European Journal of Information Systems (2006) 15, 197–210

www.palgrave-journals.com/ejis

Customising agile methods to software practices at Intel Shannon

Brian Fitzgerald[1], Gerard Hartnett[2] and Kieran Conboy[3]

[1]University of Limerick, Castletroy, Limerick, Ireland; [2]Intel Communications Europe, Shannon Industrial Estate, Shannon, Co.Clare, Ireland; [3]Department of Accountancy and Finance. NUI Galway, Galway, Ireland

Correspondence:
Brian Fitzgerald, University of Limerick, Ireland.
Tel: +353 61 202720; Fax: +353 61 202734;
E-mail: bf@ul.ie

Abstract
Tailoring of methods is commonplace in the vast majority of software development projects and organisations. However, there is not much known about the tailoring and engineering of agile methods, or about how these methods can be used to complement each other. This study investigated tailoring of the agile methods, eXtreme programming (XP) and Scrum, at Intel Shannon, and involved experienced software engineers who continuously monitored and reflected on these methods over a 3-year period. The study shows that agile methods may individually be incomplete in supporting the overall development process, but XP and Scrum complement each other well, with XP providing support for technical aspects and Scrum providing support for project planning and tracking. The principles of XP and Scrum were carefully selected (only six of the 12 XP key practices were implemented, for example) and tailored to suit the needs of the development environment at Intel Shannon. Thus, the study refutes the suggestion that agile methods are not divisible or individually selectable but achieve their benefits through the synergistic combination of individual agile practices; rather, this study shows that an *a la carte* selection and tailoring of practices can work very well. In the case of Scrum, some local tailoring has led to a very committed usage by developers, in contrast to many development methods whose usage is limited despite being decreed mandatory by management. The agile practices that were applied did lead to significant benefits, including reductions in code defect density by a factor of 7. Projects of 6-month and 1-year duration have been delivered ahead of schedule, which bodes well for future ability to accurately plan development projects.
European Journal of Information Systems (2006) **15,** 197–210.
doi:10.1057/palgrave.ejis.3000605

Keywords: agile methods; software development; XP; Scrum; method tailoring; method engineering; Intel

Introduction

Agile methods are perhaps the latest initiative in the software field to be posited as the 'silver bullet' to help address the key problems in software development, namely that software takes too long to develop, costs too much to develop, and does not work very well when eventually delivered. Much has been made of the demand for flexible development methods (Lee & Xia, 2005) that can handle constant change and evolving complexity (Lycett & Paul, 1999). Agile methods are suggested to be 'just enough' method as they seek to avoid prescribing cumbersome and time-consuming processes that add little value to the software product and actually elongate the development process (Highsmith, 1999; Fowler & Highsmith, 2001). While the principles underpinning agile methods are generally accepted as neither new nor radical paradigm shifts in software

Received: 18 May 2005
Revised: 19 June 2005
Accepted: 9 January 2006

development, there is some debate as to how these principles are applied in practice. Some argue that parts of these agile methods can be cherry-picked, deviated and replaced (McBreen, 2003), while others suggest that the overall combination of individual agile practices achieves a synergistic effect which addresses the inherent problems in software development (Schwaber & Beedle, 2002). These advocates argue that these methods cannot be applied *a la carte*, but must be applied in their entirety to achieve the desired effect.

A related stream of research has focused on the tailoring of software methods to the actual needs of the development context. Factors such as organisational issues (Doherty & King, 2001), distributed teams (Sarker & Sahay, 2004), or the existence of legacy systems (Chae & Scott Poole, 2005) often require the use of a different method, or at least changes to the existing method. This has taken two forms principally – method engineering and contingency factor approaches. In the case of method engineering, a meta-method process is followed whereby methods are precisely constructed from existing discrete pre-defined and pre-tested method 'fragments (Kumar & Welke, 1992; Harmesen *et al*., 1994; Hidding, 1996). The contingency factor research suggests that specific features of the development context should be used to select an appropriate method from a portfolio of methods (Iivari, 1989; Avison & Wood-Harper, 1991). One marked feature of both the contingency factor and method engineering research is that it has been largely deductive in nature, employing theoretical and conceptual arguments to suggest how methods should be tailored or constructed. Very little is available in terms of the practical application of these concepts in real software development practice. In the software field, practice is often ahead of research, and thus much can be learned from examining good practice. The application of agile methods in Intel Shannon is especially pertinent as it represents an industrial product development setting, where experienced software engineers adopted an *a la carte* approach to XP and Scrum. Many of the reported benefits of XP to date have arisen from studies in academic university environments (e.g. Muller & Tichy, 2001; Hedin *et al*., 2003), and therefore lessons learned from its application in a real software development context are invaluable, as few such studies have been published.

Given the above, we were interested in investigating the use and tailoring of agile methods in actual practice. Specifically, our research objective was to investigate:

- How agile methods are used and tailored in practice
- How agile methods can be combined to address the overall software process

We chose an in-depth case study to investigate this. The site in which we carried out our investigation was Intel Shannon. Our findings suggest agile methods can make a significant contribution to quality in terms of reduced defect density and delivery within schedule. Also, these methods are 'developer friendly' and committed usage has grown bottom-up among developers rather than usage mandated by management. XP and Scrum were found to be very complementary with XP particularly useful for the technical development stages, whereas Scrum provided the necessary overall project management process. Also, both methods were extensively tailored to meet the precise needs of the development context, reinforcing the view that all methods need to be tailored for successful deployment.

The paper is laid out as follows. In the next section, previous research on agile methods is presented, in particular studies to do with XP and Scrum, the two methods in use in Intel Shannon. Also, research on method tailoring is presented. Following this, the research method adopted for the study is presented. Then, details of the use and tailoring of XP and Scrum at Intel Shannon are presented. Finally, the theoretical and practical implications of the study are discussed.

Agile methods and method tailoring

Empirical research shows that method use in software practice is rather limited, and those developers who do use methods tend to use different combinations and parts of methods rather than following all the steps required by a particular method (Fitzgerald, 1996; Hidding, 1996). In fact, an empirical study by Fitzgerald (2000) found that only 6% of developers rigorously adhere to methods. Much research focused on the inability of methods to handle people factors (Boehm, 1984; Brooks, 1987; Glass, 1991). Fowler (2000) claims that the bureaucratic nature of methods has slowed development to the extent that developers are forced to abandon them. Fitzgerald (1994) refers to 'goal displacement' whereby developers become preoccupied with following the method and lose sight of the fact that their goal is to develop a software product.

The formation of the Agile Alliance in 2001 and the publication of the Agile Manifesto (Fowler & Highsmith, 2001) formally introduced agility to the field of software development. Those involved sought to 'restore credibility to the word method' within the context of software development (Fowler & Highsmith, 2001). The manifesto conveyed an industry-led vision for a profound shift in the conventional software development paradigm. Agile methods have been defined as 'a collection of philosophies that enable IT professionals to work together effectively' (Ambler, 2002). Such methods have been described as 'young and fast-moving' (Constantine, 2001), that 'compromise between no process and too much process' (Fowler & Highsmith, 2001) and 'that are flexible enough to smoothly adapt to changes in requirements and delivery schedules' (Aoyama, 1997; Jacobson, 2002). They 'mirror today's turbulent business and technology environment' (Highsmith & Cockburn, 2001) and are related to what Lee & Xia (2005) refer to as flexible systems development. They 'dispense with all but the essentials' (Boehm, 2002) and do 'not do anything that is a waste of time' (Highsmith, 1999) and Cockburn

(2001) adopts a narrower meaning of an agile method, by suggesting that alternatives to agile development arise 'as soon as the development team focus their attention on rigorous, predictable, repeatable, defect-free, traceable or even fun software development'.

