* (discussed above) was based on just such an upgrading: they produced the UML diagrams which programmers in India were meant to implement in code, saving costs on programmers in UK. When Beck says that "XP uses the metaphor of programming for its activities", we read him as saying that XP takes the iterative cycle of writing, compiling and running code as a metaphor for the whole process of software development not just for "cutting code." Arguably, XP reasserts the core value of programming on the grounds that programmers necessarily *embody* the practice of cycling through coding, compiling and running code at a fairly basic level. Their competence, craft skills and even identity as programmers rests on inhabiting that process. XP seeks to extrapolate from this implicit embodied practice to a more explicit software development process.

Many formal software development methodologies tend to place coding as just one step on a multi-stepped waterfall or a multi-looped spiral process. During more conventional software engineering processes, the phases of risk analysis, development of a statement of requirements, prototyping, simulations, development plans, requirement validation, etc. remain discrete and precede the final steps of coding, unit testing and acceptance testing (Sommerville, 2001, p. 54). Of course, nearly all software design methods practically involve shuttling back and forth to revise earlier design decisions and adjust the specifications in the light of changed circumstances. The spiral model makes these revisions and adjustments explicit. Yet the overall emphasis of software engineering has been to downplay programming as an activity and code as an artefact. This might be one sense in which XP is "extreme". It diverges from this vision of central, hiearchically controlled engineering process. In XP, every part of the designing and building of the system is oriented by both the metaphors and practices of coding. Hence the evaluative question which lies at the heart of this paper and that has, we would suggest, implications for many day to day practices in the software production would be: what actually happens when people turn away from the models and process of software engineering to focus on programming? Do their projects lose traction? Do problems of design integrity or excessive 'thrashing' of communication that Brooks and many others have addressed for the last 30 years return with a vengeance?

2.2. CARD AND NOTICE BOARD, CODE AND SCREEN

Coloured cards on a table and some cards pinned on a noticeboard were the first things we saw at KMS that had anything to do with XP. Again the irony cannot be ignored: a team building a system that would permit organisation knowledge to be extracted and effectively outsourced to call centres was itself working with pieces of cardboard in an intensely collaborative environment. If we wanted to evaluate the reassertion of programming as core activity, it might be useful to track the movements of these cards at KMS. As per Beck's specification, the cards at KMS came in three varieties – orange for "stories", blue for "tasks" and pink for "defects". An implicit hierarchy and work processes has already been coded into the colour of these cards. An orange story card will, in principle, have been written by a "customer" or user of the system; a blue task card will have been written by a developer, and a defect card either by a system tester or user. Usually the different kinds of cards were kept apart. In the following weeks, we saw cards filled out, shuffled, dealt out, handed in, "spawned" (as in "let's spawn a card for that"), displayed, swapped, counted, stored, taken back, written on, and put away. The cards circulated constantly through different members of the team, were visible in different places, and used for different purposes.

Why resort to something as archaic as index cards? It seems like a very trivial thing to bring cards into play as a way of steering software development. Surely, compared to the sophisticated project management and design tools available to software developers, these very old-fashioned looking index cards would be relatively clumsy, inflexible and limiting? The cards are printed on stiff paper as fairly simple forms with no more than 10 lines of texts in the main description field. Often, the card will only have one line of writing on it (e.g. "to load security level frame for current result") as well as a date, a story, task or defect number, and space for brief details about completion times and dates. The simplicity and limited capacity of these cards to carry information should give us pause for thought.

Cards lay on tables alone and in packs. Cards were pinned to the board in rows and sometimes in envelopes. People walked to and from the board, pinning cards on it and taking them off it. Individual cards lay beside developers workstations. On her desk, SA, the "tracker" copied numbers and dates from stacks of cards into a spreadsheet. A printout appeared each week on the noticeboard as a summary of project progress. The production, distribution and consumption of these cards threaded through much of the work of the *Universal* team. During their work, which at this stage we too are still tracking as a kind of card game, we could ask: how do the cards move around? What propels them from meeting table to noticeboard, from noticeboard to developer table, from developer table back to noticeboard, from noticeboard to tracker's desk, etc.?

It is well-known that during card-games, packs of playing cards can intensely focus the attention of card players. Like playing cards, XP cards carry different values. Intense focus and concentration on the values of cards can be seen amongst the KMS software developers at times. Like most card games also, these cards will be shuffled and dealt in rounds (see discussion below of the "release", "iteration" and "task" loops). They circulated through different hands playing different roles: customer, developer, "tracker" (the person who keeps track of the rate at which the team works), and tester. The different colours immediately differentiate some of the major kinds of work done in XP. Writing an orange story card on the table in the meeting room represents a relatively potent act. A story card, usually because it involves a substantial number of subsidiary tasks, will be the object of close scrutiny. Moreover, because it crosses the boundaries between the development team and the outside world, story cards call for careful evaluation. They cannot be tossed around lightly because of the large amount of time and work potentially involved in turning a story written by the customer into a working part of the software.

2.3. CARDS DO NOT GO QUIETLY

It would be impossible to document all the ways in which people move cards around. However this movement has a few common, salient features. Importantly, the cards do not flow automatically. The cards do not move silently. Talking steers the movement of cards. What kinds of talk? Questions, comments, argument, instructions, and directions all surround the movement of cards. Unlike playing cards whose value is fixed by rules of the game, the value of a story is open to negotiation. The "points estimated" field in the bottom corner of this card will reflect in a very condensed form a whole series of judgments, evaluation, estimations and guesses performed by the software developers during a planning meeting. The weight of a story card, how much it is worth will be worked out by the developers through a complex set of negotiations. At KMS, the story cards are read to the developers during a planning meeting. Afterwards, the tasks composing the story need to be written out. For instance, at the start of a planning meeting, PL, the XP "coach" writes the stories for discussion on the whiteboard. Only after several hours of fairly intense discussion can the field "Points Estimated" in the lower right hand field of a story card be filled in with a figure like "4.0G". This simple figure summed up hours of talk about how the story could be implemented as a set of tasks. Developers draw cards from stacks of blanks (especially blue task cards), write on them and pile them up again in substantial new stacks. A story undergoes translation into tasks through spoken interactions between different members of the team. The following conversation explores fairly tentatively how this translation could occur, and what work it will involve. The coach, PL, stands near the whiteboard and points to a line a writing:

