A case-study based assessment of Agile software development

by

William Herman Morkel Theunissen

Submitted in partial fulfillment of the requirements for the degree

Magister Scientia (Computer Science)

in the Faculty of Engineering, Built Environment and Information Technology

University of Pretoria

November 2003

A case-study based assessment of Agile software development

by

William Herman Morkel Theunissen

Abstract

This study set out to determine various aspects of the *agile* approaches to software development. These included an investigation into the principles and practices driving these methodologies; determining the applicability of these approaches to the current software development needs; determining whether these methodologies can comply with software engineering standards (as set out for example by ISO); investigating the feasibility of these approaches for the telecommunication industry; establishing whether practitioners are reaping the benefits that are advertised by agile proponents; and attempting to discover short-comings of the agile paradigm.

This dissertation examines the aforementioned issues and tries to provide answers to them. It is argued that:

Agile software development is suited to projects where the system evolves over the life cycle of the project. These methodologies are intended to seamlessly handle changing requirements. Thus, using an agile approach might provide a competitive advantage in developing e-business solutions which are tightly coupled with the business strategy and needs.

It is shown that agile methodologies can comply with software engineering standards such as ISO 12207:1995 and ISO 15288:2002. Furthermore diligent application of certain agile methodologies may result in a level 3 Capability Maturity Model (CMM) grading.

Evidence from the feedback of a case study conducted on an XP project team, supports the view that XP, and agile in general, does indeed live up to its 'promises'. However, some potential problem areas were identified that should be kept in mind when implementing these methodologies.

Finally, an *in situ* investigation suggests that there are a number of projects in the telecommunication industry that will benefit from the agile approach and its practices.

Keywords:
agile software development, extreme programming, crystal, feature driven development, case study, standards, telecommunication

Supervisor: Prof. DG Kourie
Department of Computer Science
Degree: Magister Scientia

A case-study based assessment of Agile software development

deur

William Herman Morkel Theunissen

Opsomming

Hierdie studie het onderneem om verskeie aspekte van die ‘*agile*’ benadering tot programmatuur ontwikkeling te bepaal. Die studie het onder andere ondersoek ingestel na die beginsels en praktyke wat hierdie metodologieë aanspoor; bepaal of hierdie benadering toepaslik is tot die huidige programmatuur ontwikkelings behoeftes; bepaal of hierdie metodologieë aan programmatuur-ingenieurswese standaarde (soos byvoorbeeld ISO s’n) voldoen; ondersoek ingestel oor die lewensvatbaarheid van die benadering tot die telekommunikasie industrie; vasgestel of die aanwenders van die metodologieë die geadverteerde voordele soos verkondig deur die ‘agile’ voorstaanders geniet; en enige tekortkominge van die ‘agile’ paradigma probeer ontsluit.

Die verhandeling ondersoek die bogenoemde aspekte en probeer antwoorde daarvoor vind. Daar word beredeneer dat:

‘Agile’ programmatuur ontwikkeling is gepas vir projekte waar die stelsel ontvou oor die lewenssiklus van die projek. Hierdie metodologieë is bedoel om sorgloos veranderende spesifikasies te hanteer. Dus, deur van ’n ‘agile’-benadering gebruik te maak, kan ’n kompeterende voorsprong in die ontwikkeling van e-besigheid oplossings wat sterk gekoppel is met die besigheidstrategie en -behoeftes, verskaf word.

Daar word getoon dat ‘agile’ metodologieë aan programmatuur-ingenieurswese standaarde soos ISO 12207:1995 en ISO 15288:2002 kan voldoen. Verder deur nougesette toepassing van sekere ‘agile’ metodologieë, kan ’n vlak 3-‘Capability Maturity Model (CMM)’ gradering bereik.

Bewyse van die terugvoering vanaf ’n gevallestudie wat op ’n XP-projek span onderneem is, ondersteun die standpunt dat XP, en ‘agile’ in die algemeen, inderdaad sy beloftes gestand doen. Desnieteenstaande is daar enkele potensiële probleemareas geïdentifiseer, wat in gedagte gehou moet word by die implementering van hierdie metodologieë.

Laastens, het die ter plaatse ondersoek gesuggereer dat daar ’n aantal projekte in die telekommunikasie industrie bestaan, wat sal baat by die ‘agile’ benadering en praktyke.

Sleutelwoorde:
agile sagteware ontwikkeling, extreme programming, crystal, feature driven development, gevallestudie, standaarde, telekommunikasie

Studieleier: Prof. DG Kourie
Department Rekenaarwetenskap
Graad: Magister Scientia

Acknowledgements

My sincere thanks to:

- the Almighty, for providing me the opportunity to do this work;
- my father, mother, sister, Ian and grandparents for their continuous support throughout my studies;
- Prof. D.G. Kourie for his extraordinary mentoring and valuable input;
- Telkom SA Ltd's Centre of Excellence program and members;
- the Telkom employees that answered all my queries and provided input;
- the Equinox development team, for demonstrating Extreme Programming in practice and their participation in the case study;
- the CIRG, Polelo and Ceteis research groups at the University of Pretoria, for their time and assistance.