It is important to emphasise that agile approaches are not anti-method, rather they operate on the lean principle of 'barely sufficient methodology' (Highsmith, 1999). The change in emphasis from the traditional approaches is summarised in the following value-trade-offs (Fowler & Highsmith, 2001):

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan

Advocates of the agile approaches recognise that both sides of these value statements are relevant to software development. However, they choose to emphasise the first part of each statement as more important than the second part. The overall principles underpinning the agile approaches are summarised in the agile manifesto (Fowler & Highsmith, 2001).

Many different methods have been labelled as agile, such as eXtreme Programming (XP) (Beck, 1999), Dynamic Systems Development Method (DSDM) (Stapleton, 1997); Scrum (Schwaber & Beedle, 2002); Crystal (Cockburn, 2001); Agile modelling (Ambler, 2002); Feature Driven Design (Coad *et al.*, 1999); Lean Programming (Poppendieck, 2001), and perhaps even the Rational Unified Process (RUP) (Kruchten, 2000). Two of the most popular and widely adopted agile methods are XP and Scrum. These were the basis of our investigation in this study, and an overview of both methods is provided next.

Overview of the XP and Scrum methods

As already discussed, two of the most popular and widely used agile methods are XP and Scrum, and both of these are in active use at Intel Shannon. Hence, a brief background summary of each of these approaches is provided here.

eXtreme Programming (XP) XP has been pioneered by Kent Beck, and has its origins in a project to develop an internal payroll system at Chrysler in 1996–1997. It is comprehensively described by Beck (2000), where he describes it as 'a light-weight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly-changing requirements'. XP comprises five key values: communication, feedback, simplicity, courage and respect. These are underpinned by 12 key practices, summarised in Table 1. The eXtreme Programming (XP) approach explicitly acknowledges that it is not a magic set of revolutionary new development techniques; rather, it is a set of tried and trusted principles that are well-established as part of the conventional wisdom of software engineering, but which are taken to an extreme level – hence the name eXtreme Programming.

Table 1 Key practices of XP (adapted from (Beck, 2000)

The planning game: A quick determination of the scope of the next software release, based on a combination of business priorities and technical estimates. It is accepted that this plan will probably change.
Small releases: Put a simple system into production quickly, then release new versions on a very short cycle.
Metaphor: Guide all development with a simple shared story of how the whole system works.
Simple design: The system should be designed as simply as possible at any given moment in time.
Testing: Programmers continually write tests, which must be run flawlessly for development to proceed. Customers write function tests to demonstrate the features implemented.
Refactoring: Programmers restructure the system, without removing functionality, to improve non-functional aspects (e.g. duplication of code, simplicity, flexibility).
Pair-programming: All production code is written by two programmers at one machine.
Collective ownership: Anyone can change any code anywhere in the system at any time.
Continuous integration: Integrate and build the system every time a task is completed – this may be many times per day.
40-H week: Work no more than 40 h per week as a rule.
On-site customers: Include an actual user on the team, available full-time to answer questions.
Coding standards: Adherence to coding rules that emphasise communication via program code.

Scrum Scrum (Schwaber & Beedle, 2002) is a simple low overhead process for managing and tracking software development. It attempts to control this 'chaordic' process using a project management framework that involves requirements gathering, design and programming. While it is very much influenced by Boehm's (1988) spiral model, it has its software development origins in a project by Jeff Sutherland at the Easel Corporation in 1993 where it was used in the development of an OO analysis and design tool. It's origins lie outside the field of software development altogether, in Japan in the mid-1980s where an adaptive, quick, self-organising product development process was employed (Abrahamsson *et al.*, 2002). Scrum differs from traditional approaches in that it assumes that analysis, design and development processes are largely unpredictable. At its heart, Scrum comprises a number of stages which, building on its underpinning metaphor of a rugby scrum, also follow a sporting theme.

Sprints are nonlinear and flexible. Where available, explicit process knowledge is used; otherwise tacit knowledge and trial and error is used to build process knowledge. Sprints are used to evolve the final product. The project is open to the environment until the Closure phase. The deliverable can be changed at any time during

the Planning and Sprint phases of the project. The project remains open to environmental complexity, including competitive, time, quality, and financial pressures, throughout these phases (see Table 2).

Despite the claim by its proponents that Scrum has been used on 'thousands of Scrum projects' (Schwaber & Beedle, 2002), there have been quite a few accounts of the use of Scrum in real world projects (Abrahamsson *et al*., 2003).

Method tailoring

Research to date on method tailoring has by and large tended to fall into two streams – contingency factor approaches and method engineering.

Contingency factor research (Iivari, 1989; Avison & Wood-Harper, 1991) on software development methods is typically premised on the notion that specific features of the development context are mapped to the selection of an appropriate development method from a portfolio of methods. However, one of the fundamental problems with the contingency approach in practice is that an organisation is expected to have a range of methods available to developers who, presumed to be fully *au fait* with each method, choose the most appropriate one depending on the contingencies of the situation. Close familiarity with even one method is not all that common in practice among developers; thus, achieving competence with several is not a realistic expectation. Also, the cost of sourcing and training for each method would be prohibitive. Some of the later contingency factors research (Iivari, 1989; Avison & Wood-Harper, 1991) recognised this fundamental flaw, and suggested a more pragmatic view, arguing that contingency be built-in as a feature of the method itself. Thus, rather than suggesting a repertoire of methods, the encompassing framework of the method is expected to cover all situations. However, again a fundamental flaw is the assumption that existing methods adequately cover all contingencies of any given development situation.

Method engineering research (Kumar & Welke, 1992; Harmesen *et al*., 1994) acknowledges the advantage of software development methods in their provision of a disciplined standard for development, but recognises that flexibility is necessary so that methods can be tuned to meet specific project needs. This research argues for a meta-method process whereby methods are precisely constructed from existing discrete pre-defined and pre-tested method fragments.

Among the potential problems with method engineering is the fact that a repository that can store method components is required. The experiences of CASE would suggest that electronic support for such an initiative could be problematic. Also, the meta-method process suggests new software development roles – a method engineer, for example – and this may not be welcome in many organisations.

Table 2 Key practices of Scrum (adapted from (Schwaber and Beedle, 2002)

Pre-game phase
Planning: The definition of a new release of the system based on the currently known backlog of required modifications, along with an estimate of its schedule and cost. If a new system is being developed, this phase consists of both conceptualisation and analysis. If an existing system is being enhanced, this phase consists of limited analysis.
Architecture: This phase includes system architecture modification and high-level design as to how the backlog items will be implemented.
Main game phase
Sprints: This involves development of new release functionality, with constant respect to the variables of time, requirements, quality, cost, and competition. Interaction with these variables defines the end of this phase. There are multiple, iterative development sprints, or cycles, that are used to evolve the system.
Post-game phase
Closure: Here the focus is on preparation for release, including final documentation, pre-release staged testing, and release.