> PL: This is the nasty controversial one. [Laughter]. Alright, the detailed selection tags that we've got from there [pointing at another story on the whiteboard]. Now what we kind of figured is that tags will be some fairly ordinary tags from the JSP page [Java Server Pages, a part of the webserver infrastructure used to construct the web-browser interface to *Universal*], we'll probably have some scripts that trigger things going on. So, you're probably going to be doing a bit of reuse in both cases. And this reuse might actually now be direct reuse with the exception that this tag [pointing at a list on the board] you don't want to pre-populate.
> R: Sorry, I missed something. Umm, the reuse of tags thing is going to actually be quite small isn't it?
> PL: Yeah
> R: It's the JSP page that may well have to be working out how it works again?
> PL: Well, you should be able to make the code easily using any code that's lying around.
> R: A quarter for each one then.

This is the first mention of actual programming work we have seen. The gist of the code talk with which this transcript beings concerned whether the task involves new code or reuse of old code. The last line of the transcript seems rather abrupt and unrelated, but in fact it was where the whole exchange (and many others like it) was heading. "A quarter for each one" means a quarter of an "ideal developer days work" for each task. So the exchange between PL and R, two experienced developers, arrived at a numerical estimate for the amount of coding work involved. That value was then be written on the top right hand corner of a blue card. R, in fact, immediately reached across the table and began writing on some blue cards. At the same time, all the other developers in the meeting heard how this particular piece of work could be carried out. References to and recollections of other code already written for previous tasks forms an important component of the dialogue. Hence, the same exchange both brought new cards into circulation, and at the same time, the future programming work associated with those cards had been announced within the team. A possible path for the card to move had been laid down, and an allocation of time has been agreed upon (as represented in the score). Later that month, when a developer goes to the noticeboard and takes that task card, her work will be 'scaffolded' by the talk that surrounded the card's creation. The consequences of that act of taking on a task by getting hold of the card will have been anticipated and collectively estimated.

Standing back a little from the table talk that surrounds card play, we could view the cards as kinds of lightweight signs which mark the outcome of very mobile, flexible dialogical interactions occurring in planning meetings or during conversations around the work-stations. Talk around the table crystallises in the point estimations, the new task cards, and the brief notes written on the cards. At the same time, the constellation composed of developers and cards has been somewhat restructured. Their collective order has been slightly changed and the material environment has taken on a slightly more complicated texture through the introduction of another simple artefact, a task card. Thereafter, cards circulate as tokens or indexical signs of that structuring of both the team and possibly *Universal* itself dialogically achieved in the meeting. Problems, unanticipated difficulties and implications will nearly always emerge, but at least their advent will be buffered by the collective understanding of the task already developed in the group.

Developers wrote cards, collected them, handed them round and swapped them. Despite the the primacy given to programming as both metaphor and core activity, the XP card game is an important way to generate coding talk. To say that it is a 'little thing' as Beck does may be to downplay its importance. The point-estimation work focuses that talk on just what kind of coding work is involved, something that programmers are sensitive too. Talk surrounding the creation of cards and cards will often give rise to much more talk, especially around the developer's work table. But other relations and responsibilities were also attached to cards. In particular, holding a task card usually meant working on that task for anywhere between a few hours and a few days, and translating the brief written synopsis on the card into some code. What does that work of translation look like? Is XP at KMS really just a matter of talking about building software by handing around cards?

3. Writing code: nested loops or 'harmful gotos'?

The *Universal* project faced familiar problems. Frequent staffing changes, downsizing of some parts of the company, corporate mergers with other software development companies, shifts in management direction, local aftershocks of the dot-com crash earlier in the year, and technical problems in developing and supporting a (distributed) system all affected the project. How does XP, with its looping process, help the team cope with this kind of environment? If conventional software engineering presumes some kind of stability in the organisation setting (otherwise the whole process breaks down), does XP cope with dot-com style turbulence any better?

The focus of software design methods rests on making work predictable in relation to timing, and ordering work-processes so that their timing and outcomes can be measured and steered. A mixture of different technical, social and organisational problems tend to thwart predictions and resist steering, and these have been the subject of much analysis by software engineers. XP explicitly regards problems in timing and prediction as normal rather than exceptional. The subtitle of (Beck, 2000) is *Embrace Change*. In what ways does it embrace change? Is this subtitle about changing software development methods or about a software development method adapted to change?

Like other engineering methods, XP divides work into phases or stages. Like the spiral model of software design, the stages run in loops. However, XP protects itself against unpredictability by planning work in relatively small nested loop structures ('iterations', 'releases') and by downplaying the central planning and design framework advocated by Brooks (Brooks, 1975, p. 79) and reflected in many subsequent approaches to software engineering. In principle, XP retains flexibility to rapidly initiate different moves in relation to changing circumstances by reducing the scale of its plans. Each level of nested loop takes into account different kinds of unpredictability, runs over different time scales and makes use of different ways of organising communication within the developer team. XP seeks to organise the gamut of software development in loops or iterations of different durations:

Loop name	Duration	Activities
Release	2–6 months	Release planning meetings – writing "stories", estimating stories
Iteration	1–4 weeks	Iteration planning meetings
Task/debugging	1–4 days	Stand-up meetings, task allocation
Writing unit tests and program code	1–2 hours	Pair programming, writing test cases and code

Unlike the spiral model, the loops do not get wider as the system moves into production. Their "radius" remains constant. The loops of the spiral model

(Boehm, 1988) and XP also differ in that already in its first release loop XP promises to put the system into service in the “real world” and maintain it there. Beck writes:

> The ideal XP project goes through a short initial development phase, followed by years of simultaneous production support and refinement. (Beck, 2000, p. 131)

The “iterations” that occur over the months observed at KMS were specifically marked by certain kinds of planning meetings in which difficulties and problems were voiced in discussion and conversation. (Importantly, some difficulties the project encountered were not the subject of planning meetings. XP still anticipates certain kinds of problems and not others). These included meetings planning the next release, meetings planning the next iteration, and stand-up meetings for day-to-day problems. As periodisations of work, the loops were, generally, nested inside each other, so that release planning meetings occurred least frequently (every month or so), followed by iteration planning meetings (every week or so) and finally, most often, stand-up meetings (every 2–3 days, but sometimes more often). Most of the time, developers at their workstations were working within that lowest level “loop” , and their cycles through the loop were marked by cards being pinned to and taken down from the noticeboards. Task cards on the board would be taken by developers. When they were completed, they would be put into a “ready to be tested” envelope and tested by the testing team. Periodically, SA, the project manager and XP “Tracker” (Beck, p. 144), collected all the completed cards and copied the completion dates and other details into the spreadsheet she used to keep track of the “project velocity.” These calculations were used to estimate the amount of work that the team could do in the next iteration of the release. A printout of the spreadsheet itself was usually pinned to the noticeboard.

