A Case Study of SME Web Application Development Effectiveness via Agile Methods

Peter Clutterbuck, Terry Rowlands and Owen Seamons
University of Queensland, Brisbane, Australia
p.clutterbuck@business.uq.edu.au
t.rowlands@business.uq.edu.au
o.seamons@business.uq.edu.au

Abstract: The development of Web applications is an important focus of the modern information enabled organization – whether the Web application development is in-house, outsourced, or purchased as '*commercial-off-the-shelf*' (COTS) software. Traditionally Web application development has been delivered via the dominant waterfall system. The waterfall system relies upon well-defined governance structures, linear phases, gating, and extensive reporting and sign-off documentation. An increasing number of development stakeholders criticise the waterfall system for web application development. The criticisms include a disproportionate focus on governance and process at the direct expense of flexibility and, most importantly, reduced productivity. One consequence of these criticisms is the increasing adoption of Web application development via agile-system methods. This agile-system approach centres upon smaller design teams, fewer development phases, and shorter development time tables.

This case study examines the implementation of the agile-system approach as used by a Small-to-Medium Enterprise (SME) software developer. The case study data collection involves interviews and observations across three different SME sources: project managers, Web application programmers, and customers. The case study analysis synthesises the experiences of these managers, programmers and customers to produce an overall assessment of the usefulness of Web application delivery via agile-system methods. The major conclusions from the case study are that a '*default*' agile-system approach may be tailored or fine-tuned to fit an individual developer's software process. This tailoring is based upon the developer's assessment of best practice from the overall agile-system methodology. This tailoring, however, delivers a software development process that exhibits efficiencies and risks. The efficiencies include a more fulfilling role for each development team member, greater richness and continuity in design, a simple management system that delivers key information on a timely basis to all stake-holders, and increased business and technical quality within the delivered application, and a relatively low cost for actioning changes to user requirements. The risks pivot upon experience levels, skills levels, and the quality of interaction within – and between - both the development team and customer organization.

Keywords: project management, information systems management, methodology, agile-system

1. Introduction

Information system (IS) development is a much studied, heavily practised activity. The term '*IS development*' is used in this paper to describe the overall evolutionary life-cycle of an IS. That is, capturing and validating user requirements, estimating feasibility, analysis, design, testing, implementation and maintenance. The term '*method*' is used in this paper to denote an overall strategy that is used to guide and manage an IS life-cycle.

As outlined in (Boehm, 2004), the history of IS development has been characterised by three distinct generations or paradigms of IS development methods – and consequential '*method wars*'. The paradigm comprising structured analysis and design methods initially found its voice in methodologists such as Tom Demarco, Ed Yourdon, Larry Constantine, Harlan Mills, Michael Jackson, and many others. The subsequent era of object-oriented analysis and design methods found its voice in proponents such as Jim Rumbaugh, Ivar Jacobson, Peter Coad, and many others. The current '*post-dot-bomb era*' has seen a new method paradigm emerge – championed by individuals such as Kent Beck, Martin Fowler, Robert Martin, and many others. This new method paradigm is referred as '*agile methods*'.

The current dominant software development paradigm has evolved from the structured analysis and design methods of Demarco et al. and the object-oriented analysis of Rumbaugh et al. This paradigm is referred to as the traditional approach or waterfall model and is characterized as plan-driven and process oriented. (Boehm, 2004) states that the traditional approach is perhaps best exemplified by the Capability Maturity Model for Software (SW-CMM) (Paulk, 1993) and its evolutionary successor the Capability Maturity Model Integration (Version 1.2) described in (Chrissis, 2006). As stated in (Boehm, 2004), "*Thousands of organizations have embraced the SW-CMM and have found that their software development became less chaotic.*"

Reference this paper as:
Clutterbuck, P., Rowlands, T. and Seamons, O. "A Case Study of SME Web Application Development Effectiveness via Agile Methods." *The Electronic Journal Information Systems Evaluation* Volume 12 Issue 1 2009, pp. 13 - 26, available online at www.ejise.com

During the post 2000 era, the environment in which software is conceived, specified, created, and maintained continues to change rapidly and significantly. Software systems continue to grow in size and complexity. Software system delivery is now achieved through a broad mix of in-house, outsourced, and commercial-off-the-shelf (COTS) development strategies. Software is truly ubiquitous – it is routinely found in our business, leisure, and home lives. Software quality, ease of use, and time-to-market are very important to both developers and users. Very significant emphasis has been placed on the demand for flexible development methods (Lee, 2005) that can handle this rapid environmental change and increasing complexity (Lycett, 1999). This view is echoed in (Boehm, 2004) who states "*In the past few years, the mainstream software development community has been challenged by a counter-culture movement that addresses change from a radically different perspective. This new approach, called 'agile' by its proponents, is best exemplified by their Agile Manifesto:*"

We have come to value:

Individuals and interactions over process and tools;

Working software over comprehensive documentation;

Customer collaboration over contract negotiation;

Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.

(Boehm, 2004) describes agile methods as encouraging "programmers to shed their heavyweight process chains, embrace change, and escape into agility. Advocated methods have short cycle times, close customer involvement, and an adaptive rather than predictive mind set." (Highsmith, 1999; Fowler, 2001) describes agile methods as a "just enough" method strategy because agile methods aim to avoid prescribing cumbersome and time-consuming processes that add little value to the software product and actually elongate the development process.

Whilst it is generally accepted that there are some perceived shortcomings with the traditional/plan-driven IS development process (Grisham, 2005), it is very important to all IS stakeholders that agile methods are not prematurely or unjustifiably seen to be the latest '*silver bullet*' solution to existing problems in software development. Whilst there is a growing body of research into IS development via agile methods, it would seem that this work focuses upon large organizations and/or industrial product settings. The quantitative survey described in (Rumpe, 2002) focused mainly on the IS developer. (Macias, 2003) compared developer effort under both the traditional and agile methods. Productivity was investigated in (Wood, 2003), defect management/maintenance described in (Poole, 2001), and the customising of agile methods within the software process of Intel Ireland in (Fitzgerald, 2006).