Agile method tailoring

While XP is acknowledged as not being a 'one size fits all' approach suited to every development context (Cockburn, 2001; Stephens & Rosenberg, 2003), this is not at all surprising, as 'the idea of a one best way or set of best practices are holdovers from scientific management, and have no place in software development' (McBreen, 2003). For example, Bowers *et al* (2002) illustrate the need to tailor XP to suit large-scale projects, and support the claims of Drobka *et al* (2004) that the method also needs to be adjusted to suit projects that are deemed mission critical. Stotts *et al* (2003) study the adjustments required to traditional pair programming to cater for a distributed environment. Some studies have advocated an *a la carte* approach such as 'XP Lite', where an existing agile method is 'defanged' (Stephens & Rosenberg, 2003), However, breaking a method apart, and only using certain principles contravenes the claim of synergy that the whole of the method is greater than the sum of its parts, and that the benefits are achieved through the combination of all practices (Jeffries *et al*., 2000; Schwaber & Beedle, 2002). An important aspect of XP is that several of the practices overlap to some extent and thus serve to complement and reinforce each other. Refactoring, simple design, collective ownership and coding standards provide an example of such inter-dependencies (Beck, 1999). As Schwaber & Beedle (2002) put it, 'XP values and their underlying practices and techniques are not divisible and individually selectable; they form a coherent, whole process'.

A marked feature of both the contingency and method engineering research is that they are largely deductive in nature and employ theoretical and conceptual arguments to support how methods should be tailored or con-

structed. Very little is available in terms of practical applications of these ideas in real development practice. Fitzgerald *et al*. (2003) report on a sophisticated approach to method tailoring in Motorola whereby an overarching method was first established within the company, albeit tailored at a macro-level to the general needs of the company. Subsequently, for each individual development project, some precise micro-level tailoring of development practices at a finer level of granularity was conducted so as to suit the actual demands of the particular project context.

Research approach

Background to the case

Intel Shannon is based in the west of Ireland and is part of Intel's Infrastructure Processor Division. The main Intel plant in Ireland near Dublin employs 4200 people. The Intel Shannon organisation employs 125 people, and about 90 are involved in engineering, software development and silicon design. The products under development are network processors for networking equipment, typically for SMEs, the small office/home, and 3G wireless markets. For these products, requirements analysis is typically done in the US, and the software and silicon design is done in Shannon. Intel Shannon has seen significant growth in their workforce over the past few years. The study at Intel Shannon is based on the software development of two product families, the IXP2XX and IXP4XX network processors. During this study, the IXP2XX project involved approximately 15 engineers split into four teams across three sites, and had a development duration of 18 months. The IXP4XX product consisted of five teams and over 30 engineers, across two sites, with a development duration of 24 months.

In terms of software development, Intel Shannon has been formally assessed at Level 2 on the Capability Maturity Model (CMM). While this has led to some discipline in the development process, the rapid time-to-market pressures has led Intel Shannon to consider agile methods. Further, they are a company who embrace innovation and seek to rigorously assess new techniques and methods that could meet their market needs. Intel Shannon have been deploying a range of agile methods over the past 5 years, principally two flavours of agile methods, XP for the technical engineering aspects of software development, and Scrum for the project planning and tracking.

While the move to CMM certification was driven more as a top-down mandate within the organisation, in marked contrast, Scrum and XP were introduced at a grassroots engineering level as optional techniques. As such, their adoption has grown organically over time. They were not compulsory as the techniques were being introduced in parallel with CMM implementation. Indeed, contrary to the conventional wisdom, agile methods and CMM were found to be quite compatible.

Research method

The objective of this research was not simply to examine the use of XP and Scrum in practice, but to gain a better understanding of how these methods were fragmented and tailored, and also how they might be combined to complement each other.

We decided to adopt an *interpretive, exploratory research methodology* for the study. An interpretivist stance is considered appropriate in new and evolving fields such as IS (Walsham, 1995a, b). Within this field, agile methods have only recently attracted the attention of researchers, and very little is currently known about how they are used in practice. Also, little or no research has focused on how agile methods are tailored, or how they may complement each other. Interpretivist research is also considered most appropriate when it is necessary to consider the context of the study and the 'often complicated relationship between people, ideas and institutions' (Travers, 2001). Such a focus is necessary given that agile methods in Intel Shannon were strongly championed by the grassroots software developer community, and agile methods themselves 'value people over processes and tools' (Fowler & Highsmith, 2001).

The *case study* approach (Benbasat *et al*., 1987; Lee, 1989; Yin, 2003) was adopted as it is considered appropriate where the research has a descriptive and exploratory focus (Marshall & Rossman, 1989). Case studies can be very valuable in generating an understanding of reality (Yin, 2003), allowing authentic representation of the situation 'in its own terms' (Hammersley & Gomm, 2000). Thus, an appropriate approach is an in-depth study which a single case provides, what has been termed the 'revelatory case' (Yin, 2003). A single case strategy is also strongly recommended by Mintzberg (1979).

One of the often-cited limitations of the case study method is its lack of generalisability, as the data collected is often specific to the particular situation at a particular point in time. However; Lee & Baskerville (2003) identify a fundamental and long-standing problem with the type of generalisation based on the type of statistical sampling frequently sought in research, namely the problem of attempting to generalise to any other settings beyond the current one. Following this conventional model, researchers have typically suggested increasing sample size or number of case study organisations, but Baskerville and Lee argue cogently for the ultimate futility of this flawed strategy. Furthermore, the 'thick descriptions' (Yin, 2003) provided by the case study were considered much more valuable than generalisability of results. The rich findings generated from this study may then be used to generate hypotheses suitable for testing in a more quantitative fashion.

Data collection involved a series of formal and informal personal interviews, conducted over a 2-year period with the project managers and key staff responsible for agile deployment at Intel Shannon. In all, interviews were conducted with 12 different individuals, with some

interviewed on several occasions. This was facilitated by the fact that the Intel Shannon site is close to the universities where two of the authors work. Also, Intel Shannon has a very close relationship with both universities, attending workshops and speaking at seminars. In turn, access by researchers to the Intel Shannon development teams has been very open. Interviews were generally of 1–2-h duration, and informal interviews were used to clarify and refine issues as they emerged. Interviews were used to encourage the interviewee to relate experiences and attitudes relevant to the research problem (Walker, 1988), A reflexive approach (Silverman, 1998), was deliberately allowed in the interview phase adopted in this study. This facilitates 'controlled opportunism' (Eisenhardt, 1989), and has been identified as important in exploratory research as it facilitates refocusing as the research progresses, in that responses to certain questions can stimulate new awareness and interest in particular issues which may then require additional probing (Trauth & O'Connor, 1991). Thus, we did not use a structured interview guide. We were interested in the usage of agile methods, and the nature of the use or non-use of each of the methods. The individual components of both methods served as a coding structure with which to categorise and analyse interview findings. Also, an e-mail survey of the developers and project managers working across the projects was used to gain extra details on the usage of XP. In the case of Scrum, the post-Scrum workshops for individual projects were also a source of a great deal of information. Defect densities were calculated by matching queries of the change management system with scripts run on the source code to calculate non-commented lines of code. Accuracy to development schedule was calculated by comparing baseline schedule dates with actual release dates.

Although the study did not focus on the comparison of opinions over time, this study adhered to recognised best practices regarding any *longitudinal research*, as the interviews were conducted over a reasonably long period of time, For example, Intel Shannon employees interviewed at an early stage in the study were contacted again to offer any updated or refined information, in order to be consistent with those interviewed later in the process, as suggested by (Silverman, 1998).

The use and tailoring of XP

Intel Shannon has been using XP for 5 years. However, even though they have been committed users of XP, they have been quite pragmatic in choosing only those aspects of XP which they perceived as relevant to the needs of their development context. The XP practices that have been deployed, however, have been carefully monitored and the impact measured. Only six of the 12 XP practices have been implemented. These practices were Pair-Programming, Testing, Refactoring, Simple Design, Coding Standards and Collective Ownership. The unused practices were the Planning Game, Small Releases, Continuous Integration, 4-Hour Week, On-Site Customers and Metaphor. Table 3 summarises this usage pattern, which is discussed in more detail below.