This rule-governed work, of course, did not go totally according to plan, partly because this was the first time KMS had tried XP development. Acceptance tests, for instance, rather than coming near the end of a project, should come at the end of the first release, and this first release will occur a few months into the project, not years. The question of who would write the acceptance tests for *Universal* and when they would be written was a slightly vexed question at KMS. In theory the customer should be the one to write them, but at KMS the customer had not been ‘trained’ enough to do that, with the result that in one meeting, PL, the coach, came in with a stack of printed pages which he referred as “the first acceptance tests we have ever had.” Professional Services, the group upstairs who would have to actually support *Universal* at customer sites, had complained that there were no manuals or documents they could use to find out how to use to the system. Rather than write those manuals, the development team was going to begin writing acceptance tests from which Professional Services could infer what the system should do. PL had started doing that at home the previous evening, hence the stack of pages. While the relatively small size of KMS as an organisation meant that this gap in documentation was not a major problem, the proposal that XP is just 'programming with a few small things added' began to look a bit shaky at this point. Quite a bit has to be added. Perhaps this incident corroborates Beck's own claim that although the components of XP are simple, holding them all together is hard. In any case, given that KMS was trying XP for the first time on the *Universal* project, we would naturally expect it to hit unexpected bumps.

4. Keeping in the loop takes work: Feedback and gauges

Leaving aside the more obvious teething difficulties such as this, we want to focus on the question of the organisation of programming work. In this part of the paper, we will look at five quite small incidents in which key XP practices such as automated unit testing, pair programming, collective ownership of source code and the continuity between production and maintenance came into contact with local programming practices and local roles at KMS. These incidents show, we argue, that the way XP uses programming as a metaphor for the co-ordination of programming work has both powerful leverage and significant problems.

The feedback loops in the XP process can be seen as flowing from the metaphor of programming as an activity involving constant feedback between writing, compiling and testing code. It could also be seen as drawing on the role of loops as central code constructs which control the flow of data in programs. Almost every software developer would be conscious at some level of the role of loops in programs. Hence basing a software development process on loops could have quite a strong appeal. Yet almost any project management technique involves constant monitoring of how actual progress compares to schedule. Is feedback actually accomplished in XP in ways that differ from the somewhat hierarchical administrative approaches found in many conventional software engineering projects? How do the KMS team maintain the periodicity of these loops? Kent Beck suggests that after programming steering is the other major metaphor for XP:

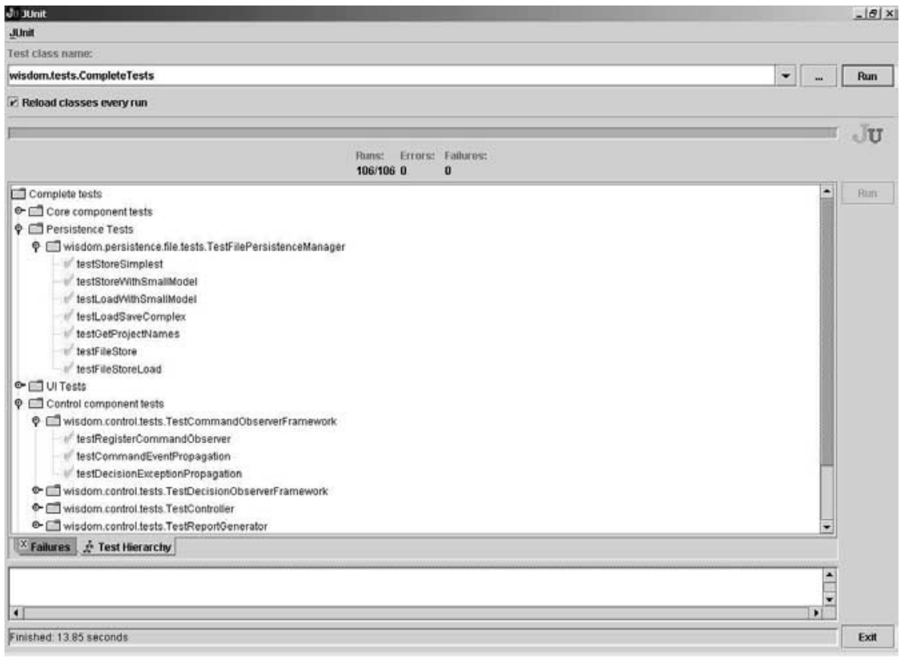
> [S]oftware development is like steering, not like getting the car pointed straight down the road. Our job as programmers is to give the customer a steering wheel and give them feedback about exactly where we are on the road. (Beck, p. 28)

Again, the implicit contrast here runs between hands-on work (programming or steering) and more distant or detached forms of control (managing, pointing). It is also interesting that in this formulation, feedback is given to the 'customer' rather than the programmers themselves. As we have already seen, XP explicitly responds to the idea of constant change, but the question of how XP deals with change still needs to be addressed. "Everything in software changes" Beck writes (p. 28) and for that reason, founds XP on a set of core values ("communication", "simplicity", "feedback" and "courage"). Let's leave aside the question of values and look at the practices that organise work within loops. These include "the planning game", frequent small releases, metaphor, simple design, testing, refactoring, pair programming, collective code ownership, continuous integration, 40-hour week, on-site customer, and coding standards (Beck, p. 54). How does these practices 'give the customer a steering wheel'? More importantly, in what ways do they alter the kinds of control and communication within and around software development?