Acronyms

AM	Agile Modeling
ASD	Agile Software Development
BBF	Build By Feature
BUD	Big Upfront Design
C3	Chrysler Comprehensive Compensation
CM	Configuration Management
DBA	Database Administrator
DBF	Design By Feature
DSDM	Dynamic Systems Development Methodology
EIA	Electronic Industries Association
FDD	Feature Driven Development
HCI	Human Computer Interaction
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
KISS	Keep It Simple Stupid
LISP	Linked Investment Service Provider
QA	Quality Assurance
RAD	Rapid Application Development
ROI	Return On Investment
RUP	Rational Unified Process
SABS	South African Bureau of Standards
SEI	Software Engineering Institute
SW-CMM	Software Capability Maturity Model

TQM	Total Quality Management
TDL	Telkom Development Lab
XP	Extreme Programming
YAGNI	You Aren’t Gonna Need It

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The predominant multi-year-multi-million-dollar nature of software development projects associated with projects from the 1960's to 1990's seem to have declined over the last decade. Since the inception of the Internet, the life-cycle time-frames for an increasing number of software development projects have shrunk to a mere few month and/or even weeks and days. Ever changing technologies, higher levels of competition, global economies (also known as the 'Global Village'), development approaches such as object-orientation and components have caused the software development industry to once again rethink its strategy. An increased number of today's business models and processes do not stay stable enough to justify supporting software development efforts that can only deliver in two to five years.

The waterfall approach to developing software no longer applies to today's projects. This fact has been debated and accepted by academia and practitioners alike.

The replacement methodologies are usually based on some form of iterative development. Examples are Hewlett-Packard's Fusion [Fusion Summary, 1998], Boehm's Spiral [Boehm, 1988], Rational's Rational Unified Process (RUP) [Kruchten, 2000]. These methodologies addressed the need for faster development. However, they still lack, to a high degree, the ability to deliver working/usable software in the limited time-frames of days and/or weeks.

As we enter the beginning of the 21st century, the words of Tim Berners-Lee "What is a Web year now, about three months?" [O'Reilly Interview, 1997], seem to be the order of the day. *Internet time* has shifted the software development industry into an even higher gear, requiring even faster development cycles than ever before. In these cases software has to 'go-live' so fast that 'traditional' approaches for facilitating the development effort have become a bottleneck for development teams.

Another symptom of these Internet-speed type of projects is the tendency of system requirements changing throughout the project's life cycle. This may be attributed to different factors, including:

- The customer only having a vague idea of what he/she wants from the system at inception;
- Competition by competitors causing changes to business models and processes;

- The increased importance of aligning software systems (especially e-business systems) with the business goals.

In the light of these insufficiencies, methodologists have developed and designed new methodologies to address the problems experienced by software developers with regard to the aforementioned trends. This resulted in a paradigm shift in the nature of developing software for Internet-speed projects. These new methodologies became known as 'light' methodologies due to the fact that a large portion of artifacts traditionally associated with software development were discarded. In 2001 a group of these methodologists came together to discuss their different approaches and to determine if common ground existed between these approaches. This meeting resulted in a mutual understanding between the attendees, the formation of the Agile Alliance and the signing of a manifesto. Chapter 2 provides a more detailed discussion on this meeting and its repercussions. Since then, 'light' methodologies have been re-dubbed as *agile* methodologies.

The agile movement was spawned by practitioners in industry. This leads to the question: how does the academic community interpret these new methodology trends? In the search for an answer to the previously stated question, one struggles to find comprehensive academic literature on the agile movement as a whole. Most of the papers lack global perspectives, preferring to focus on specific portions of agile software development. However, Abrahamsson et al. [2002] provided one of the first review and analysis publications to span over most of the major agile methodologies. During 2002 and 2003, the interest in agile software development by the academic community has grown tremendously. This growth may be observed from the increased number of publications in journals such as IEEE Software, IEEE Computer, Cutter IT and Software Developer Magazine to name a few.

1.2 Research Objectives

This dissertation will attempt to address the following matters:

- the need for yet another class of methodologies
- the main principles underlying agile software development
- the ability of agile methodologies to comply with standards
- an investigation of how agile software development might work in practice

These matters are considered in slightly more detail below.

1.2.1 The need for yet another class of methodologies to light

Several of the contributing factors have already been touched on in the preceding paragraphs. Various other influences will now be mentioned.

In the past 55 years (taking EDSAC[1] as a starting point) of writing programs for computers, the notion of the correct process for coding systems has changed over and

[1]Electronic Delay Storage Automatic Calculator - Developed by Maurice Wilkes from the University of Cambridge. It was the first stored program computer. The computer was based on ideas for the USA Electronic Discrete Variable Automatic Computer (EDVAC) that was the replacement for the Electronic Numerical Integrator and Computer (ENIAC). The concept of not only storing data in a computer but also the program that is run was developed by John W. Mauchly and J. Presper Eckert, the heads of the ENIAC team [Gough-Jones et al., 1993; Wilkes, 1985].

over again, never really stabilising. Since those early days we have seen the process used by the OS 360 team, a multi-year, multi-million dollar project with over a thousand developers, consuming more than five thousand man-years [Brooks, 1995]. From this, Brooks – the project manager – developed the concept of a *surgical-team* (see Section 6.3.1) approach to developing software (see [Brooks, 1995]). Then the years of the *waterfall* model ensued. Some organisations are still clinging to this approach or some derivative. In the wake of the waterfall model came the *spiral* approach [Boehm, 1988], born from the realisation of the shortcomings associated with the waterfall model. With the advent of object-orientation the realisation of its incompatibility with the waterfall model caused the search for yet another new approach to software development. This search resulted in the iterative and evolutionary methodologies. As mentioned previously, the age of interconnectivity that resulted from the Internet has brought to light new challenges that software development methodologies need to address.

One such challenge is the fact that speed-to-market is becoming a primary driving factor in software development. Research by Baskerville *et al.* addresses the question of "Is Internet-speed software development different?" (See [Baskerville et al., 2003]). Their conclusion is that speed is "paramount" with 'Internet software development' and that the cost and quality is of less importance.