Pair-programming

Pair-Programming is perhaps the best known of the XP practices, with generally positive reports on its usage, although Muller & Tichy (2001) suggest that it decreases overall productivity. While most of the other XP practices have been applied across all of the individual software teams at Intel Shannon, Pair-Programming has been selectively applied. Most teams consist of between two and six software engineers with a wide range of experience. Pair Programming was applied initially by two teams on two components of the software for the IXP2XX network processor. On the later IXP4XX network processor it was again employed by two teams.

Pair-Programming was perceived as having a number of significant advantages at Intel Shannon. Firstly, it was estimated that the required code quality level was achieved earlier. On the IXP2XX project, the pair-programmed components had the lowest defect density in the whole product. The defect densities were a factor of 7 below the component with the highest density. On the IXP4XX project, two of the three Intel Shannon teams used Pair-Programming. One of the teams achieved

Table 3 Usage of XP practices in Intel

XP practices in use
Pair-programming: Widely used at particular stages in the development process. Stages of the development process where pair programming does not work well have been identified. Found to facilitate the collective ownership practice.
Testing: Widely used in unit-test code strategy. Helped developers gain better understanding of required functionality.
Refactoring: Done early as it was found to eliminate bugs. Also facilitated continuous simple design.
Simple design: Design done on whiteboard for each block of code, thus allowing design to emerge in parallel with code implementation.
Collective ownership: Useful as it ensured several project team members could maintain code if any individual too busy. However, confined to single teams and not practiced across teams.
Coding standards: Already a strong feature of development environment.
Unused XP practices
The planning game: This is largely accomplished by Scrum.
Small releases: Not practical as software releases are tied to silicon availability.
Continuous integration: Unused due to the need for external test equipment and the complexity of the software.
40-H week: Not achievable where workers collaborate across time zones (Ireland and US).
On-site customers: Unused as in early conceptual stages of development there are no specific customers.
Metaphor: Unused, although some correspondence between silicon design interfaces and software API functionality.

zero-defect quality. The team with the highest defect density was the team that did not. The three teams all had similar experience profiles with Pair-Programming, developers did not get stuck wondering what to do next. If one person was unsure, the other probably did know. Developers also believed that they learned quite a lot from each other and that they remained more focused on the job at hand, and less likely to go off on a tangent.

The essential nature of Pair-Programming where one person is effectively looking over the other's shoulder meant that minor errors were caught early ,saving considerable debugging time. Also, it was useful for testing and debugging, as a fresh viewpoint could spot obvious flaws that were not obvious to the pair partner. The overall process also ensured that more than one developer gained a deep understanding of the design and code, thus facilitating collective ownership (discussed below). Developers suggested that they had more fun, and found the work more interesting. They also seemed more enthusiastic about their work.

However, there were a number of problematic aspects associated with the use of Pair-Programming also. For example, it was found to be unsuitable for simple or well-understood problems, which could be fixed as quickly as a single developer could type. In a similar vein, when doing lots of small changes, it tended to get frustrating. Some developers found Pair-Programming could break their flow of concentration as they needed to pause to communicate non-obvious ideas to the pair partner. Indeed, some developers expressed the view that it was difficult to reflect and concentrate with someone by their side.

To overcome the limitations described, Intel Shannon have documented a number of lessons which will guide their future use of Pair-Programming:

- Some basic rules of pair working etiquette are required, for example, no keyboard wrestling.
- Consideration needs to be given to neighbours to keep background noise to a minimum.
- Use large fonts.
- Set clear objectives at the start of a programming session.
- Planning and coordination may be necessary to prioritise programming over other activities (e.g. helping other engineers, phone calls, meetings), otherwise both people may not be free simultaneously.
- Pair-Programming was not seen as valuable during sustaining activities on the project when the amount of coding is not as significant.

Testing

Intel Shannon also implemented a test-code development strategy, that is, writing the unit-test code while writing production code. They found this had a number of advantages. It set a direction for the immediate development, namely to get the test case working. It also helped developers get a better understanding of what functionality was required of the software from a client point of view. The unit-tests are also implemented as part of a regression test suite and all component unit tests are run on the code repository nightly. Integration tests are also developed to test the individual components in concert and 'smoke tests' are run daily with external test equipment in the weeks leading up to a release.

Refactoring

Refactoring was another XP technique that was quite widely used at Intel Shannon. They found it worked best when it was done early, as it eliminated a lot of bugs, which would have taken up a lot of debugging time otherwise. Refactoring also became akin to a continuous design activity, which is discussed next.

Simple design

In this case, design was done on a whiteboard before each block of code was written. As a result, the design document emerged on an ongoing basis in parallel with the code implementation. Quite significantly, however, they have not subscribed to the XP concept of the code being the design, as documentation is an integral part of the product deliverable at Intel Shannon. Simplicity increasingly became the guiding principle, and over time, developers stopped trying to second-guess the client code and just implemented the requirements. As already mentioned, this practice was very closely linked to refactoring.

Collective ownership

This practice led to a number of benefits. Firstly, it ensured that several members of the project team knew the code well enough to make changes, and if one person was busy, another person could make the requested change. Also, in the Intel Shannon context, changes in team composition were quite common. In the past, this meant that developers had to choose between bringing any code they wrote with them and continuing to maintain it, or spending time teaching the code to someone else and handing over responsibility. Collective Ownership allowed management more flexibility as it resulted in teams being able to maintain the code base since several of the original members would know it well enough to maintain it.

However, Intel Shannon found that Collective Ownership was only appropriate on a single-team basis. Code ownership across multiple teams was not applied. The software engineering team on the whole product could be as many as 30 engineers and the team felt collective ownership could not scale to this wide a population.

Coding standards

Intel Shannon defined a C-coding standard early in the project and referred to it extensively during coding and code inspections. Coding Standards were already a very strong feature of their development environment prior to the application of XP.

Unused XP practices

The Planning Game was not used as many aspects of planning are covered by the Scrum technique, discussed later. From a business priority perspective a product-marketing team have the responsibility for deciding feature priorities. They are in a separate organisation most of whom are not physically co-located. In future, however, they intend to use some prioritisation aspects of the Planning Game.

The XP practice of Small Releases is not feasible early in the product schedule as software releases are tied to silicon availability. Once silicon is available the team typically delivers minor releases every 4 to 6 weeks and major releases every two quarters.

While Continuous Integration is practiced for each component, given the complexity of the overall software and the need for external test equipment, full system integration is done only in the fortnight leading up to a release.

The 40-H Week was seen as a great aspiration but it was not consistently achievable in the Intel Shannon development context, where the discrepancy in time zones between Europe and the US serves to extend working hours.

On-Site Customers are not available. These projects are tied to the design of silicon and in many cases do not have specific customers during the early conceptual stages. The product marketing group act as a customer proxy, prioritizing features based on potential revenue.

Metaphor was not explicitly used, but at a high level the software components do correspond to the interfaces on the silicon and have common patterns of functions on the APIs.

The use and tailoring of scrum

Scrum has been used for 3 years at Intel Shannon although some of the engineers had used it for almost 5 years in their previous organisations. Scrum has really only been documented in book form since 2002 (Schwaber & Beedle, 2002). Up to then the technique was documented on a number of websites. The Intel team also employed a number of techniques from 'Episodes' (Cunningham, 1995). Table 4 summarises the tailoring of Scrum, and this is discussed further below.

Scrum was initially piloted by one team and its use has grown organically to the extent that it now is used by most of the teams in Intel Shannon. They believe the key reason for this enthusiastic embrace of the technique is down to one of the customisations this initial team made. The daily Scrum meeting took place around a board covered with yellow post-it notes. The team recorded tasks for the 24-h period on post-its. This made Scrum very visible in the organisation, and curiosity from other teams helped the initial spread of the technique. Figure 1 below illustrates a sample meeting record with Post-Its attached.