Specific practices regulate coding work done at KMS. While we cannot examine the implementation of all the practices in this discussion, we can look at some of the loops that provide feedback, and see whether they differ from other common programming practices. A number of crucial elements of the workings of the KMS team converge on units tests. The team collectively "owns" the source code for *Universal*. Anyone can modify any part of the code. But source code created or modified during the course of a task (the smallest chunk of work done by a programming pair – see discussion below) can only be committed into the team's source code repository when all the *unit tests* for both that code *and* the rest of the application run cleanly. Other groupwork software development environments impose stricter controls over who can change code and when they can change it. For instance, IBM's *Visual Age* development environment relies on notions of privileges, code ownership and publication permissions to control code editing. The use of a source code repository such as CVS (Concurrent Versioning System) stands at the core of XP since it allows constant and untrammelled sharing and modification of code.

While the use of source repositories is very common, and needless to say, almost indispensable to programming teams, XP adds to that practice a rule specifying that only code that has passed all the unit tests can go into the repository. An XP team collectively accepts that their work is founded on correctly written and functioning unit tests. The unit tests exist as Java code and they themselves are written by developers and shared via the CVS. The test code is deeply embedded in the source code for *Universal* rather than standing apart from it. Rules concerning unit tests counterbalance the complete freedom of access to code. Developers commit code to the repository only on the condition that all the unit tests have succeeded. The use of unit tests (that is, tests that verify the behaviour of some relatively small chunk of the code) allows melding of changes to the shared code to occur relatively uneventfully. Without a way of melding changes to code, group ownership of source code and scripts would become unworkable. Conversely, without commonly shared code, the idea of writing unit tests that must run fully before declaring any task finished would be meaningless.

When they retrieve code from the CVS repository, the KMS developers assume that it will pass all existing unit tests. As a first step, they often run all the existing tests to see that the code passes them. They check out the latest source code from the CVS repository, compile it and then see if all existing unit tests succeed. Completion of their own task (as written on a blue task card) consists in developing some new unit tests and some new code that passes those unit tests. The tests and the code will then be put back into the code repository. KMS use a simple piece of software, *JUnit*, to automatically run the tests.



wisdom.tests.CompleteTests

```
Runs:    Errors:  Failures:
106/106  0        0

Complete tests
  Core component tests
  Persistence Tests
    wisdom.persistence.file.tests.TestFilePersistenceManager
      testStoreSimplest
      testStoreWithSmallModel
      testLoadWithSmallModel
      testLoadSaveComplex
      testGetProjectNames
      testFileStore
      testFileStoreLoad
  UI Tests
  Control component tests
    wisdom.control.tests.TestCommandObserverFramework
      testRegisterCommandObserver
      testCommandEventPropagation
      testDecisionExceptionPropagation
    wisdom.control.tests.TestDecisionObserverFramework
    wisdom.control.tests.TestController
    wisdom.control.tests.TestReportGenerator

Finished: 13.85 seconds
```

Figure 1. JUnit test runner.

In *JUnit*, a solid green line shows a completely successful set of tests. A red line means some test has failed. The KMS developers follow the practice of never putting program code into the CVS repository (maintained on a dedicated computer at the end of the developers' table) without having first obtained the green light from *JUnit*. While seeing that green line does not guarantee that they have completed their own task as required (since they may not have written adequate unit tests for their own code), it does allow them to gauge that the code they changed does not "break" any existing tests.

The *JUnit* window shows something else important. Developers nearly always work on a localised part of software project. At KMS, developers work on different parts of the system, and rarely on the system as a whole. (Only the semi-automated work of building a release of *Universal* could be said to involve the whole system.) Yet their work nearly always impinges on other parts of the system under development. Rather than just testing their own code, they run all the tests written to date by the team to ensure that hidden conflicts between their changes and other parts of the system become visible as early as possible. Running all the tests before committing code constitutes an important feature of their daily, even hourly, work. It means that system integration problems which might otherwise only appear late in the development cycles appear early and often. The practice of "testing" merges here with continuous integration. As (Latour, 1996) writes: "[b]y accumulating little solidities, little durabilities, little resistances, the project ends up gradually becoming somewhat more real" (Latour, 1996, pp. 45-46). The focus on unit tests tries to sidestep the usual problems of testing. Brooks, for instance, writes how regression testing is a theoretical ideal that can only be approximated in practice and is very costly (Brooks, 1975, p. 122).

4.1. PAIR PROGRAMMING: TALKING CODE INTO EXISTENCE

Stating the central importance of programming to the XP model, Kent Beck writes: "At the end of the day, there has to be a program. So, I nominate coding as the one activity we know we can't do without" (Beck, 2000, p. 44). Given the centrality of coding to software development, the XP question is: how to code? Yet what counts as 'coding' is open to contest. We have just seen something of that in relation to the rules about writing and running unit tests. Writing unit tests as part of developing code is hardly a universal accepted part of everyday coding practice. If XP indeed manages to convince KMS programmers to write the unit tests before they even write code for *Universal* itself, the activity of coding will have undergone considerable changes. Unit testing works against the unexpected side effects of ongoing code changes. It also allows production and maintenance to go on simultaneously, at least in principle. (Again, this departs from most other software engineering methods that split production and maintenance into different phases.) The green or red line on *JUnit* shows pretty quickly if existing tests have been broken by newly added code. But running *JUnit* obviously assumes the existence of some code, and it presumes that code has been written so that it passes all other unit tests. Here again, the modest claim of 'programming with a few small things added' begins to look a little understated. Actually, coding according to these rules involves quite a lot of organisation and negotiation.