This paper describes a Small-to-Medium Enterprise (SME) case study focusing upon the efficiency of Web centric information system (IS) development. The case study SME implements a software process via a combination of two agile methods (Scrum and Extreme Programming or XP). Specifically our two research objectives were to investigate the following:

- How the methods were used in practice – with an emphasis placed on any fine-tuning or tailoring of each method
- The efficiencies and risks of agile development as implemented within the context of the case study SME.

This paper unfolds in the following format. Section two firstly provides an overview of currently available agile methods, and then proceeds to describe Extreme Programming (XP) and Scrum in greater detail. Section three discusses the research method underpinning this paper. Section four presents an analysis of the results from this research. Section five concludes the paper.

2. Agile methods

All agile methods are described in (Abrahamsson, 2002) as displaying the following attributes:

- *Incremental development*: small software releases with rapid development cycles.
- *Cooperative development*: close customer and developer interaction.
- *Method simplicity*: easy to learn, modify and document.
- *Adaptive development*: simple and effective change management at any point within the overall software life-cycle.

 ©Academic Conferences Ltd

This section will firstly describe in overview the most commonly encountered range of agile methods. The methods will be introduced in alphabetical order. The section will then treat in more detail the two agile methods underpinning this research (Extreme Programming, or XP, and Scrum).

2.1 Agile methods overview

Adaptive software development: ASD (Highsmith, 2000) promotes a change-oriented strategy to the software development of large, complex systems. The method encourages incremental and iterative development with constant prototyping. (Abrahamsson, 2002) states that "*ASD claims to provide a framework with enough guidance to prevent projects from falling into chaos, but not too much, which could suppress emergence and creativity.*"

Agile modeling: (Ambler, 2002) describes the key points of AM as the agile practices and cultural principles. The AM modeling practices encourage developers to produce sufficiently advanced models to meet design needs and all documentation purposes. The cultural principles promote communication, team structure organization and team work practices.

Crystal family: (Cockburn, 2000 and 2002) describe a framework of related methods that address the variability of the environment and the specific characteristics of projects. The term "*Crystal*" is used as a metaphor to describe the "*color*" and "*hardness*" or "*heaviness*" of each method. The appropriate Crystal method is selected according to development team size and project criticality. Crystal methods share two fundamental values: the appropriate level of effective communication and a high tolerance of change within the project.

Dynamic systems development method: (DSDM Consortium, 1997) and (Stapleton, 1997) describe more of a framework for developing software rather than a particular method. The five phase DSDM life cycle provides for project management activities and risk management. (Abrahamsson, 2002) states that: "*The fundamental idea behind DSDM is that instead of fixing the amount of functionality in a product, and then adjusting time and resources to reach that functionality, it is preferred to fix time and resources, and then adjust the amount of functionality accordingly.*" DSDM is consistently described as the first truly agile software development method.

Feature-driven development: FDD (Palmer, 2002) focuses on simple process, efficient modeling, and short, iterative cycles. (Boehm, 2004) describes how "FDD depends heavily on good people for domain knowledge, design, and development. A central goal is to have the process in the background to support rather than drive the team." FDD does not assign collective ownership of project tasks (including code base) unlike other agile methods including Extreme Programming. (Boehm, 2004) states that the FDD focus on architecture and "getting it right the first time" is very much the "antithesis of XP's collective ownership" and that "this makes FDD strong for more stable systems with predictable evolution, more vulnerable to nonpredictable 'architecture-breaker' changes."

Rational Unified Process: RUP (Kruchten, 1999) works closely with the Unified Modeling Language (UML). Indeed RUP and UML were designed concurrently by Rational Corporation (now a division of IBM). RUP is characterized by a large volume of process guidelines and is therefore often viewed as a plan-driven, "*heavy*" process. RUP does, however, also display many agile philosophies and is therefore better classified as a "*hybrid*" – incorporating ideas from the agile and disciplined/plan-driven paradigms (Boehm, 2004). RUP addresses business workflows and development economic factors that are usually not specifically covered in other methods. (Boehm, 2004) states that "*RUP is currently being extended to address customer economics and return-on-investment considerations.*" RUP is consistently described as better suited to large projects.

2.2 Extreme programming (XP) and scrum

Section 2.1 overviewed the range of agile methods that feature most prominently within the existing software development environment. This section will now describe the two agile methods that have been studied within this case study research. The two methods are Extreme Programming and Scrum

Extreme Programming (XP) is the most widely recognized agile method (Boehm, 2004). XP has been pioneered by Kent Beck and is described in (Beck, 2000) as "*a light-weight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly-changing requirements*". XP originated as a prototypical C3 payroll system development project within the Daimler-Chrysler organization. XP is based on four values and an initial set of twelve practices. The four values are as follows:

- *Communication*: Most project problems occur because of poor communication – therefore XP strongly promotes communication in a positive fashion.
- *Simplicity*: Develop the simplest product that meets the customer's needs.
- *Feedback*: Developers must obtain and value feedback from the customer, from the system, and from each other.
- *Courage*: Be prepared to make hard decisions that support the other principles and practices.

The twelve key practices of XP are shown in Table 1.