Team members arrive at the daily meeting with their new Post-Its for the next 24 h. The Post-Its in their named area are the tasks that were committed to at the last meeting. If a task is too big for the next 24 h, they write a subset of it on a new Post-It. During the Scrum meeting the team members move completed tasks into the 'done' area. Moving the Post-Its around helps achieve a shared group visualisation of the tasks and project progress.

Table 4 Usage of Scrum practices in Intel

Pre-game phase
Planning: Planning was simplified, there was no analysis of task inter-dependencies, plans were verified by an external team. Time-boxing was not used and contingency plans were put in place, an item not usually included in Scrum. Two planning stages were used instead of one.
Architecture: Much of this was pre-determined by silicon design.
Main game phase
Sprints: Major tasks were split across sprints instead of being contained within them. Tasks were published online.
Post-game phase
Closure: Modified to include a wrap-up session and 'lessons learned' report.

They have also experimented with other innovative practices. For example, one team member took notes and then published the tasks on a web-page. However, they found this was a significant overhead for that team. They also tried running the meeting with each individual taking notes in a personal notebook, but this reduced the shared group visualisation of the project. Overall they found the shared Post-It board the most useful.

The Post-Its encourage people to prepare more thoroughly in advance for the daily meeting. Continuous preparation happens as developers stick new Post-Its to their PC screens during their work in the interim between daily meetings.

Until recently all teams were geographically co-located so the simple low-tech Post-It technique has worked very well. Interestingly, they now have one distributed team that has commenced using the technique using a shared spreadsheet and networked meeting software. It is too early to report on the results of this project, but early indications are promising, thus indicating that some agile methods may be more applicable to distributed development than has been suggested up to now (McBreen, 2003).

Scrum planning

Intel Shannon has made some modifications to the planning process also. They use two planning stages, one at the start of each sprint and one at the start of the project.

Planning is kept simple. There is no complex Gantt chart with complex inter-dependencies between tasks. The overall plan is a series of sprints (see Figure 2). Internal or external milestones can be lined up with

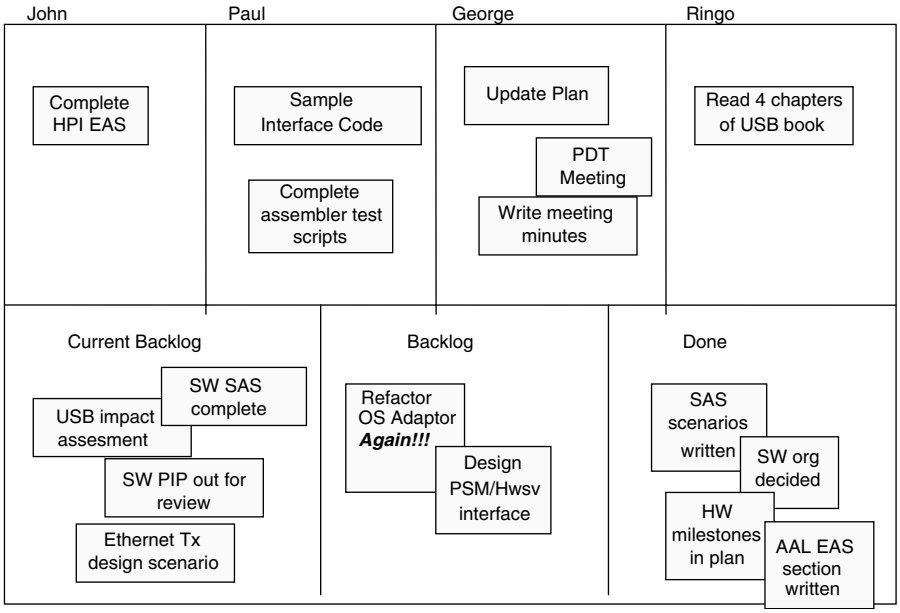

Figure 1 Sample Scrum daily meeting post-it record.

effective number of engineers		2.6	2.6	2.7	2.8	2.8	2.8	
Sprint Tasks	Baseline	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	unassigned
Atmd.HLD.workshops	12.0							12.0
Aal5/Aal0/Aal2.workshops	2.0							2.0
Fpath	64.5							64.5
QMgr	43.5							43.5
Atmd	111.5							111.5
Aal5Acc.FuncSpec.Draft/Review	10.0							10.0
Aal5Acc.DesignSpec	7.0							7.0
Aal5Acc.Code	6.0							6.0
Aal5Codelet.Spec	7.5							7.5
Aal5Codelet.Code	7.0							7.0
Total	271	0	0	0	0	0	0	271.0
Fixed Overhead								
Holidays.Public	4.0							4.0
Holidays.Vacation	45.0							45.0
Training	4.0							4.0
Total	53	0	0	0	0	0	0	53.0
		0	0	0	0	0	0	

Figure 2 Scrum planning.

Sprint completions, but the dependencies between the tasks within the sprint are not worked out in advance.

Each team lead does a plan outlining all of the sprints to the end of the project. Initial meetings are conducted by the engineers to get high-level estimates that can be allocated and distributed across a number of sprints. In one of the projects the wide-band Delphi technique was used to generate the estimates (Linstone & Turoff, 1975). Dependencies between teams are made between end-of-sprint milestones.

In terms of deliverables, the team lead provides a list of sprint milestones and the contents of each sprint to the overall project lead. Intel Shannon do not use sprint time boxing which is part of some implementations of Scrum. The high-level tasks are split to distribute them across sprints. They then continue to distribute and split tasks until the duration of each sprint is at most 20 working days. Contingency is built into the plan and effort estimates are done based on ideal engineering effort. The contingency factor is tuned as the project progresses.

Scrum sprints

At the start of each sprint the team decides which tasks are going to be done in the next sprint. They look at the start of project sprint plan and look at any new backlog items that may have come up during the last sprint. Tasks are allocated to individuals to spread the load. The sprint

protects the team from the environment surrounding it for a meaningful amount of time.

Sprint closure

At the end of the sprint the team lead writes a wrap-up report, listing the tasks completed including extra tasks that that were not part of the original sprint plan. The report will also contain 'lessons learned' and a measurement of the actual effort expended in the sprint *vs* the estimate at the start-of-project. Other end-of-sprint deliverables could include a demo, a project review or a release.

As regards Scrum, project teams have had excellent success delivering projects on time and within budget. An early project of 5.5 months duration with four team members delivered their final release within 3 days of the original plan. The IXP4XX release 1.0 software was delivered 1 week ahead of schedule on a project with an original planned duration of over a year. The team consisted of five teams and over 30 engineers, and all teams used Scrum.

The key advantages of Scrum that the team observed were:

- Planning and tracking become a collaboration involving the whole team
- Excellent communication builds up within the team, thus building morale and helping the team to 'gel'
- The team lead has more bandwidth for technical work
- It enables the team to deliver on-time

The early adoption of Scrum has led to the formulation of internal training courses and in short time the use of Scrum has reached critical mass. Intel Shannon have adapted it very much to their needs with the highly visible daily meeting report. Also, the use of Scrum has led to consistent meeting of development schedules on very complex projects with long project durations, but with no degradation in product quality.

Discussion

The first research objective of this study was to investigate the usage and tailoring of agile methods in practice. A review of previous research literature highlighted two approaches to method tailoring, namely contingency factors and method engineering, and we considered the applicability of both in the context of Intel Shannon.