A second change touchs on the question of who does coding. XP plays down myths of heroic *individual* coding efforts carried out by taciturn headphone-cocooned programmers buzzing along on a constant intake of "Jolt" Pepsi. It replaces the myth with an unusual organisation of programming work in pairs, or as *pair programming*. For most programmers, pair programming could well be the most startling feature of XP. It seems to require an abdication of coding autonomy and its connotations of individual creativity and expression. At KMS, almost all work on *Universal* was being done in pairs. The pairs changed daily. Occasionally, a team member would work alone away from the group at a work station in the corner of the developer's suite, but not on coding. (The two members of the team doing *acceptance testing* almost always worked alone.) Even the hardware installed in the developer's area reflected a commitment to pair programming. Development work was done on laptop computers connected via a laptop dock to an external monitor. Various programming practices had to be accommodated in the pairing of programmers together. Although some pairs might have formed more frequently than others, a good deal of mixing and swapping of partners occurred. Sometimes laptops moved around between pairs. Some developers seemed to keep a closer hold on a particular development machine than others. One developer, D, for instance, installed a complete Linux development environment on his machine, including the substantial application server and web-server components needed to run *Universal*.

The amount of *talking* that pair programming generates is surprising. Usually programming and software development is not described as a talking activity. Let's follow some of the talking that occur during a typical pair programming session at KMS. Here SM and m (a relatively new member of the team) have formed a pair to deal with a defect. They take the card labelled "Template.java defect" from the board where a tester, MA, has pinned it and move to a machine. This machine has a single screen which they both look at. SM sits at the keyboard. As a first step, SM runs the unit tests.

> SM: Right, let's just see how far we go with this and then go back.
> M: So there's no specific place for running the system?

The green line comes up on JUnit.

> SM: . . . Ah, there we go.
> SM: This is the version that should have everything substituted into it.
> M: Right, so is that the form of the tag that the template [yeah]?
> SM: [Taking a piece of scrap paper and writing on it] What happens is that you have a template file with most JSP pages, and in that file it has all the variable bits delimited by underscores. So that by the time it gets rendered into html, all the underscore something or rather underscore should have been converted into something real.
> . . .
> SM: The way they substitute URL is . . . [looks for another piece of paper] here it is. Now, this is all right, this is a ready drawn piece of paper. Before . . . Right, so you have a template, you do [writing] 't = new Template'. And you give it a string full of ordinary text, with things delimited by markers that are going to substituted. That creates your template. You then do a whole load of sets. . . . You then do String results = t.value. t.value( ) which is the method we've changed, is the thing that actually substitutes all the underscore thingy-ma-jigs underscore into actual values.
> Right, that's not a problem at all if you do it the way we used to do it, which is that you go through the hashtable of things that you're going to substitute. Because when you do one of these, it basically puts it in a hashtable. We way we used to do it is that you'd go through the hashtable looking for underscore name underscore. When you found it, you do a global . . . Sorry, when you found it, you'd go to the hashtable, iterate over all the keys, for each of the keys, you'd go and look in here and do a global search and replace on underscore name underscore for Vladimir. With me?
> M: No
> SM: [Laughs]

We could follow this interaction further as SM explained to M the underpinnings of the defect they've taken responsibility for. In this case the dialogue unfolded as a kind of lesson. SM and M sat at the computer but they did not actually write code for the program. Code fragments were written on a piece of paper as SM explained a part of *Universal*'s workings to M. No working code comes from that, but M begins to understand how code could be written to deal with the defect. Understandings of the system design, how it works and how it can be made to work differently often passed between pairs rather than through written documentation. But how did code get written? A great deal of it was written without comment, but frequently the developers had to speak with each in order to understand what happened or should happen in some part of *Universal*. In the following snippet, a pair (NA and SM) tried to explain some anomalous database behaviour:

1. NA: So you think it's throwing an exception and the "finally" [refers to a particular line in the program code] is not executing for some reason.
2. PL: Finally always executes.
3. SM: Even when Jbuilder [NA: JUnit?] JUnit is . . .
4. PL: What?
5. SM: is polling the . . .
6. PL: Yeah, "finally" *always* executes.
7. NA: I don't think JUnit has some magical control over this.
8. PL: No.
9. NA: Yeah, it's just a classloader. When it invokes a method, it probably just does it in a "try" and a "catch." And if you don't throw it up, you don't ever see it, right?
10. SM: Doesn't it give you a bit more of a [performance]?
11. NA: I don't think it's going back to the old database for whatever reason.
12. SM: Well it must be.
13. NA: After the second run.
14. SM: It must be something we've done.
15. NA: Oooh
16. SM: It must be!
17. NA: Go ahead, try it.
18. PL: Do you want a stand-up?

Several interesting strands of pair programming as a practice intersect in this example. SM and NA are discussing why a particular set of unit tests fail. As we have seen, success with unit tests usually means completion of a programming task. In this exchange, they struggle to decide whether the cause of the test failures lies with the database or whether the problem stems from the way *JUnit* runs the tests. At this point, PL, an experienced developer, weights into the exchange and directly asserts that a certain Java code construct can only work in one way: "finally always executes" (line 2, line 6). NA argues that *JUnit* does nothing "magical" or hidden from their understanding of the situation. SM thinks that *JUnit* does do something they don’t understand. Hence, they are compelled to tak about how *Junit* actually works. This is why a few lines into the exchange, the discussion moves back into a more basic question about how Java code executes (line 9).

Having agreed on basic questions about Java, they return to the question of the database’s involvement. This allows NA then to advance his interpretation of *JUnit*’s behaviour, and in particular to reinforce the point that *JUnit* cannot be responsible for the current problem. In response, SM moves to a slightly different and less specific position. He suggests that the problem must “be something we’ve done.” NA does not accept this move and in turn suggests that they no longer talk about what happens but try it out in practice. PL, for his part, goes in the opposite direction. He switches roles slightly also at this point into XP “coach” mode, and asks, “do you want a stand-up?” In other words, he proposes escalating the problem into an impromptu but slightly formalised general discussion amongst the whole development team. (As it turns out, SM and NA did not want that. They resolve the problem by running some code.)

Apart from PL’s role in prompting them about a ‘stand-up’, little in these incidents reflects anything specific to XP. What they do suggest, however is that the pair-programming arrangement increases the chances that problems or difficulties will quickly become an object of collective attention and discussion. It was uncommon at KMS for such discussions to go unnoticed by other developers. Often they would join in.