Table 1: XP Twelve key practices

Key Practice	Explanation
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	Produce a simple working system quickly, and 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 of time.
Continuous testing (or Test driven development)	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, simplicity and flexibility. Refactoring strongly focuses upon the removal of code duplication.
Pair-programming	All production code is written by two programmers at one machine.
Collective ownership	Any programmer can change any code anywhere in the system at any time.
Continuous Integration	Integrate and build the system every time a task is completed. It is a fundamental requirement to always have an up-to-date working prototype.
Forty hour week	Work no more than 40 hours per week as a rule.
On-site customers	A customer representative (i.e. a subject matter expert) works full time within the development team.
Coding standards	Adherence to coding rules that emphasise communication via program code

Scrum (Schwaber, 1995; Schwaber, 2002) is depicted in Figure 1. Scrum is a simple low overhead process for managing and tracking software development. Scrum has a very clear project management emphasis. Scrum is predicated on the concept that software development is not a cleanly defined process, but a series of '*black boxes*' with complex input/output transformations. The Scrum process begins with the creation of the *Product Backlog* comprising the prioritized product features required by the customer. The next phase of Scrum centres upon a series of 30 day Scrum *Sprints*. During each *Sprint* the Scrum team will complete a working set of features that have been selected (during a Scrum *pre-Sprint* planning session) from the overall *Product Backlog*. Short (e.g. 15 minute) meetings are held by the Scrum tea m on each day of the Scrum *Sprint*. Each daily meeting allows the team to monitor project status and discuss problems and issues. The conclusion of each 30 day *Sprint* involves the software demonstration of the product features that have been completed during that *Sprint*.

 ©Academic Conferences Ltd

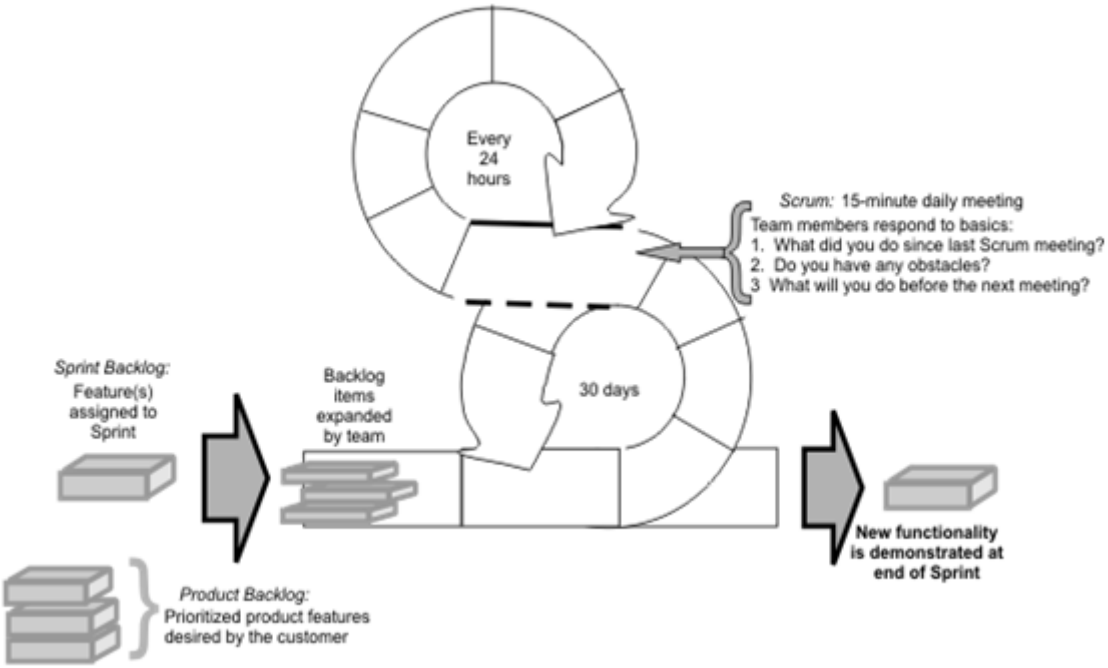

Figure 1: Scrum (Source: ControlChaos.com)

3. Research methodology

The objective of this research was to examine the use of agile methods within a SME software developer, and to gain an understanding of the enabling and limiting factors associated with the usage of these agile methods. The research study has been carried out between November 2007 and March 2008. The SME software developer deploys a project team of seven staff (plus one consulting customer representative) to produce software that is best characterised as web driven client service interfaces to back end database services.

In overview, an interpretative, exploratory case study research methodology was adopted for this investigation. An interpretive methodology is considered appropriate in relatively new and evolving fields such as Information Systems (Walsham, 1995a, b). Within the overall IS area, agile methods have only recently attracted research attention, and little or no research is available as to the efficiencies and risks of agile methods within the SME software development sector. (Travers, 2001) also states that interpretivist research is considered most appropriate when it is necessary to consider the "*often complicated relationship between people, ideas and institutions*". Case study research is comprehensively discussed in (Yin, 2003). (Benbasat, 1987, Yin, 2003, Marshall, 1989) suggest the case study approach is appropriate where the research has a descriptive, exploratory focus. (Yin, 2003) promotes that case studies can be very valuable in generating an understanding of reality, and describes the single, in-depth case study as the "*revelatory case*". (Mintzberg, 1979) strongly recommends a single case study strategy. (Zelkowitz, 1998) describes how case studies in software engineering facilitate the testing of theories and the collection of data in "*an unmodified setting*". This is also very much the view of (Kitchenham, 1995) where case studies are viewed as "*research in the typical*". The case study of this research is exploratory and therefore the results obtained cannot be immediately generalised to any other settings beyond the studied SME and the specific development project. Whilst this result suggests a lack of external validity within this research – it is stressed that the exploratory nature of this investigation aims to generate findings that may subsequently be used to generate hypotheses suitable for testing in a more quantitative fashion.