Another challenge is the increased notion of satisfying the changing needs of the customer over adhering to the predefined plan and scope. See Highsmith and Cockburn [2001].

The foregoing discussion briefly summarizes why the need for agile software development has arisen. It adresses the first objective of this study and will not be specificaly taken up in the remainder of this dissertation.

1.2.2 Other Objectives

The remaining objectives to be discussed in this dissertation are as follows.

Firstly, *what is the mainprinciples and ideas behind the agile initiative?* In order to analyse agile software development one needs to gain an understanding of its origins and beliefs. The history and background of the agile concept will be explored in Chapter 2. Some examples of agile methodologies are explored in Chapters 3, 4 and 5. These chapters will form the bases of the theoretical understanding of agile software development.

Secondly, with today's trends of outsourcing and the magnitude of unknown software vendors in the market, the need for standards and compliance to them has increased. Software acquirers are starting to make use of accredited suppliers when buying software products. *Compliance of agile approaches to international standards* – in particular ISO standards – is investigated in Chapter 6.

Finally, the growth in the hype associated with the agile approaches brings the question of "*How does agile work in practice?*" to mind. To address this question, the implementation and feasibility of agile methodologies in an XP environment (Chapter 7) and a non-agile environment (Chapter 8) are examined.

The *in situ* case study on XP (Chapter 7) was conducted on a development team with a long standing connection to the University of Pretoria. These developers were approached because of their high reputation and their pursuit to gain an in-depth practical understanding of XP.

Through the second case study (Chapter 8), the feasibility of implementing agile practices in a non-agile development environment was investigated. The feasibility study was conducted by means of an analysis of a unit within Telkom SA Ltd. the current representative software development environment.

The aforementioned questions and the answers gained from this study is summarised and presented in Chapter 9.

Chapter 2

Agile Software Development

2.1 Introduction

For three days, starting on February 11th 2001, seventeen people got together. The location was The Lodge at Snowbird ski resort in the Wasatch mountains of Utah [Agile History URL]. These people included the authors and advocates of Extreme Programming (XP), Crystal, Feature Driven Development (FDD), SCRUM, Adaptive Software Development, Dynamic Systems Development Methodology (DSDM) and Pragmatic Programming. All the people attending this meeting had perceived a need for new approaches to software development. These new approaches rejected the traditional document-driven, so-called heavy methodologies paradigm. Due to the opposite nature of the new methodologies, these were called 'light-weight' methodologies. Through the meeting the attendees hoped to discover common ground between these diverse 'light-weight' methodologies. The result of the meeting allegedly exceeded the attendees expectations; not only was common ground found ; it was considered extensive enough to warrant the formation of an alliance. The common ground was expressed in a manifesto that was signed by the attendees. The concern by some of the attendees that the use of 'light-weight' to classify their methodologies may lead to a misunderstanding, was resolved by adopting the classification name of *agile*[1].

Section 2.2 provides a discussion on the manifesto mentioned above. This manifesto outlines the beliefs and principles that form the common ground of agile methodologies. An extract from the manifesto, as given in Fowler and Highsmith [2001], is provided as Appendix A.

The aforementioned alliance is explored in Section 2.3. Several of the methodologies that are classified as agile is briefly introduced in Section 2.4.

The concept of agile modeling is addressed in Section 2.5.

2.2 The Manifesto on Agile Software Development

Appendix A contains an extract from the *Manifesto on Agile Software Development* that forms the basis for the mutual ground whereupon all agile methodologies are based. The manifesto outlines four values and twelve principles that describe the philosophy behind agile software development.

[1]The concise Oxford dictionary defines agile as "Quick-moving, nimble, active." Fowler and Fowler [1964]

These values are now described in turn.

2.2.1 Individuals and interactions over processes and tools

This value highlights the importance of the human factor in software development. By having a paradigm shift in the way methodologists and managers view developers and stakeholders, the productivity and quality of software products can be maximised. Thus people should be enabled to bring out their best performance without restricting them through the processes and tools they are required to use. This fact had previously been discussed by DeMarco and Lister in the late eighties in their book *Peopleware*. (See DeMarco and Lister [1987]) Practices that are found in some of the agile methodologies that are based on this value, include *pair-programming*[2]; *on-site customer* and co-located small teams. Agile methodologies also promote face-to-face interaction to maximise communication and information flow between stakeholders.

2.2.2 Working software over comprehensive documentation

Due to the similar nature of this value and the ***Responding to change over following a plan*** value, they will both be addressed together. Any software development effort should focus on the primary goal – the resultant software product. Agile developers believe that some methodologies and/or their implementations have lost their focus on the primary goal of software development. These implementations of methodologies have started to stress the delivery of artifacts that describe the way the process is implemented. Requiring developers to write comprehensive reports on how each phase of the process was conducted and the findings thereof, is regarded a time consuming and an inefficient allocation of human resources. This practice thus increases the cost associated with development. In business it is considered a fundamental rule that *time is money* and paying for highly intelligent and proficient people's services is very expensive. Thus as can be easily deduced from the previous statements, the more time used for software development the more expensive the project and deliverable will be. This expense needs to be justified by the return-on-investment (ROI) of the resultant artifacts that are delivered through the project. It is a common occurrence in software development that the documentation describing the software being developed quickly loses synchronisation with the actual working software.

As experience in industry has shown, defining exact models of the planned software is often impossible, and the practical implementation sometimes brings to light situations that could not be foreseen. This induces clever adjustment to the implementation of models to enable the software to work. Such adjustments are then only seen in the code and not in the original documentation.