4.2. MOVING DISTANT HORIZONS CLOSER

As well as installing gauges such as *JUnit* to show that looping conditions have been met, and wrapping coding in conversation, XP also removes some important thresholds and stages that other software development processes highlight. System integration, an important step in waterfall and spiral models, really does not figure as a discrete step in XP. That particular source of uncertainty felt by all software development teams on the day they first put all the sub-systems together disappears. The question “will the system hang together?” doesn’t loom large because the continuous integration entailed in writing and running unit tests has pre-empted it.

It would be wrong to imagine that the idea of continuous integration works effortlessly. Many small incidents at KMS show the difficulties in translating that idea into practice:

I: What was the issue then with ST?
SM: Just saw something that surprised us when we updated our files.
I: Local copy?
SM: Yes. We’ve got a builder script, and ST changed the way that it works so that it actually changes the date of the files. It actually touches the files.
I: And that meant that you weren’t sure whether you had the latest version?

SM: Well, it meant we weren't sure whether we should check them in or whether we should leave them alone.

This incident concerns how continuous system integration actually occurs. SM and NA had made some changes to the code, run the tests, seen *JUnit* go green and were ready to place their work in the shared CVS repository. They go to the CVS machine at the end of developer's shared table. Just as they are poised to commit their changes, the WinCVS program shows some files with red icons. They expected to see them all white. That would have reassured them that nothing needs to be done to them. (Again the WinCVS program serves as a sort of gauge, this time on the state of the repository.) Red icons mean the files have been modified but not committed or finally checked in by someone else. If ST and NA commit their work now they might overwrite changes recently made by someone else. Instead a conversation ensues in which SM asks ST to help him interprets this anomaly in the continuous integration. It turns out that from now on, certain classes of files (those "touched" by the "builder script") will always look as they have been changed when in fact they haven't really.

The cost of continuous system integration includes complicated negotiations between members of the development and the development of shared, contingent interpretations of what "gauges" mean. In this case, from now on, red icons on certain selected build files in the CVS repository probably only mean that the build script has been run. They can be safely ignored. Two points emerge from this incident. Firstly, the proximity of developers to each other around a shared table does not arise just from a commitment to the shared "values" of communication or feedback. Without the possibility of quickly constructing verbal agreement on what those icons meant, SM and NA's work would be on hold. They could not commit their changes to the code repository without risking overwriting someone else's work. Secondly, removing a major stumbling threshold such as the conventional system integration phase means re-distributing all the configuration work needed to integrate the system throughout the duration of the project. Instead of "big bang" integration, XP promotes "steady-state" expansion of the system.

4.3. "WE'RE IN SET-UP HELL": CONFIGURATION WORK IN XP

Inevitably some work on *Universal* fell outside the planned XP loops. Sometimes the gauges failed. Sometimes iterations were not fully complete before company priorities changed. For instance, one morning we arrived to find three developers at work configuring a different version of *Universal* (the "Corp X. Branch") as a demonstration installation for two customers. This configuration task was not scheduled on any of the task cards which normally describe work to be done on the project.

SA tells us that they need to have the demonstration ready by lunchtime. It is now just after 10 A.M. Around the main table, normal activity was buzzing. But the *three* developers seated along one side of the table (R, D, & B) were troubled by the failure of the database configuration scripts to work. They need to run these scripts to create tables in a relational database. R decides to find ask the whole team about it:

R: Eh, [shouting], can I just get your attention for a moment.
[Some talk continues]
R. and G as well: . . . [all talk stops] We're running the Corp X. branch at the moment. So it's the older version. And we're having problems with the database stuff. It's, it's not right. Can anyone think of the reasons why if we get the latest database creation scripts, all the schema and other stuff, if anything has changed that could cause any issues?
SM: [from the other side of the table] – Yes.
R: It has?
SM: There's all the security stuff that's gone in there [describes "Entity beans" columns]
R: No, no. If you're adding columns it's ok, its only trying to use columns that exist. But it's whether things have changed . . . SM: I'm not sure if that's completely true. I don't understand. You shouldn't be having a problem anyway . . . PL: Is the problem still just the Problem table?
R: No, we're having a RollBackException in EJBException in insert TriggerOnCommunityTable
PL: insert TriggerOnCommunityTable. . .?
SM: Did you run the whole thing? I thought you were just going to cut out the bit with the rollback table.
R: No, we just ran the whole thing.
SM: Right, you needed just the Problem table.
ST: I've run the whole scripts last week
SM: The new scripts?
ST: No, for the Corp X.
PL: But SM said his script's on the main branch.
R: Oh. Right.
ST: I don't understand why you were on the new branch.
R: No, it doesn't matter.

At this point, R turns back to the computer (a laptop connected to an external monitor), starts up a WinCVS program and retrieves the database scripts from a different section of the CVS repository, the Corp X. branch. Everyone else returns to their own work and conversations. What happens in this fairly typical episode? How did R, D & B come to be apparently using the wrong database script? What does this kind of episode say about the ideal work-process of nested loops outlined in the XP how-to manuals?

The place of *configuration-work* remains unclear in XP. The three developers happened to be working on a configuration task. This is regarded as isolated or exceptional work in XP: on a previous occasion, weeks earlier, working on this same branch, and trying to get it working with an upgraded version of the

BEA *WebLogic* Application Server, SM, working with B, says somewhat despairingly, "we're in set-up hell." In other words, configuration work has no official place in XP, partly because it can't be very easily estimated using task cards and partly because it can't be easily unit tested. Although there may be information 'configuration gurus' in the KMS team (one team-member, ST, for instance, knows a lot about setting up and configuring the *Weblogic* commercial application server on which *Universal* runs), none of the loops, gauges or dialogical forms that XP promotes explicitly address configuration. Configuration-work or "set-up hell" incessantly troubles software systems, as well as many other technical practices. Here it interrupts for a few hours the nested-loops of the XP software development process. Forcing a jump outside the nested-loops of task, iteration and release, setting up a demonstration installation of the system consumes a lot of energy. Like a harmful "goto" statement, configuration-work can end up in tangles. If we regard configuration work not as some unlucky accident that befalls software development from the outside, but as part of the normal course of events, what does this episode, whose resolution we have yet to see, show us? In some respects, XP attempts to minimise configuration work by maintaining a common source code, a full and automated set of unit tests, and, less clearly, a set of configuration information (such as database scripts) consistent with the source code. However, while the common source code and fully running unit tests eliminate some configuration problems, other configuration-work remains unavoidable. We could ask: what place does the work the developers were doing that morning have in the XP process?