The data collection within this case study was conducted via qualitative research methods. A series of primary and secondary personal interviews were conducted over the four months of the case study with the SME project manager and several key project stake-holders. Primary interviews averaged two hours in duration. Secondary interviews averaged twenty minutes duration and were used to clarify and refine issues as they emerged. Primary interviews were semi-structured (Patton, 1990) and comprised open-ended questions relating to the use of XP and Scrum within the overall SME software development process. Questioning centred upon a factor listing of all individual components within the '*default*' XP and Scrum processes. Interviewees described how each factor list entry had been implemented within the project and

also assessed the enabling/limiting issues associated with the specific entry. Each interviewee was then requested to assess each factor list entry according to the following set of ordinal values: {*strongly helpful, helpful, improvable, difficult, not-workable*}. Interview transcripts were then coded and analysed using the Glaser-Strauss' *constant comparison* method (Glaser, 1967) to elicit the major efficiency and risk themes. The summarised themes were presented to all research participants at project end to validate the semantic analysis. These summarised themes are presented as research results in the next section.

Concurrent protocol analysis was used to investigate and quantify the cost of requirements change (measured in effort, i.e. person/hour) occurring within the software development cycle up to (but not post) product delivery. Concurrent protocol analysis is an empirical research method for studying the cognitive behaviours and thought processes used by problem solvers (Ericsson, 1993). Concurrent protocols are generated when the problem solver verbalises his/her thoughts while working on a specific task. The verbalisations are recorded during the process and analysed at a later time. Two requirements relate to the validity of concurrent protocol analysis. The first requirement is that the verbalisation of thoughts will not affect the problem solving process. Whilst research continues in relation to this requirement, (Ericsson, 2003) has concluded that concurrent verbalisation does not alter the structure of thought processes. The second requirement is that the problem solving process has a conversational characteristic and therefore lends itself to subsequent semantic analysis by the researcher. This requirement is met in this research by reducing the language/protocol tokens (i.e. the words spoken and recorded) to the following simple language/protocol described in Table 2.

Table 2: Protocol analysis verbal tokens

Date : time started	[**dd:mm and hh:mm**]
Activity Type	[**Design Change** OR **Refactoring** OR **Error Fix**]
Activity Task	[**Analysis**] OR **[Coding]** If coding: **[Class Name:name] [Method Name:name] [Line Number(s):nn]**
Date /time ended	**[dd/mm and hh:mm]**

The verbal tokens described in bold font in Table 2 ensure that the protocol remains very lightweight, non-intrusive upon developer concentration/thought, and easily learned. The developer verbalises the starting *date:time* – then verbalises the current activity type and current activity task. If the developer verbalises **Analysis** then nothing further is required until there is a change of activity – or the session ends. If the developer verbalises **Coding**, then the developer verbalises the **Class Name:name**, **Method Name:name**, and **Line Number(s):nn updated**. A protocol analysis session may iterate through one or more instances of Activity Type and Activity Task. The differentiation of **Analysis** and **Coding** enables separate data capturing for '*thought'* process effort (i.e. Analysis) and coding effort (i.e. Coding). The developer verbalises the end time when the session has concluded. The recorded sessions are then analysed and the relevant data collected.

4. Research results

This section will firstly present project data to normalise this investigation within the overall paradigm of agile method case study research. This normalisation is important because exploratory case study research results cannot be generalised beyond the studied SME and specific project. Normalisation will mitigate this limitation in as much as the results from this research may be compared within an overall context of agile method case study investigation.

This section will then present the major deliverables from this research: the description of how XP and Scrum have been tailored (i.e. fine-tuned) within the SME software developer, and the research assessment of the overall SME software development process.

4.1 Normalisation

Normalisation of this case study is based on the Extreme Programming Evaluation Framework (XP-EF) presented in (Layman, 2006). The XP-EF comprises eight dimensions: *developmental factors*, *sociological factors, project-specific factors, technological factors, ergonomic factors, geographical factors, planning adherence metrics*, and *testing adherence metrics*. This normalisation section will use two of these eight

dimensions: *sociological factors* (Table 3) and *project-specific factors* (Table 4. It is felt that the data treated in the remaining six dimensions will be largely covered by the results presented within the Section 4.2 (*Tailoring and Research Assessment*).

Table 3: Sociological factors

Sociological Context Factor	Value
Team size (number of developers)	7 + 1 tester
Team education level	Bachelors: 7 + 1 tester (customer/business expert) PhD: 1
Team experience level	1 to 5 years: 6 + 1 tester (customer/business expert) 6 to 10+ years: 1
Domain expertise	Medium
Language expertise	Medium to High
Project management expertise	High
Personnel turnover	12.5% (defined as the percentage number of weeks of incoming new staff – relative to the overall project staffing number of weeks)
Morale factors	None (defined within this case study as personnel issues requiring managerial or staff association intervention)

The *sociological factors* in Table 3 show that the project team within this case study were technical competent and led by and experience project manager. The personnel turnover resulted from two people leaving the project (and being replaced immediately) at the eight week mark. Morale throughout project life was very good.

Table 4: Project specific factors

Project-Specific Context Factor	Value
New and Changed User Specifications	18
Domain	Web interface client – database service
Relative complexity	Moderate
Total Component Classes	350
Total Component Methods	482
KLOEC (thousand lines of executable code)	71

The *project-specific factors* in Table 4 show a small size project of moderate complexity. *New and Changed User Specifications* represent the effort expended in capturing user requirements for the software application (i.e. business analysis/requirements engineering).

4.2 Tailoring and research assessment

This section will firstly discuss the tailoring of Scrum within the targeted SME's software development process. The section will then discuss the overall research assessment of the tailored Scrum/XP software development process as measured by the investigative approaches outlined in Section 3.

Tailoring of Scrum and XP within the SME development process is shown in Figure 2.