Changing requirements is a common occurrence in many software development projects. This occurrence can be attributed to a diverse range of reasons. Those that tend to dominate are: firstly, a customer's inability to really articulate their needs at the start of a project; secondly, the new breed of projects, namely e-business solutions, that need to be developed in *internet time*[3]. Accepting and incorporating changes into software is required to ensure that the customer needs are met and that the software delivered provides the maximum value to the users. By denying changes the development

[2]A technique where two programmers write code together using one computer.

[3]A term to describe the very fast time periods associated with the development and lifespan of Internet-related software.

team faces the possibility of alienating the customer and, in turn, causing the failure and breakdown of the project.

The agile advocates believe that devising a detailed plan and forcing adherence to this plan, discourages the 'embracing of change' and encourages the idea of planning for tomorrow's *possible* needs. Planning for possible eventualities is believed to increase a solution's complexity. Some of these possible eventualities may never even arise and their solution is thus never used. Once again slavishly following a plan may increase the cost associated with the project, even with agile methodologies. Creating a detailed plan utilises a lot of effort, thus incurring cost. Making changes to this plan later on will incur even more cost. Requiring changes to detailed plans translates into adjusting to unforeseen circumstances. The original plan was therefore incorrect and, in turn, also the budget. Once again cost piles up on top of cost.

As stated previously, the ROI of artifacts needs to be determined and used to guide the appropriateness of these artifacts. The guideline may then be stated as answering the question: "Does this artifact provide more value than the cost associated in producing it?".

2.2.3 Customer collaboration over contract negotiation

The days are long gone when developers compile detailed specifications, forcing the customer to sign off on them, go away and then returning after a year or two with a software system without further interaction with clients. Keeping the customers at a distance reduces the quality of the system and increases the the risk of failure of the project. Maximising the interaction between the customer and the developers is mutually beneficial. For example, developers no longer need to assume anything regarding specifications and customers receive software that provides what they really need.

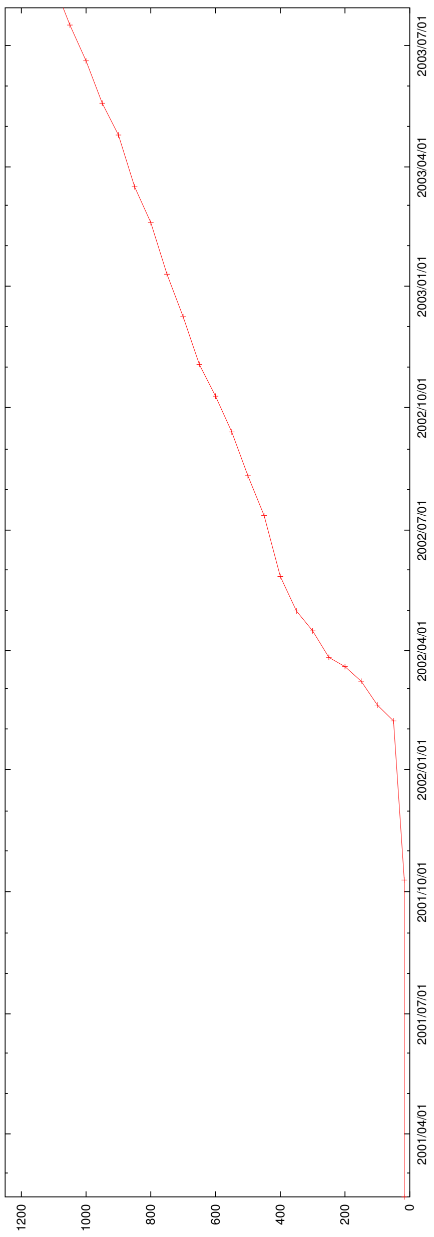
It should be noted that even though agile software development (ASD) emphasises the left hand side of the enunciated values, the right hand side is not rejected. For example, when appropriate or required, the compilation of documentation should be done. However by focusing on the left hand side the primary goal of developing working software is retained.

2.3 The Agile Alliance

As stated in Section 2.2, the meeting in February 2001 resulted in the formation of the *Agile Alliance*. The alliance started off with seventeen people. The same seventeen people co-authored the 'Manifesto for Agile Software Development'. Appendix A lists the original seventeen signatures and members of the alliance. The alliance is a non-profit organisation that promotes agile software development. The alliance is divided into a range of *programs* that represents the different facets of the organisation. These programs include the conference program, a program for each major methodology, the administration program, etc. Using the signatories of the manifesto as guide, the growth of supporters may be observed. This growth is indicated in Figure 2.1, the data is taken from the signatory information published on the agile manifesto website, see Agile Manifesto URL.

The alliance also coordinates the gathering of publications on agile software development for hosting on the alliance website. An annual conference on ASD is organised by the alliance.

Figure 2.1: Growth in Signatories (11 Feb. 2001 – 30 July 2003)



2.4 Example Methodologies

In this section, the author lists some of the methodologies that are regarded as agile. A brief description of each example is provided without the intent to analyse or criticise. In subsequent chapters, three of the most prominent agile methodologies will be analysed in greater detail. These are Extreme Programming (Chapter 3), Crystal (Chapter 4) and Feature Driven Development (Chapter 5).

The list of methodologies claiming to be agile is growing regularly. Even Microsoft is trying to jump onto the bandwagon with their Microsoft Solutions Framework (MSF) and Microsoft Operations Framework (MOF) [2002, White Paper]. Using all the right buzzwords and statements associated with agile methodologies does not necessarily mean the methodology is based on the values proclaimed by agile.