Contingency factors

The study of Intel Shannon revealed that they did not adopt a strict contingency factor approach (Avison & Wood-Harper, 1991). This would require Intel Shannon to have explicitly investigated the methods available to them, agile or otherwise, and then select an overall method based on their context and environment. The adoption and tailoring of XP and Scrum was proactively driven by a small number of committed champions within the organisation. Rather than explicitly comparing and contrasting available methods against a set of criteria, the methods were adopted in a more pragmatic fashion as discussed above. Nevertheless, while Intel Shannon did not compare and contrast methods before the selection of XP and Scrum, they did consider comparisons after the projects were completed. As a thought experiment, the developers tried to imagine how the software would have turned out if the more traditional development process they were familiar with had been followed. Thus, the comparison of methods against the context of Intel's projects was conducted after the event, rather than before.

Method engineering

The Intel Shannon approach has more in common with the method engineering model (Kumar & Welke, 1992) of a meta-method constructed from existing method fragments. It is also similar to the model in use in Motorola (Fitzgerald *et al*., 2003) in that at a macro-level, the XP and Scrum methods provide an overall framework within which a micro-level tailoring occurs at individual project level. For example, Intel Shannon deconstructed XP, and only used six of the 12 key practices, with project personnel believing that it would be impossible, or at least impractical to implement all practices strictly. This supports McBreen's (2003) view that developers must overcome the naïve view of one best method or set of practices. It must be noted, however, that Intel Shannon did not simply pick the few parts of XP and Scrum that they liked, while ignoring the rest. An interesting distinction between this study and previous research is that Intel Shannon considered all principles before discarding some, as opposed to studies where a small number of principles, for example, pair programming, are examined without adequate consideration to the others. The XP practices that were unused in Intel Shannon were carefully considered and eventually omitted on the grounds that they were not applicable in the development context.

Also, there were some departures from the conventional wisdom in the agile practices that were adopted at Intel Shannon. Quite notably, the agile maxim that the 'code is the documentation' was just not practical in their development context where products are shipped to an external customer who will not be in a position to communicate directly with the development team. Therefore, comprehensive and accurate documentation is critical in these circumstances, a scenario that probably applies more widely than presumed by agile advocates.

The practice of collective ownership was also interesting in that it provided much flexibility in relation to maintenance of code. This was especially useful in the Intel Shannon context where team membership was quite volatile. However, the eventual constraint that collective ownership be confined to a within-team basis is a useful yardstick. Too limited collective ownership would probably result in failure to realise the benefits, whereas too wide a scale of collective ownership could

introduce communication problems in relation to maintenance and configuration management.

As well as selecting and tailoring individual components of the XP and Scrum methods, Intel Shannon displayed an even more advanced use of method tailoring on two other fronts. Firstly, the method tailoring effort was run across both methods simultaneously. For example, the planning game, a key XP practice, was omitted because it was already accomplished by Scrum. This shows that Intel Shannon were not just tailoring one method, but drawing from across a palette of both methods. Secondly, not only did they select parts of methods, they also modified some practices and replaced others with substitute practices. For example, Scrum plans were verified by an external team, contingency plans were put in place, and time-boxing was not used. These examples illustrate how Intel Shannon adapted the Scrum method to suit their organisation and development context.

The maxim that a picture paints a thousand words also has resonances in the Scrum daily meetings with Post-Its on a whiteboard. As well as creating a shared visual experience and a reminder of work to be done each day, this low-tech innovation has had other useful side-effects. Firstly, the existence of the high-profile daily meeting board served as a useful catalyst to grow the use of Scrum at Intel Shannon. Other teams could see the meeting board and witness the daily meeting. The low-tech nature meant it was easy to replicate for other teams who could try it out without any mandate by management. This led to very committed usage of the method over time. However, it also provides the focus for teleconference meetings that allow distributed teams get a visual sense of shared experience.

Another point of relevance to method engineering research is that the conceptualisation of agile method fragments is often over-simplified in research studies. For example, pair programming is not as simple as just having two people code together. Some etiquette and rules of behaviour need to be worked out in advance. There are definitely some scenarios where pair programming does not work well – for example on simple or well-understood problems which could be fixed as quickly as a single developer could type, or when doing lots of small changes as it tended to get frustrating. It was also found to break the flow of concentration if one of the pair needed to pause to communicate non-obvious ideas to the pair partner. There were also some practical impediments to the spread of pair programming at the individual engineer level, as the perception arose that individual ownership of code components could be of more value when employee performance reviews were being conducted. This is a concrete illustration of the fact that software development takes place in a complex socio-political organisational context. Any useful software development practice must also be suitably reinforced rather than undermined by accompanying organisational reward mechanisms.

Synergistic combination of agile practices

The literature on agile methods offers contrasting opinions as to whether XP and Scrum can be fragmented, and whether each fragment may be treated completely independently of the others. Although Intel Shannon's experience certainly indicated an *a la carte* use of XP (Stephens & Rosenberg, 2003), in that they only adopted six of the 12 practices; it must be noted that there were considerable synergistic interdependencies among the practices they did adopt. For example, pair programming facilitated collective ownership although the latter was confined to collective ownership within a project team. Similarly, the use of the refactoring practice reinforced simple design. In general where pair-programming was adopted it tended to lead to a smaller code base, and as defect rate is directly correlated with code length this has led to more efficient use of resources. Thus, synergies can arise through the combination of a subset of the overall agile practices. Therefore, this study concludes that a more nuanced view is necessary which combines elements of both Stephens and Rosenberg's (2003) belief in the *a la carte* approach, and that of Schwaber & Beedle (2002) which states that practices are not divisible and individually selectable. This study shows how subsets of principles are interdependent, but not that every XP or Scrum practice is dependent on all of the others.

Combining agile methods

The second objective of this research study was to examine how agile methods could be combined to complement each other. The literature review highlighted a dearth of knowledge in this area. The Intel Shannon context was an ideal setting to examine this area in more detail, as XP and Scrum were both used on the same project at the same time. The overall finding of the study was that, while XP and Scrum may each be incomplete in their coverage of the overall development process, they are very complementary in that XP provides good support for the more technical and coding aspects of development while Scrum provides a very good framework for project planning and tracking (see Figure 3). Also, both methods are very developer friendly; indeed, the usage of the methods is championed at engineer level and has grown organically from the grassroots engineer level. This is in stark contrast to many organisations, where the use of development methods is mandated by management which often leads to far less actual usage of these methods (Fitzgerald, 1998). The dynamic interaction among small teams at Intel Shannon has helped with morale-building and improved communication among developers, thus facilitating collaboration. Thus, the developer-centric qualities of agile methods lead to considerable benefits at the people and participation level.