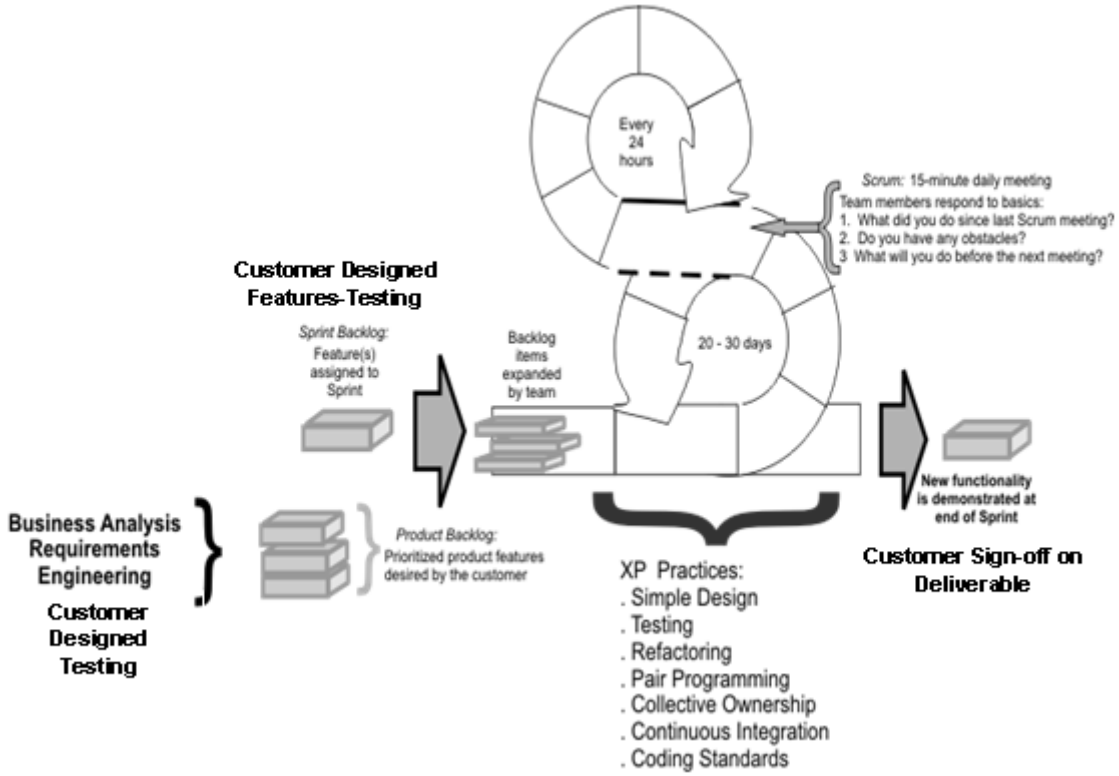

Figure 2: Scrum and XP Practices as implemented within SME

This tailoring described in Figure 2 involves (1) the Scrum *planning* or *front end* stage, (2) the *pre-Sprint* stage, (3) the *Sprint* stage (in which seven of the twelve XP practices have been incorporated), and (4) the Scrum *closure* stage.

Tailoring of the Scrum *planning* or *font end* stage is as follows:

- The addition of a detailed *Business Analysis* of the proposed software application by the project manager and the customer. This analysis aligns the business model, the required business functionality and business processes (both existing and proposed) with the proposed software project automation. It complements the *Requirements Engineering* that is conducted during the Scrum *planning* stage to provide the developer with an overall project management context.
- The addition of *Customer Designed Testing*. This addition is the logical extension by the SME of '*test driven development*'. The test suite is written by the customer (with the assistance of the project manager) and reflects the overall business process functionality of the project deliverable.

Tailoring of the Scrum *pre-Sprint planning* stage is as follows:

- The addition of a *Customer Designed Features Testing*. The test suite is written by the customer (with the assistance of the project manager) and tests the business process functionality of the specific project features that have been selected for the imminent (i.e. next) Scrum *Sprint*. The test suite will be used during the *Sprint* stage. This test suite will be complemented by the conventional *unit-tests* and *integration-tests* that will be developed by the project developers with the aim of gauging code integrity/correctness.

Tailoring of each Scrum *Sprint* stage is as follows:

- The SME has inbuilt further flexibility to each *Sprint* timetable. The *Sprint* duration is planned for 20 days, but can expand to a maximum of 30 days. This timetable flexibility is an explicit risk management control. It is specifically applied to the *Sprint* stage (as contrasted with the overall Scrum cycle) because the SME considers the *Sprint* stage to be the most undefined, complex management component of the overall software development process.
- The *Sprint* stage comprises the following XP practices: *simple design, testing, refactoring, pair programming, collective ownership, continuous integration*, and *coding standards*. The following XP practices have not been incorporated: *planning game, forty hour week, short release cycles*, and *metaphor.* The XP practice *on-site customer* is not included in the Sprint stage. The customer is '*on-site*' as a consultant with the development team during Scrum *planning* stage, the *pre-Sprint* stage, and the

closure stage. The customer does not significantly co-locate with the development team at any stage during the project.

Tailoring of the Scrum *Sprint closure stage* is as follows:

- The *Customer Sign-off on Deliverable* has been inbuilt into each Scrum *Sprint closure* stage. This sign-off applies to unit deliverables, integrated deliverables, and ultimately, the final product deliverable.

The research assessment of the tailored Scrum/XP software development process is now presented. Table 5 shows the tailored Scrum process assessment. Table 6 shows the tailored XP process assessment. Table 7 shows the research assessment of those XP practices that are either partially implemented or not implemented within the SME software development process. Each table is structured as follows:

Table 5: Research assessment of tailored Scrum processes

Stage/Practice	As outlined in Section 3, each interviewee (i.e. all project members, including the customer) was requested to assess each Scrum/XP stage/practice from the following set of ordinal labels: {*strongly helpful, helpful, improvable, difficult, not-workable*}. The Overall Assessment represents the modal value from this data set.
Overall Assessment	
Efficiencies	Interview transcripts with all project members were coded and analysed using the Glaser-Strauss' *constant comparison* method (Glaser, 1967) to elicit the major themes (both positives and risks). Efficiencies describe the major positives identified by the analysis.
Risks	Interview transcripts with all project members were coded and analysed (as outlined above in Efficiencies) facilitating the identification of the major risks associated with a particular stage/practice.