The methodologies chosen as representative of agile were selected on the basis of their authors' role in the establishment of the agile alliance and/or on their perceived popularity.

2.4.1 Extreme Programming (XP)

XP may be regarded as the most published agile methodology. There are multiple books, websites, forums and articles dedicated to this methodology. This may be partly due to the radical principles behind the methodology and partly due to the author's tenacity. Formulated during the Chrysler Comprehensive Compensation (C3) project of 1996, it has seen an increased adoption rate.

With C3, Beck brought his ideas on software development together. Using these practices, Beck restarted the failing project and successfully completed it in a year with the aid of 10 programmers.

He based XP on the following four values:

Communication: This means keeping everyone on the project informed regarding everything in the project.

Simplicity: Make use of the simplest solution that can work.

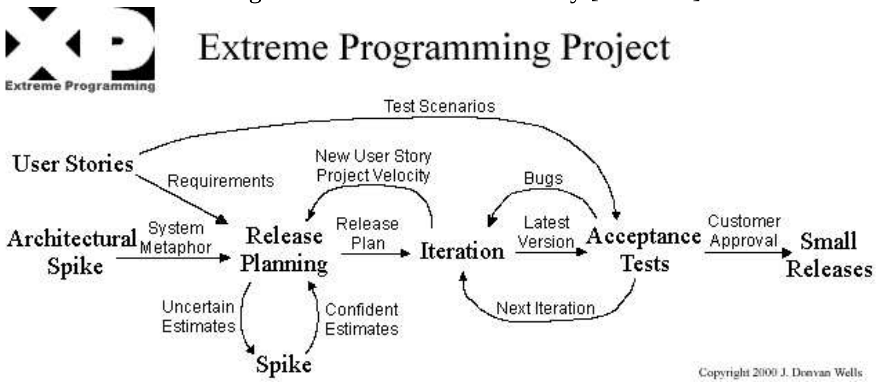
Feedback: Minimise the response between doing something and getting the result. This includes: the time between asking a question and getting an answer, if necessary from an on-site customer representative; and the time between implementing a feature and confirming its correctness using well defined automated test cases.

Courage: This refers to the ability to make difficult decisions and thus correct the project's direction.

XP is characterised by the following practices:

- *The planning game*: 'Plan' the scope and milestones for the next release.
- *Small releases*: Release the system iteratively in short cycles.
- *Metaphor*: Use a simple story of how the system works as a guideline for the project team.
- *Simple design*: Ensure that the system is as simple as possible at any given moment.

Figure 2.2: XP Process Summary [XP URL]



- *Testing*: Continuous testing of code should take place. This includes unit test cases that are defined and built by the programmers and feature test cases that are compiled by the customer in conjunction with developers. These tests ensure that at all times the system runs correctly, in respect of the test cases.
- *Refactoring*: Continuously simplify and restructure the system without changing its behaviour. The purpose is to improve the quality of code.
- *Pair programming*: Develop code using two programmers at one machine.
- *Collective ownership*: The code belongs to everyone on the team and may be changed by anyone as needed.
- *Continuous integration*: Integrate each feature as it is finished into the system.
- *40-hour week*: Team members' working hours are limited to 40 hours. It is believed that this increases productivity and moral.
- *On-site customer*: A user who can answer questions as they arise, should be part of the team.
- *Coding standards*: Code is written according to a selected standard to ensure uniformness and optimal communication.

XP emphasises both a partnership of and the separation of business and technical areas of decisions. Figure 2.2 provides a visual summary of the process.

Some key elements of XP are:

User stories. Requirement specifications are written in natural language using a single index-card. These stories are short, to the point and written by the customer with assistance from the developers.

Spikes. This term refers to a quick experimental phase, used to explore uncertainties encountered during planning and development.

Chapter 3 provides a detailed overview on XP.

More information may be found in Beck [2000]; Beck and Fowler [2000]; and Jeffries et al. [2000].

2.4.2 Crystal

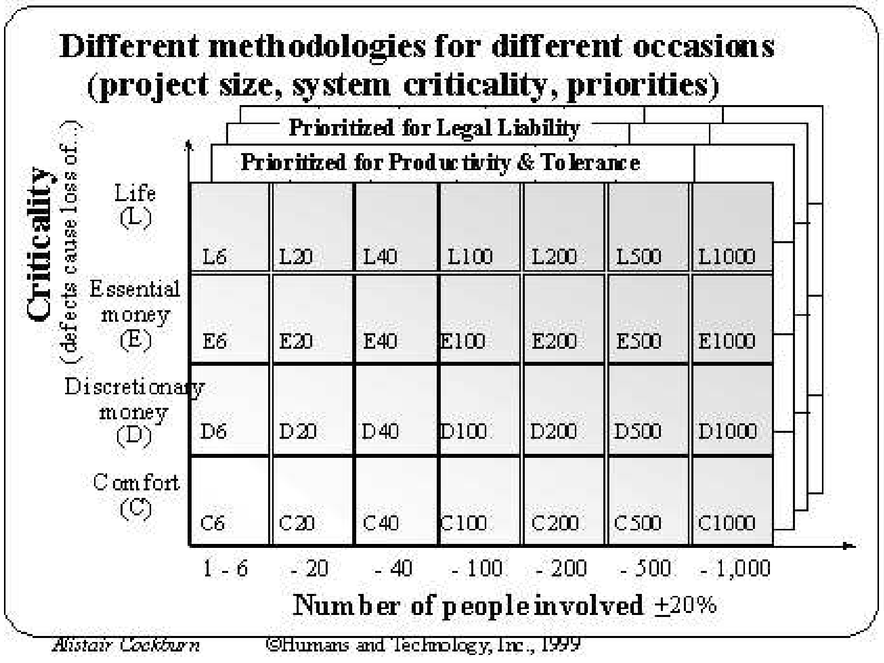
The Crystal Family [Cockburn, 2002a] is a group of methodologies developed by Cockburn. These methodologies are based on a common set of principles. They are also distinguished by the fact that they encourage adjustment to different circumstances and tuning throughout the development cycle.