4.4. THE PROBLEM OF THE DOPPELGÄNGER: SIMULTANEOUS VERSIONS

The work of the three developers this morning lies on a branch different to that of the main team to a certain extent. They have not chosen to do go out on a limb like that. Their planned work, as described by the task cards, has been shunted onto the branch by a management directive. A possible customer for the *Universal* system has been promised a demonstration by midday. The problems they have in configuring the system stem from an exit outside the planned XP loops. Whereas XP predicates the release of a succession of versions, it does not envisage working on different versions at the same time. The ”Corp X. branch“ is a slightly different version of the system existing and being developed at the same time as the main branch. It has not come into existence just this morning. It has been in existence for weeks now.

Again, we can ask how XP handles this quite legitimate and common need to have different versions of the same system in development at the same time. Sometime later, R still struggling to get the system up and running, asks PL:

R: PL,
PL: Hello
R: Is there any chance in getting everything working last week for us on this machine that either you or ST (who isn't here at the moment)

> SA: [project manager, arriving to stand behind the developer's] ST's there.
> R: could have updated anything on our machine, the Java source on our machine, about that Urgency [a feature of *Universal* that allows queries to be prioritised as urgent] or the notification.
> PL: No. Ah, we didn't actually have anything to do with what you were doing the whole week.
> R: There's going to come a point, right, where, well, we've not even applied it yet. We've got differences on our machine, different to the Corp X. branch in CVS. And we've not made those changes.
> PL: On the Java code?
> R: On the Java code. And we've not made those changes.
> PL: [to ST] did you change any code on R's machine?
> ST: Did I change any code on his machine?
> R: Last week, when you were getting stuff working, about Urgency
> PL: He couldn't have done it on Friday because he was at Corp X.
> R: On any day.
> ST: I wouldn't have done anything on your machine anyway.
> R: It's not, I'm not pointing a finger or anything. BR was doing stuff directly on my machine. He might have changed the Java code.
> ST: I wouldn't change it. Well no, it wasn't me.
> R: [pause, then to D & B, who are sitting by]: I mean, worst case scenario is it all works and we commit what I've got on my machine into that branch of Corp X.
> PL: But the whole question remains of how it got changed.
> R: Well more, why and what else has been changed that we don't know about that we're going to commit into CVS.
> PL: Did you do a commit Friday?
> R: I've not done a commit or anything with the Java stuff. The Java stuff has worked fine.

A bit later, after the developers look through some more files on their machine, this episode continues:

> . . .
> R: We've just done a diff between what's on my machine and what's on the Corp X. branch, and there's differences.
> SA: Well they should be exactly the same.
> R: I've only seen a difference on the Query EDA
> ST: I, I recut that. [Pause] I changed the Query entity and committed that on Thursday. I made changes to get it to work with the Corp X. branch.
> D: That's it!
> R: Right, cool. So we're just reading CVS wrong. Which is beautiful. It makes perfect sense. All we've got to do then . . . in fact, we can just do "get latest".
> 3 September 2001

In this second scene, more people have been drawn into the incident. Now rather than three developers (already one more than the usual programming pair found in XP), 5 or 6 people have become involved, including the project manager, the team-leader and other developers. The XP-set-up of the workspace, with nearly all the developers seated around one large oblong table, facilitates that involvement. Again talk, driven by a problem elicits memories, recollections, and alternative understandings of the configuration of the development set-up (CVS) which eventually allow a configuration problem to be resolved.

5. Conclusions

At the beginning of this paper we asked: could software development become 'extreme' in the sense popularised in extreme sports? More specifically, is programming the extreme activity in XP? Several striking features of XP emerging from this study show that in practice, reducing software development to an 'extreme' in this sense is problematic and laden with irony.

Firstly, as the discussion of the card game suggested, XP involves a lot more than programming. The card games knit together in a rule-governed process a very disparate set of work processes and relations involving management, 'the customer' or client and all the members of the software development team. All these parties participate in software development. At least to a certain extent, the game spans the usual boundaries between project managers and software developers. It shifts a great deal of the project management work over to the programmers themselves. The cards simply and effectively contrive to specifiy and keep track of work done by different participants in the *Universal* project. They also act as lightweight, mobile tags that index shared understandings of what *Universal* will do, how it will do it and when it will do it. Without these cards, the nested loop structure through which XP work was organised at KMS could not have been maintained. Gaining possession of, swapping and signing off on cards represented visible markers of progress and allowed almost immediate visual inspection of collective work progress.

Secondly, as the discussion of the various programming incidents suggest, programming work or *coding* becomes central to XP only against a background history of trials with other software development methods. The emphasis on programming as core activity and as metaphor for all other activities has a somewhat ambivalent status. On the one hand, by treating all organisational activities on the model of programming (loops, cycles), XP rhetorically repackages project management in ways that are already familiar and perhaps more palatable to programmers. In this sense, it re-embodies programming work as a way of doing project management. On the other hand, the injunctions to do unit testing, pair programming and accept collective ownership of the source code entail substantial changes in the day-to-day work practices of programmers. Again, doing 'programming with only a few small things added' is not so simple in practice.

These practices, as some of the incidents show, trigger new kinds of collaborative problems for which local work-arounds and customizations continually need to be negotiated.

The artifices used to co-ordinate the work of software development in XP – unit tests, card games, code repository – directly address the problems of co-ordinating work against an unstable background of economic and organisational change.[1] Something as simple as a set of cards, or a small piece of software such as *JUnit* can create irreducibly new and unique forms of activity when they manage to combine and synthesise different facets of organisational life and work process effectively. Yet it is tempting to say that the card games, the co-location of the team, writing unit tests, pair-programming and collective ownership of the source code are all strategems that intensify the collaborative work of programming at the cost of a somewhat closed relation to outside the team. While the 'customer' officially belonged to the team, in KMS' case, he was rarely in the developer's area.