Stage/Practice Overall Assessment	Efficiencies	Risks
Scrum Planning *Strongly Helpful*	Business Analysis (BA) and Customer Designed Testing produce richer and more accurate specifications, with fewer subsequent changes, for customer and developer. BA facilitates a business process redesign approach that more fully captures (for the customer) the potential efficiencies of proposed software automation. Builds stronger trust – at the earliest stage - between customer and developer. Facilitates risk management from the earliest stage.	Customer - project manager disconnection: at social, cultural, geographical or business levels. Project manager deficiency: BA technical competency, social and management skills, software development experience, consultative decision making, continuity for project duration. Customer deficiency: business process knowledge, level of authority within business, consultative decision making, continuity for project duration.
Scrum pre-Sprint Planning *Strongly Helpful*	Alignment of overall project goal with individual milestones. Allows different methodologies for various project estimations. Facilitates risk management and contingency planning within next Sprint (based on the minimal 20 day Sprint duration). *Customer Designed Features Testing* aligns business functionality with software correctness – the right product for customer.	Over-complex planning (e.g. intricate task inter-dependencies) Project personnel difficulties (vacancies, recruitment of experienced developers). Customer deficiency: business process knowledge, level of authority within business, continuity for project duration. Overdesigned customer features.
Scrum Sprint	Assessed in terms of individual XP practices (see Table 5)	
Scrum Sprint Closure *Strongly Helpful*	Confirms quality (i.e. business process accuracy and technical integrity) of each project component as completed. Simplifies integration of the produced components. Better and timelier information for customer.	Changed/changing (i.e. since *Sprint* commencement) business environment or business processes within customer organization.

Table 6: Research assessment of tailored XP practices

Stage/Practice Overall Assessment	Efficiencies	Risks
Pair Programming *Improvable*	Increased quality (lower defect densities) and productivity (earlier reaching of milestones). Increased technical problem solving. Simpler code design. Greater adherence to programming standards. Increased morale within well matched programming pairs.	Not productive for simple coding tasks. Non-compatible programming pairs (on social level, problem-solving/analytical level, experience level). Inappropriate work load allocation to unevenly matched pairs. Difficult to align with induction of new project staff.
Continuous Testing / Test Driven Development *Strongly Helpful*	Alignment of business functionality and software quality – the right product deliverable for the developer and customer. Timelier and more accurate design specifications. Fewer changes to design specifications.	Customer's business process knowledge not aligned with project module currently under design. Difficult to establish the correct number of tests.
Refactoring *Helpful*	Reduced debugging time. Simpler and cleaner software architecture.	Time delay problematic. Complicated by inadequate design specifications. Ineffective automation tools. Incompatibility with some quality control standards.
Simple Design *Improvable*	Delivers necessary and sufficient end product.	Can create an over-reliance on the code being the documentation. Difficult to establish a generalised standard or protocol.
Collective Ownership *Improvable*	Promotes team work – flatter (less hierarchical) project team. Greater knowledge of overall software architecture. Increased adherence to software architecture standards.	Incompatibility with some quality control standards. Does not scale well in terms of project team numbers.
Continuous Integration *Improvable*	Greater knowledge of overall project architecture. Increased capability for integrated testing. Increased flexibility in personnel management of project team.	Does not scale well in terms of project team numbers or project size/complexity. Complicated by complex, over-engineered module interfaces.
Coding Standards *Very Helpful*	Ensures readability of software architecture. Increased quality control automation.	Acclimatisation time for new project members.

Table 7 below shows the '*default*' XP practices that have been either not implemented or partially implemented (*on-site customer*) within the SME software development process.

 ©Academic Conferences Ltd

Table 7: 'Default' XP practices that have been either not implemented or partially implemented.

Stage/Practice Overall Assessment	Reasons for Non-Use
XP Planning Game *Not-workable*	Considered to focus exclusively upon technical issues (code quality, etc) and personnel issues (staff continuity, expertise, etc). Does not focus adequately on business analysis and building trust between developer and customer.
Small Releases *Not-workable*	Very difficult to produce feature-rich, working software in short time cycles. Non-attractive cost/benefit analysis from customer perspective.
Forty hour week *Not-workable*	Considered unworkable by all project members (including customer). Over-regulation of staff, '*blunt-instrument*' approach to personnel management.
Metaphor *Not-workable*	Concept lacks definition for practical application. Considered too-simplistic and one-dimensional for achieving quality outcome for customer and developer.
On-site customer (full-time) *Not-workable*	Non-attractive cost/benefit analysis from customer perspective. Organisational and resourcing difficulties. Emphasis on a significant trust link between project manager and customer – not practical to establish same link between customer and all development team members. Partially implemented: customer/project manager dialogue during the Scrum Planning, pre-Sprint, and closure stages.

The investigation of the overall development method effectiveness within this case study included a strong focus upon quantifying the cost of requirements change that occurs within the development cycle up to (but not post) product delivery. The cost of requirements change would be measured by developer *effort* – which really is a proxy for *person hours*. That is, the research would measure, for each change to the project design, the total number of *person hours* expended in implementing the change. In designing the case study it was also noted that implementing change comprises multidimensional elements. The researches decided to measure two of these elements:

Analysis effort	The '*thought*' effort that is expended in understanding the change and then planning the integration of the change into the existing design.
Coding effort	The effort that is expended in actually expressing the code with correct syntax quality assured via successful unit testing.

The Research Methodology section described how *protocol analysis* – via a very simple *protocol* design – was utilized to capture the required data for analysis. The protocol design provided for the developer actioning the change to flag what activity (i.e. analysis OR coding) he/she was undertaking. Each recorded session was then analysed and data collected.