This family of methodologies evolved from Cockburn's philosophy of a unique/just-in-time methodology for each project. He believes that the methodology should evolve during the project.

Cockburn defined a matrix (Figure 4.1) to suggest a methodology for use in a given project.

This is done by determining the number of people required for the project on the x-axis. Cockburn indexes these values by colour: Clear, Yellow, Orange, Red, Magenta, etc. The y-axis is used to specify the 'hardness'/criticality of the system. The indexed values are: Life (loss of life is possible if a problem occurs in the system); Essential money (loss of essential money might cause bankruptcy); Discretionary money (some money might be lost due to faults in the system); Comfort (merely causes discomfort for the user if a problem occurs). The crosspoint indicates which methodology to use.

Figure 2.3: Cockburn's Methodology Matrix [Cockburn, 2002a].


Different methodologies for different occasions
(project size, system criticality, priorities)
Prioritized for Legal Liability
Prioritized for Productivity & Tolerance
Criticality
(defects cause loss of...)
Life (L)
Essential money (E)
Discretionary money (D)
Comfort (C)
L6 L20 L40 L100 L200 L500 L1000
E6 E20 E40 E100 E200 E500 E1000
D6 D20 D40 D100 D200 D500 D1000
C6 C20 C40 C100 C200 C500 C1000
1 - 6
- 20
- 40
- 100
- 200
- 500
- 1,000
Number of people involved ±20%
Alistair Cockburn
©Humans and Technology, Inc., 1999

For example, consider the development of a informative only website. The task is assigned to a developer and a graphical designer. The nature of the system indicates a risk of discomfort if a problem exist. The matrix classification is thus a C6 project, requiring the minimal amount of 'ceremony' (such as documentation and validation). The methodologies associated with C6 includes Crystal Clear and XP.

Key practices of Crystal include: pair programming; methodology tuning through the use of reflection from workshops; iterative development; and writing test cases.

Chapter 4 provides a more detailed overview on Crystal.

2.4.3 Feature Driven Development (FDD)

FDD was created by Coad and De Luca in 1997 and later refined by Palmer among others [Coad and De Luca, 1999; Palmer and Felsing, 2002].

FDD is comprised out of five processes, namely:

1. Develop an overall model.
2. Build a detailed, prioritised feature list.
3. Plan by feature.
4. Design by feature.
5. Build by feature.

Processes 4 and 5 are grouped together and together used iteratively.

As one may observe from these processes' names and also the methodology's name, the building block of FDD is a feature. A feature is defined as function that provides value to a client.

The practices that FDD considers important are:

- Domain object modeling;
- Developing by feature;
- Individual class ownership;
- Using feature teams;
- Inspections;
- Regular builds;
- Configuration management.

Several of these practices are not unique to FDD, however the combination thereof is claimed as being distinct.

The above mentioned processes and practices are further described in Chapter 5.

2.4.4 SCRUM

First presented by Schwaber at an OOPLSA'95 Workshop on 'Business Object Design and Implementation' [Schwaber, 1995]. Schwaber based Scrum on the work by Non-aka and Takeuchi published as 'The new product development game' in the Journal of Harvard Business Review (p.137-146, January-February, 1986). Sutherland teamed up with Schwaber to extend, formalise and implement Scrum. The name Scrum is derived from the rugby game term 'scrum' that denotes "getting an out-of play ball back into the game" through team effort [Schwaber and Beedle, 2001].

Scrum is more of a managerial process to support other development methodologies and is not restricted to software development. In a software development context this means that no specific software development techniques are defined. This paradigm of thinking leads to a black-box approach for some elements of the process. Incorporation with other methodologies may be observed through the example of XP@SCRUM (XP and Scrum) [Schwaber and Beedle, 2001]. The *sprint*[4] phase is an example of this black-box approach.

Scrum consists out of three phase groups namely: Pregame, Game and the Postgame.

The Pregame phase is divided into planning and system architecture design. The new release is defined using the current *backlog*[5] and estimations of schedules and costs. High level design and system architecture for the planned release' implementation is developed.

The Game phase consists basically of multiple *sprints*, each *sprint* being followed by a review.

The last phase, also known as the Conclusion or Postgame phase, includes the steps where the software products are prepared for release to the users. This may include writing user documentation, executing integration and system testing etc.

Some interesting characteristics of Scrum are:

- It relies on a small team of no more than six members, with the option of using multiple teams;
- It has a flexible schedule and deliverables, through the ability of prioritising the *backlog*;
- It relies on short iterations (sprints) of one to four weeks;
- It advocates a 15 minute Scrum meeting everyday to help the team to stay up to date with internal progress.

Figure 2.4 shows a graphical representation of the process. Simply stated, a project starts off with a high level planning phase that results in a backlog. The backlog is constantly updated as the project proceeds. After the planning phase, a number of sprints are executed until the development is finished.

As stated previously the sprints are treated as black-boxes and thus defined on a per project and/or organisational basis. The only requirement is that each day starts off with a +/- 15 minute stand-up meeting. Each sprint should conclude by being able to demonstrate the new features added as defined by the backlog for the sprint.