The metaphor of programming grafts effectively into software development processes because it plays on familiar, embodied practices of programming and pits them against more conventional central and hierarchical methods of software engineering. It draws on the fairly indisputable necessity of programming to propose an explicitly minimal and hands-on approach to software development. The stress on the core value of programming or coding has to be seen against the background of overt attempts to engineer the craft-processes of coding and manage them from above. This rhetorical move on the part of XP has important consequences for how we think about co-operative activities more generally. (Suchman and Trigg, 1996), in an analysis of how AI researchers used whiteboards, suggested that

> like any product of skilled practice, the formalism inscribed on the board leaves behind the logic of its own production and use, seen here as collaborative craftwork of hands, eyes, and signs. But analyses of situated practice ... point to the contingencies of practical action on which logic-in-use, including the production and use of scenarios and formalisms, inevitably and in every instance relies. In this way such analyses provide an alternative to idealized (sic) formulations of reasoning as disembodied mental operation. (Suchman and Trigg, 1996, p. 173)

In their study, Suchman and Trigg wanted to effectively counter-balance the abstract models of mental operation predominant in AI research with the concrete, localised and embodied practices of AI researchers. Through their analysis of craftwork, they sought to render visible the "dark matter" of AI research, especially as it transpires in the course of conversations and whiteboard writing. Coming from a completely different motivation, XP does something similar for software development. Rather than leaving 'behind the logic of its own production and use', XP effectively turns the craftwork of hands, eyes, keys, screen and signs into the basis of an effective formalism.

A final word on XP as an extreme activity: as we have seen, XP orients software development projects to a metaphor of "programming plus a few small things added." The few small things include the cards games and talk, the unit testing and code sharing practices, and pair programming. In fact, these are hardly small things. Like the SAAB advertisement I mentioned at the outset, an element of irony plays out in the metaphor of "extreme". Like extreme sports, software development and many other collaborative activities do face relatively unstable and risky environments. XP explicitly tackles the "risk" in software development not by armouring itself with heavy formalisms. As we saw in the nested loops, it addresses the question of how to bring a flexible organisation of time into centre stage in software development. At the same time, as the "set-up hell" episode illustrates, and as the SAAB ads show in their own way, engineering projects including software development projects do not deal solely or even principally with physical, natural or even technical risks. Some things can never be anticipated.

Note

1. In an article that sought to move CSCW debates around interaction between humans and artefacts in new directions, Marc Berg wrote: 'What I am after ... is how distributed and interrelated entities can create new forms of activity, irreducible to and spanning over the actions of isolated elements' (Berg, 1999, p. 376). Extreme Programming, I would argue, constitutes an illuminating instance for the development of this line of "relational" investigation. XP involves highly collaborative activity. It insists strongly on the value of co-location, effectively in the space of a single room.

References

Anderson, R. and Sharrock, Wes (1993): Can Organisations Afford Knowledge? *Computer Supported Cooperative Work* (CSCW), vol. 1, pp. 143–161.

Auer, Ken and Roy Miller (2001): *Extreme Programming Applied.* Addison-Wesley Publishers Ltd.

Bannon, Liam and Susanne Bodker(1997): Constructing Common Information Spaces. In J.A. Hughes, W. Prinz, T. Rodden and K. Schmidt (eds.): *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work.* Dordrecht: Kluwer Academic Publishers.

Beck, Kent (2000): *Extreme Programming Explained. Embrace Change*. Addison-Wesley Publishers Ltd.

Berg, Marc (1999): Accumulating and Coordinating: Occasions for Information Technologies in Medical Work. *Computer Supported Cooperative Work*, vol. 8, pp. 373–401.

Brooks, Frederick P. (1975): *The Mythical Man-Month. Essays on Software Engineering*. Reading, MA: Addison-Wesley Publishing Company.

Button, Graham and Wes Sharrock (1996): Project Work: The Organisation of Collaborative Design and Development in Software Engineering. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 5, pp. 369–386.

Clark, Andy (1997): *Being There. Putting Brain, Body, and World Together Again.* Cambridge, MA: MIT Press.

Curtis, Bill, Herb Krasner and Neil Iscoe (1988): A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* (November), vol. 31, no. 11, pp. 1268–1287.

Grint, Rebecca A. (1997): Doing Software Development: Occasions for Automation and Formalisation. In J.A. Hughes, W. Prinz, T. Rodden and K. Schmidt (eds.): *ESCW97 Proceedings of the Fifth Conference on Computer Supported Cooperative Work*. Dordrecht: Kluwer Academic Publishers.

Latour, Bruno (1996): *Aramis, or the Love of Technology*, C. Porter (trans.). Cambridge, MA: Harvard University Press.

Lynch, Michael (1993): *Scientific Practice and Ordinary Action: Ethnomethodology and Social Studies of Science*. Cambridge: Cambridge University Press.

Ó Riain, Seán (2000): Working for a Living Irish Software Developers in the Global Workplace. In M. Burawoy (ed.): *Global Ethnography Forces, Connections, and Imaginations in a Postmodern World*. Berkeley and Los Angeles: University of California Press.

Quintas, Paul (1996): Software by Design. In R. Mansell and R. Sliverstone (eds.): *Communication by Design: the Politics of Information and Communication Technologies*. Oxford: Oxford University Press, pp. 75–102.

Sommerville, lan (2001): *Software Engineering* (6th edn.). Harlow: Pearson Education & Addison Wesley.

Star, Susan Leigh and Karen Ruhleder (1996): Steps Toward an Ecology of Infrastructure: Design and Access for Large Information Spaces. *Information Systems Research* (March), vol. 7.1, pp. 111–134.

Succi, Giancarlo and Michele Marchesi (2001): *Extreme Programming Examined*. Addison-Wesley Publishers Ltd.

Suchman, Lucy A. and Randall H. Trigg (1996): Artificial Intelligence as Craftwork. In *Understanding Practice: Perspectives on activity and context*. Cambridge: Cambridge University Press.

Wake, William C. (2001): *Extreme Programming Explored*. Addison-Wesley Publishers Ltd.