It is also generally held in the agile literature that plan-based methods such as those implied by the Capability Maturity Model (CMM) are incommensurate with

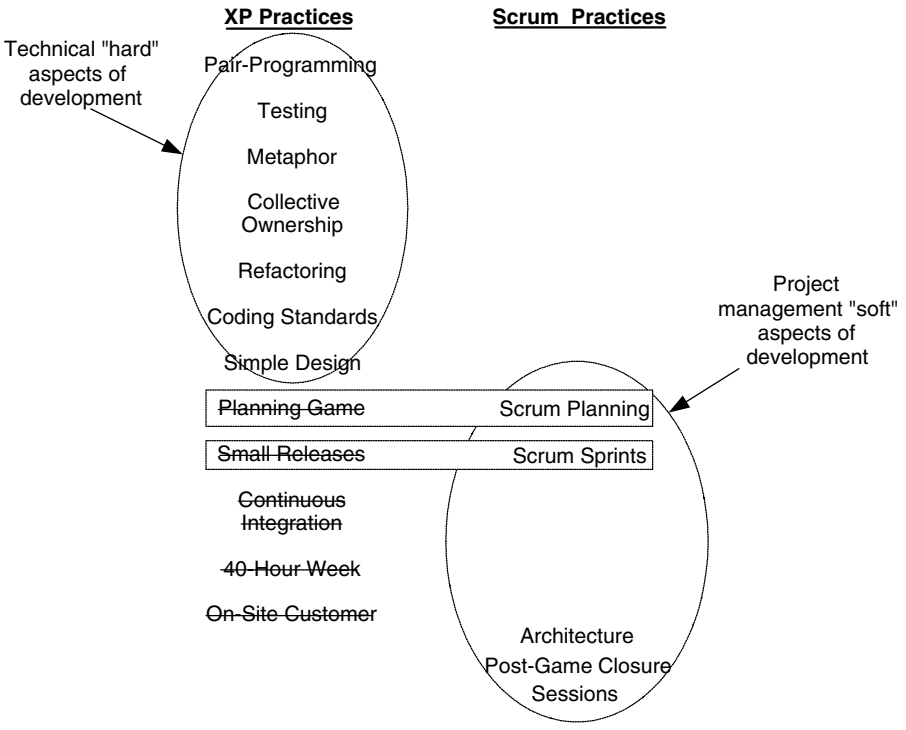

Figure 3 Complementarity of methods in Intel.

an agile approach, although this has been shown conceptually to be an oversimplification (Paulk, 2001). This study provides practical evidence from actual development practice that software process improvement initiatives such as CMM can co-exist with sophisticated and successful agile development. Intel Shannon had already been certified as CMM Level 2, and agile methods were introduced in parallel. The use of agile methods has not caused any perturbation in relation to their CMM certification level.

Another interesting aspect of the study was the relationship between agile methods and distributed development. It has been suggested that agile methods are not applicable for distributed development, primarily due to the fact that they usually require small co-located teams or on-site customers. The deployment of Scrum on a distributed development project at Intel Shannon suggests that Scrum may be more amenable to distributed development than has been assumed up to now.

One of the limitations of this study might appear to be its lack of generalisability, as the data collected is specific to the particular context of Intel Shannon. We have already discussed the misplaced and perhaps misguided attempts to improve generalisability in research (Lee & Baskerville, 2003). Here we sought to get a rich picture of the situation as it applied in one real development context. Indeed, we are heartened by Mintzberg's (1979, p. 583) very apt question: 'what, for example, is wrong with samples of one?'

Summary and conclusions

Overall, there are many lessons from this research at Intel Shannon. The study is useful in being solidly based on the rigorous and disciplined implementation of agile approaches in a real development context involving experienced software engineers, with a careful reflection on subsequent results, rather than a study of student teams in the more artificial context of a university project. The latter would allow more experimental control undoubtedly, but the realism of context that this study provides is arguably an equally important trait. The study also provides many insights into the process whereby Intel Shannon moved from the textbook version of XP and Scrum to the variants they eventually used. It illustrates in detail how both methods were tailored, and shows not just the practices adopted, but just as importantly, the ones that were omitted and why. It also reveals how the methods were fragmented and implemented in a fashion such that they complemented each other. Furthermore, it describes Intel's experience of the synergistic relationship that may arise among a subset of some fragments, although dismissing the idea that a method can only be used if it is used in its entirety.

Although a key contribution of the paper lies in its description of how Intel Shannon tailored, deviated, replaced and combined parts of XP and Scrum, one of the limitations of the study is that it is difficult to assess whether Intel's final method is superior to (i) traditional methods and (ii) XP and Scrum if left untouched. This

limitation is partially overcome through an analysis of a thought experiment Intel Shannon ran after the project was completed, where the team were asked to consider the merits and demerits of the method they used in comparison to more traditional methods. They believed it would have taken in or around the same time, and any discrepancies would be lost in the noise of overhead. However, they felt the traditional code would probably have been quite a bit more complex and long to cater for situations that would probably never occur. As mentioned above, since the defect rate is a constant, this would equate to more bugs. The development statistics reported above illustrate conclusively that agile methods, and more specifically, pragmatically selected fragments of agile methods, can deliver quality software within schedule. Indeed, the country manager for Intel in Ireland has identified Intel Shannon's delivery of extremely high-quality software within schedule as the 'key competitive edge' for Intel Shannon.

In conclusion, it is clear that contemporary agile methods such as XP and Scrum are not anti-method, and require an equally disciplined approach, and as much tailoring as any traditional method. Indeed, we might leave the last word to Tom DeMarco, one of the pioneering figures of the structured approach, whose observations a quarter century ago are still as relevant today:

I find myself more and more exasperated with the great inflexible sets of rules that many companies pour into concrete and sanctify as methodologies. Use the prevailing methodology only as a starting point for tailoring (DeMarco, 1982, 13).

Acknowledgements

This research was supported by the Science Foundation Ireland Grant 02/IN.1/I108 and the European Commission FP6Grant 4337(CALIBRE).

About the authors

Professor Brian Fitzgerald holds the Frederick A Krehbiel II Chair in Innovation in Global Business and Technology at the University of Limerick, Ireland, where he also is a Research Fellow and Science Foundation Ireland Principal Investigator. He holds a Ph.D. from the University of London. Having worked in industry prior to taking up an academic position, he has more than 20 years experience in the software field.

Gerard Hartnett is a software architect in Intel's R&D facility in Shannon Ireland. He managed the global team that developed software/firmware for the IXP4XX product line. He is currently working on new product architectures. He has over 15 years experience in software development with companies like Tellabs, Digital, and Motorola.

Kieran Conboy, prior to joining NUI, Galway, worked for Accenture Consulting across many industrial sectors, such as financial services, communications, and the public sector. These engagements involved organisations across the UK, central Europe, the Nordics and the U.S. Kieran is a member of the Chartered Institute of Management Accountants, and is now completing a Ph.D. at the University of Limerick.

References

ABRAHAMSSON P, SALO O, RONKAINEN J and WARSTA J (2002) *Agile Software Development Methods: Review and Analysis*. VTT Publications, Finland.

ABRAHAMSSON P, WARSTA J, SIPONEN M and RONKAINEN J (2003) New directions on agile methods: a comparative analysis. *Proceedings of 25th ICSE*, Portland, OR.

AMBLER SW (2002) *Agile Modeling: Best Practices for the Unified Process and Extreme Programming*. John Wiley & Sons, New York.

AOYAMA M (1997) Agile software process model. Proceedings of the 21st International computer software and applications conference (COMPSAC 97), 11–15 August *IEEE Computer* 454–459, IEEE Computer Society Press, Washington, DC.

AVISON D and WOOD-HARPER A (1991) Information systems development research: an exploration of ideas in practice. *The Computer Journal* **34,** 98–112.

BECK K (1999) *Extreme Programming Explained*. Addison-Wesley, Reading, MA.

BECK K (2000) *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA.

BENBASAT I, GOLDSTEIN D and MEAD M (1987) The case research strategy in studies of information systems. *MIS Quarterly* **11(3),** 369–386.

BOEHM B (1984) *Models and Metrics for Software Management and Engineering*. IEEE Computer Society Press, NY.

BOEHM B (1988) A spiral model of software development and maintenance. *IEEE Computer* **21,** 61–72.

BOEHM B (2002) Get ready for agile methods, with care. *IEEE Computer* **35(1),** 64–69.

BOWERS J, MAY J, MELANDER E, BAARMAN M and AYOOB A (2002) Tailoring Xp for large mission critical software development. In *XP/Agile Universe* (WELLS D and WILLIAMS L, Eds), Springer, Chicago, IL.