A more detailed description may be found in [Schwaber, 1995; Schwaber and Beedle, 2001].

2.4.5 Dynamic Systems Development Methodology (DSDM)

DSDM was conceived in 1994 through the DSDM Consortium. Since then it has grown to become the framework of choice for rapid application development (RAD). As of writing the current version is 4.2. The DSDM Consortium is a non-profit organisation whose responsibilities include the maintaining and improving of DSDM. Another responsibility is the licensing of the framework to members for use with projects.

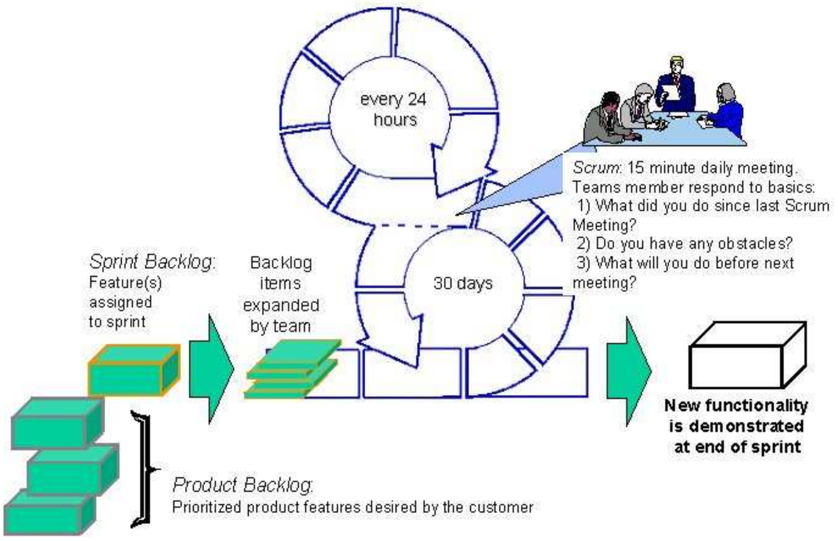
The principles behind DSDM as stated in [DSDM URL] are:

1. Active user involvement is imperative.

[4]Scrum's term for describing an iteration. "A Sprint is a set of development activities conducted over a pre-defined period, usually one to four weeks."[Schwaber, 1995]

[5]A Scrum term to describe the list of tasks/features still to be done.

Figure 2.4: Visual presentation of the SCRUM process [SCRUM URL].


every 24 hours
Scrum: 15 minute daily meeting. Teams member respond to basics: 1) What did you do since last Scrum Meeting? 2) Do you have any obstacles? 3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
30 days
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

2. The team must be empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are baselined at a high level.
8. Testing is integrated throughout the life-cycle.
9. Collaboration and cooperation between all stakeholders is essential.

The core techniques used by DSDM are: timeboxing, modelling, prototyping and configuration management. Each of these are now briefly described.

2.4.5.1 Timeboxing

In DSDM the project delivery date is fixed. Thus a fixed time-frame exists wherein the development should occur. This time-frame is broken down into smaller time-boxes of two to six weeks. Each of these time-boxes passes through three phases. Firstly, the *Investigation* phase – to determine the correctness of decisions by the team. Secondly, the *Refinement* phase acts on the feedback from the investigation phase. Lastly all the loose ends are tied-up during the *Consolidation* phase. Each time-box contains a prioritised list of requirements that should be completed at the end of the time-box. The list should contain different priorities to enable flexibility when the original plan needs adjusting.

MoSCow is a technique for prioritising requirements in a DSDM project. The acronym refers to the following rules:

Must haves: things that are fundamental to the projects' success.

o –

Should haves: things that are important but the projects' success does not rely on them.

Could haves: things that can be easily left out without impacting the project.

o –

Won't haves: things that can be left out this time around and done at a later date.

2.4.5.2 Modelling

DSDM uses modelling to enhance understanding and communication of business needs. Due to DSDM's RAD nature, the modelling techniques used should not incur bureaucratic overhead to the project. The selected modelling method should try to breach the gap between developers and business users with a focus on being user-oriented.

2.4.5.3 Prototyping

Through prototypes the users are able to validate the implementation against the requirements. Requirement articulation is also increased by enabling the user to envision more capabilities for the system. As can be deduced, this technique provides bi-directional feedback between the developers and the users.

2.4.5.4 Testing

DSDM approaches testing as a constant action throughout the life-cycle of the project. DSDM also acknowledges the importance of testing by non-technical users.

2.4.5.5 Configuration Management

The prototyping nature of DSDM requires good configuration management to enable the team to return to a previous prototype version if the current effort produced unsatisfactory results.

DSDM acknowledges the fact that it may not be suited for every project and provides a comprehensive 'suitability/risk list' to assist in determining the feasibility of DSDM on a per project basis.

DSDM is continually refined and extended through the work of the Consortium. As may be seen from an e-business tailored version named e-DSDM that has been introduced.

A more detailed analysis, review and comparison with other agile methodologies is provided in Abrahamsson et al. [2002].

2.4.6 Lean Programming

Through her experience with Lean Manufacturing and Total Quality Management (TQM), Mary Poppendieck has come to the conclusion that the same principles used in manufacturing to improve production may also be applicable to software development. The adaptation to software development is known as Lean Programming (also known as Lean Development).

Lean Manufacturing (also known as the Toyota Production System) was developed by Taiichi Ohno on request by Toyoda Sakichi, the founder of Toyota Spinning and Weaving company, to enable the production of automobiles. It is based on two values, namely rapid product flow and build-in quality.

At approximately the same time Total Quality Management was being taught by Dr. W. Edwards Deming in Japan. TQM and Lean Manufacturing were incorporated together by Toyota.