Figure 3 shows the total cost of implementing the sixteen changes that were required during the project's development life (excluding the maintenance stage) – and the timing of these changes in relation to the five *Sprint* stages comprising the overall project. Figure 3 also shows a typical cost of change curve that is routinely associated with a plan-driven software development. The plan-driven cost of change curve was initially reported by several US corporations in the 1970s. The initial findings reported a consistent 100:1 ratio between a *post-implementation* stage change cost and a *requirements* stage change cost. (Boehm, 1981) found that while the 100:1 figure was generally true for large software development projects, a 5:1 figure was more in tune with the cost of change in small projects (i.e. 2 to 5 KLOC or Thousand Lines Of Code). More recently (McGibbon, 1996) reported a cost of change range from 70:1 up to 125:1. The plan driven cost of change curve in Figure 3 uses a 5:1 figure.

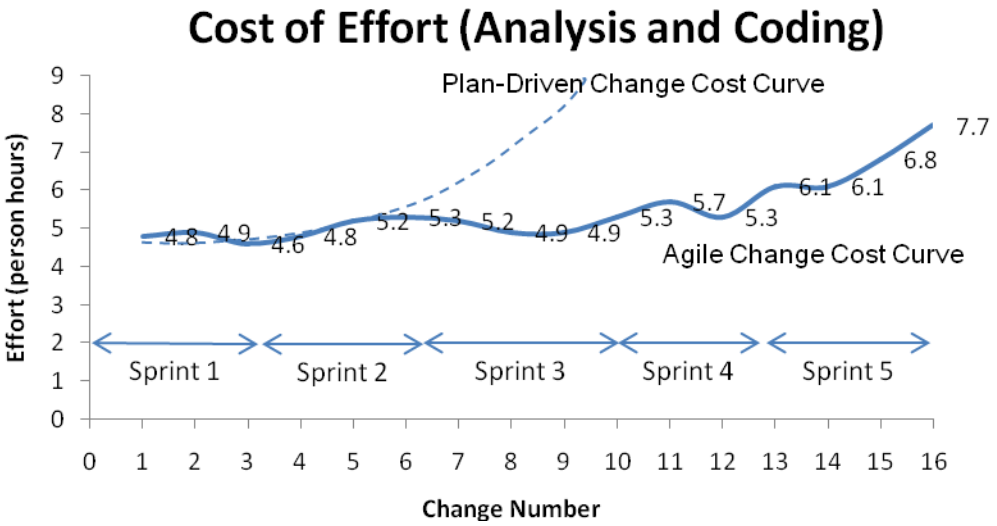

Figure 3: Cost of change curve comparison (agile and plan driven)

As stated earlier in this paper, very little empirical data has been reported within the literature. Indeed (Boehm, 2004) states: "*Although Beck and others have provided anecdotal data on agile change experiences...no empirical data was found for small, agile projects.*" The *anecdotal data* within Boehm's quote refers to the agile cost of change curve presented in Figure 1 of this paper (Introduction).

The agile change of cost data described in Figure 3 initially appears quite impressive. It should be noted, however, that the increase in effort from change 1 (4.8 man hours) to change 16 (7.7 man hours) represents a compound increase of 2.9%.

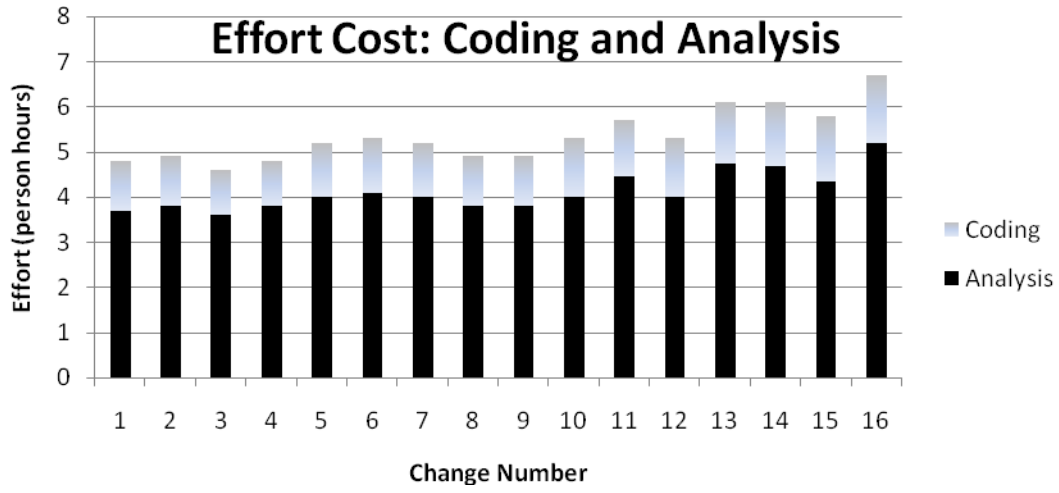

Figure 4: Cost of coding effort and analysis effort

Figure 4 shows the effort costs (again expressed in person hours) for coding and analysis. The compound increase rate for analysis is 2.1% (per change). The compound increase rate for coding is 1.9% (per change). The development team was not surprised by these figures at the post-project review. The development team view was that analysis (meaning the 'thought' that precedes code changes) would consistently be the more time-consuming task.

5. Conclusions

The goals of this case study research were to describe how Scrum and XP practices had been tailored for use within a SME software developer, and to assess the efficiencies and risks of this tailored use. The research results describe those practices that have been '*cherry-picked*' by the SME from the full spectrum of Scrum/XP practices. This is consistent with what is reported in the literature as an emerging trend (Fitzgerald, 2006) with respect to agile methodology usage. The research results also report the major considerations of the SME project team and customer as to why specific practices have been selected in or

 ©Academic Conferences Ltd

out. These considerations confirm that business software development within the SME sector is a complex mix of people, technology, and business processes that at best can be described in a highly abstracted format. The results also reveal some interesting data trends in relation to the cost of requirements change within the development cycle. Change does cost even when using agile methodologies. The designer of XP stated in (Beck, 1999): "*If a flattened cost curve makes XP possible, a steep change cost curve makes XP impossible*". The results in this case study show that change did increase as a function of development cycle time. Consequently change – and its cost – must be carefully risk managed during the project life. Agile development methodologies within the SME business software process contribute many efficiencies – whilst still leaving significant risks for control.