BROOKS F (1987) No silver bullet: essence and accidents of software engineering. *IEEE Computer* **20(4),** 10–19.

CHAE B and SCOTT POOLE M (2005) The surface of emergence in systems development: agency, institutions, and large-scale information systems. *European Journal of Information Systems* **14,** 19–36.

COAD P, DE LUCA J and LEFEBRE E (1999) *Java Modelling in Color*. Prentice Hall, Englewood Cliffs, NJ.

COCKBURN A (2001) *Agile Software Development*. Addison–Wesley, Reading, MA.

CONSTANTINE L (2001) Methodological agility. *Software Development* June 67–69.

CUNNINGHAM W (1995) Episodes: a pattern language of competitive development. In *Pattern Languages of Program Design* (KERTH N, KOPLIEN J and VLISSIDES J, Eds) Addison–Wesley, Reading, MA.

DEMARCO T (1982) *Controlling Software Projects: Management Measurement and Estimation*. Prentice-Hall, Englewood Cliffs, NJ.

DOHERTY N and KING M (2001) An investigation of the factors affecting the successful treatment of organisational issues in systems development projects. *European Journal of Information Systems* **10,** 147–160.

DROBKA J, NOFTZ D and RAGHU R (2004) Piloting Xp on four mission critical projects. *IEEE Software* **21,** 70–75.

EISENHARDT K (1989) Building theory from case study research. *Academy of Management Review* **14,** 532–550.

FITZGERALD B (1994) The systems development dilemma: whether to adopt formalised systems development methodologies or not? In *Proceedings of the Second European Conference on Information Systems* (BAETS W, Ed), Nijenrode University Press, Holland.

FITZGERALD B (1996) Formalised systems development methodologies: a critical perspective. *Information Systems Journal* **6,** 3–23.

FITZGERALD B (1998) An empirical investigation into the adoption of systems development methodologies. *Information and Management* **34,** 317–328.

FITZGERALD B (2000) Systems development methodologies: the problem of tenses. *Information Technology and People* **13,** 13–22.

FITZGERALD B, RUSSO N and O'KANE T (2003) Software development method tailoring at motorola. *Communications of the ACM* **46,** 64–70.

FOWLER M (2000) Put your process on a diet. *Software Development* **8(12),** 32–36.

FOWLER M and HIGHSMITH J (2001) The agile manifesto. *Software Development*, August.

GLASS R (1991) *Software Conflict: Essays on the Art and Science of Software Engineering*. Yourdon Press, Prentice Hall, Englewood Cliffs, NJ.

HAMMERSLEY M and GOMM R (2000) Introduction. In *Case Study Method* (GOMM R, HAMMERSLEY M and FOSTER P, Eds), Sage, London.

HARMESEN F, BRINKKEMPER S and OEI H (1994) Situational method engineering for is project approaches. In *Methods and Associated Tools for the Is Life Cycle* (VERRIJN-STUART A and OLLE T, Eds), North-Holland, Amsterdam.

HEDIN G, BENDIX L and MAGNUSSON B (2003) Introducing software engineering by means of extreme programming. *Proceedings of 25th ICSE*, Portland, OR.

HIDDING G (1996) Method engineering: experiences in practice. In *Method Engineering: Principles of Method Construction and Tool Support* (BRINKKEMPER SKL and WELKE R, Eds), Kluwer, London.

HIGHSMITH J (1999) *Adaptive Software Development*. Dorset House, NY.

HIGHSMITH J and COCKBURN A (2001) Agile software development: the business of innovation. *IEEE Computer* **34(9),** 120–122.

IIVARI J (1989) A methodology for is development as organisational change. In *Systems Development for Human Progress* (KLEIN H and KUMAR K, Eds) North-Holland, Amsterdam.

JACOBSON I (2002) A resounding yes to agile processes – but also to more. *Cutter IT Journal* **15,** 18–24.

JEFFRIES R, ANDERSON A and HENDRICKSON C (2000) *Extreme Programming Installed*. Addison-Wesley, Reading, MA.

KRUCHTEN P (2000) *The Rational Unified Process: An Introduction*. Addison-Wesley-Longman, Boston, MA.

KUMAR K and WELKE RJ (1992) Methodology engineering: a proposal for situation-specific methodology construction. In *Challenges and Strategies for Research in Systems Development* (COTTERMAN W and SENN J, Eds), John Wiley & Sons Ltd, Washington.

LEE A and BASKERVILLE R (2003) Generalising generalisability in information systems research. *Information Systems Research* **14,** 221–243.

LEE A (1989) A scientific methodology for MIS case studies. *MIS Quarterly* **13,** 33–50.

LEE G and XIA W (2005) The ability of information systems development project teams to respond to business and technology changes: a study of flexibility measures. *European Journal of Information Systems* **14,** 75–92.

LINSTONE H and TUROFF M (1975) Introduction. In *The Delphi Method: Techniques and Applications* (LINSTONE H and TUROFF M, Eds), Addison-Wesley, Reading, MA.

LYCETT M and PAUL R (1999) Information systems development: a perspective on the challenge of evolutionary complexity. *European Journal of Information Systems* **8,** 127–135.

MARSHALL C and ROSSMAN G (1989) *Designing Qualitative Research*. Sage Publications, California.

MCBREEN P (2003) *Questioning Extreme Programming*. Addison-Wesley, Boston, MA.

MINTZBERG H (1979) *The Structuring of Organisations*. Prentice-Hall, Englewood Cliffs, NJ.

MULLER M and TICHY W (2001) Extreme programming in a university environment. *Proceedings of 23rd ICSE,* Toronto, Canada.

PAULK M (2001) Extreme programming from a CMM perspective. *IEEE Software* **18(6),** 1–8.

POPPENDIECK M (2001) Lean programming. *Software Development Magazine* **9,** 71–75.

SARKER S and SAHAY S (2004) Implications of space and time for distributed work: an interpretive study of US–Norwegian systems development teams. *European Journal of Information Systems* **13,** 3–20.

SCHWABER K and BEEDLE M (2002) *Agile Software Development with Scrum*. Prentice-Hall, Upper Saddle River, NJ.

SILVERMAN D (1998) Qualitative research: meaning or practice. *Information Systems Journal* **8,** 3–20.

STAPLETON J (1997) *DSDM: Dynamic Systems Development Method*. Addison Wesley, Harlow, England.

STEPHENS M and ROSENBERG D (2003) *Extreme Programming Refactored*, Apress, NY, ISBN 1-59059-096-1.

STOTTS D, WILLIAMS L, NAGAPPAN N, BAHETI P, JEN D and JACKSON A (2003) Virtual teaming: experiments and experiences with distributed pair programming. *Extreme Programming/Agile Universe 2003* (MAURER F and WELLS D, Eds), pp 129–141, Springer.

TRAUTH E and O'CONNOR B (1991) A study of the interaction between information, technology and society. In *Information Systems Research: Contemporary Approaches and Emergent Traditions* (NISSEN H, KLEIN H and HIRSCHHEIM R, Eds), Elsevier, North Holland.

TRAVERS M (2001) *Qualitative Research through Case Studies*. Sage, London.

WALKER R (1988) *Applied Qualitative Research*. Hampshire, Gower.

WALSHAM G (1995a) The emergence of interpretivism in is research. *Information Systems Research* **6,** 376–394.

WALSHAM G (1995b) Interpretive case studies in is research: nature and method. *European Journal of Information Systems* **4,** 74–81.

YIN R (2003) *Case Study Research: Design and Methods*. SAGE Publications, Thousand Oaks, CA.