References

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. (2002) "Agile software development methods: Review and Analysis", [online] Technical Research Centre of Finland, VTT Publications 478, http://www.inf.vtt.fi/pdf/publications/2002/P478.pdf.

Agile Alliance. (2001) "Manifesto for Agile Software Development", [online], http://www.agilealliance.org.

Ambler, S. (2002) *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process,* John Wiley & Sons, New York.

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., Mead, M. (1987) "The case research strategy in studies of information systems". MIS Quarterly, Vol 11, No. 3, pp 369-386.

Boehm, B. (1981) *Software Engineering Economics*, Prentice-Hall PTR.

Boehm, B., Turner, R. (2004) *Balancing Agility and Discipline*, Pearson Education, Inc.

Chrissis, M. B., Konrad, M., Shrum, S. (2006) *CMMI: Guidelines for Process Integration and Product Improvement (2nd Edition)*, Addison-Wesley.

Cockburn, A. (2000) *Writing Effective Use Cases: The Crystal Collection for Software Professionals*, Addison-Wesley, Boston.

Cockburn, A. (2002) *Agile Software Development,* Addison-Wesley, Boston.

DSDM Consortium. (1997) *Dynamic Systems Development Method, Version 3*. DSDM Consortium, Ashford, England.

Ericsson, K. A. and Simon, H. A. (1993) *Protocol Analysis Verbal Reports as Data*, The MIT Press, Cambridge, Massachusetts.

Fitzgerald, B., Hartnett, G., Conboy, K. (2006) *Customising agile methods to software practices at Intel Shannon*, European Journal of Information Systems. Vol. 15. pp 200-213.

Fowler, M., Highsmith, J. (2001) "The agile manifesto", Software Development. August.

Glaser, B.G., Strauss, A.L. (1967) *The Discovery of grounded theory: strategies for qualitative research*. Aldine Publishing Company, Chicago.

Grisham, P. S., Perry, D. E., (2005) "Customer Relationships and Extreme Programming", Human and Social Factors of Software Engineering (HSSE). ACM, May, pp 1-6.

Highsmith, J. (1999) *Adaptive Software Development*, Dorset House Publishing, New York.

Highsmith, J. (2000) *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems,* Dorset House Publishing, New York.

Kitchenham, B., Pickard, L., Pfleeger, S.L. (1995) "Case studies for method and tool evaluation", IEEE Software, Vol. 12, pp 52-62.

Kruchten, P. (1999) *The Rational Unified Process,* 2nd Edition. Addison-Wesley, Reading, MA.

Layman, L., Williams, L., Cunningham, L. (2006) "Motivations and measurements in an agile case study". Journal of Systems Architecture (Elsevier), Vol. 52, pp 654-667.

Lee, G., 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, Vol. 14, pp 75-92.

Lycett, M., Paul, R. (1999) "Information systems development: a perceptive on the challenge of evolutionary complexity", European Journal of Information Systems, Vol. 8, pp127-135.

McGibbon T (1996) *Software Reliability Data Summary*, Data Analysis Center for Software Technical Report.

Macias, F., Holcombe, M., Gheorghe, M. (2003) "A Formal Experiment Comparing Extreme Programming with Traditional Software Construction". Proceedings of the Fourth Mexican International Conference on Computer Science (ENC 2003), September 8-12. IEEE, pp 73-80.

Marshall, C., Rossman, G. (1989) *Designing Qualitative Research,*. Sage Publications, California.

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

Palmer, S., Felsing, J. (2002) *A Practical Guide to Feature-Driven Development*, Prentice Hall, Upper Saddle River, NJ.

Patton, M.Q. (1990) Qualitative evaluation and research methods (2nd Edition), Sage Publications, CA.

Paulk, M. C. (1993) Capability Maturity Model for Software, Version 1.1, CMU/SEI-93-TR-24, ADA263403. Pittsburgh: Software Engineering Institute, Carnegie-Mellon University.

Poole, C., Huisman, J.W. (2001) "Using Extreme Programming in a Maintenance Environment" IEEE Software, Vol. 18, No. 6, Nov/Dec. pp 42-50.

Rumpe, B., Schroder, A. (2002) "Quantitative Survey on Extreme Programming Projects" Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), May 26-30, pp 95-100.

Schwaber, K., (1995) “Scrum Development Process”, OOPSLA’95. Workshop on Business Object Design and Implementation. Springer-Verlag.
Schwaber, K., Beedle, M. (2002) *Agile Software Development with Scrum*, Prentice-Hall, Upper Saddle River, NJ.
Stapleton, J. (1997) *DSDM, Dynamic Systems Development Method: The Method in Practice*, Addison-Wesley, Reading, MA.
Travers, M. (2001) *Qualitative Research through Case Studies*, Sage Publications, London.
Walsham, G. (1995a). “The emergence of interpretivism in IS research”. Information Systems Research, Vol. 6, p376-394.
Walsham, G. (1995b) “Interpretive case studies in research: nature and method”, European Journal of Information Systems, Vol. 4, pp 74-81.
Wood, W., Kleb, W. (2003) “Exploring XP for Scientific Research”, IEEE Software, Vol. 20, No. 3, (May/June), pp 30-36.
Yin, R. (2003) *Case Study Research: Design and Methods,* Sage Publications, CA.
Zelkowitz, M.V., Wallace, D.R. (1998) “Experimental models for validating technology”, IEEE Computer, Vol. 31, pp 23-31.

 ©Academic Conferences Ltd