The adaptation of the above techniques to software development has brought the following 'Lean' rules to light, as defined by Poppendieck [2001]:

1. *Eliminate Waste*. Eliminate anything that does not add value to the final product.

2. *Minimize Inventory (Minimize Intermediate Artifacts)*. In software development the inventory represents the documentation that is generated and not part of the final program. The value of each document to be produced needs to be evaluated. The detail level required from the documentation also needs to be ascertain to minimise the 'inventory' and eliminate waste.

3. *Maximize Flow (Drive Down Development Time)*. The concept of reducing cycle times and the reduction of work-in-progress is applied to software development through the use of iterative development.

4. *Pull from Demand (Decide as Late as Possible)*. Responding to change is advocated through this principle. Develop for the customers current needs and adjust the solution to reflect the requirement changes needed by the customer as these changes occur.

5. *Empower Workers (Decide as Low as Possible)*. Through documentation the decision making is taken away from the developers. It is Lean Programming's viewpoint (and agile's in general) that the developers should be empowered to do what they do best – providing solutions to solve a requirement. This means that the developers are told "what needs to be done, not how to do it" Poppendieck [2001].

6. *Meet Customer Requirements (Now and in the Future)*. The 3^{rd} value of agile states "Customer collaboration over contract negotiation". As discussed in Section 2.2, it is more important to address the needs of customers than to force them into signing-off on a fixed specification.

7. *Do it Right the First Time (Incorporate Feedback)*. As with Lean Manufacturing's approach of building tests into the process to detect when the process is broken, Lean Programming builds tests into the development process for ensuring that changes do not break the system. This is accomplished through unit and *regression*[6] tests, preferably using a test driven approach – writing the test cases before the implementation.

8. *Abolish Local Optimization (Sub-Optimized Measurements are the Enemy)*. The Lean Manufacturing concept of optimising the overall process over the optimisation of sub process can be applied to software development's scope management.

[6]Regression testing is the processes of validating that changed code does not adversely affect unmodified code.

Trying to manage the scope of the project when changes starts to occur by keep to the originally envisioned scope size may only result in decision making that is based on the planned system and not the required system. Thus by managing the original scope of the project substantial resources may be squandered rather than investing these resources in providing a system to the users that is of value.

9. *Partner with Suppliers (Use Evolutionary Procurement)*. The practise of using contracts to dictate all the variables associated with a project may be more harmful to the relationship than being forward coming. Contract negotiation may also result in substantial cost. Software development should follow the example of *Supply Chain Management* in building mutually beneficial relationships with suppliers.

10. *Create a Culture of Continuous Improvement*. Making use of iterative development enables the adjustment of both the implementation and the process.

A detailed description of Lean Programming may be found in [Poppendieck, 2001] and [Poppendieck and Poppendieck, 2003].

2.4.7 Other Agile Methodologies

Other methodologies that are classified as agile include *Adaptive Software Development* by Highsmith (see [Highsmith, 2000; Adaptive URL]) and Pragmatic Programming by Hunt and Thomas (see [Hunt and Thomas, 1999; Pragmatic Programming URL]).

2.5 Agile Modeling

Derived extensively from XP and the underlying agile values, Agile Modeling (AM) is being developed by Ambler. Ambler describes AM as “a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner” [Agile Modeling URL].

The values referred to above by Ambler are basically the four values defined by XP: communication, simplicity, feedback and courage. However, AM has a fifth value, namely *humility*. Humility refers to being able to acknowledge that one does not know everything and that all the stakeholders involved in a project have certain expertise that brings value to the project.

As with XP, AM also divides its principles up into core and supplementary principles. A number of these principles are derived and/or adopted from XP’s principles. Where this is the case the reader is referred to literature on XP for a more detailed discussion on these principles. The core principles are:

- *Assume simplicity*. Use the simplest solution that will satisfy the needs and assume this to be the best solution.

- *Embrace change*. Accept the fact that requirements change as the project progresses.

- *Enabling the next effort is your secondary goal*. Ensuring that the delivered system can be extended later on is important and should be kept in mind. As Cockburn noted through his game analogy the secondary goal of the software development game is to setup for the next game. In essence it means that just enough documentation should be generated to help the next team to pick up the threads of the system with as much ease as possible.

- *Incremental change*. It is usually impossible to correctly define a model of a system with a high level of detail in a single step. Instead an iterative approach may help enable a model to evolve over time as the requirements change.
- *Maximize stakeholder investment*. Before and during modeling a modeler should keep in mind that stakeholder resource utilisation should be maximised for the whole project. This means that a thorough evaluation of the return on investment (ROI) should be done when deciding on generating models. Only generate model that will provide real value to the project and that are justifiable against the cost associated with it.
- *Model with a purpose*. When creating or modifying a model it is important to know two factors before attempting to do so. Firstly, answer the question of who the audience of the artifact will be. Secondly, get an understanding of the purpose for the model. For example, a model to explain a project to management should be different from a model to help oneself to refactor an existing design to a better pattern.
- *Multiple models*. When developing software, one needs to make use of a diverse set of models to illustrate the system. Different models highlight different aspects of the system. Even though there is an extensive range of models that may be used, one needs to select only the most appropriate ones that will satisfy the current needs.
- *Quality work*. When the result of work done is of a high quality it should be much easier to extend and refactor it.
- *Rapid feedback*. It is important to use practices that enable rapid feedback on actions taken. This results in closer relationships between the stakeholders.
- *Software is your primary goal*. Any activity in software development should only be undertaken with the fundamental aim off fulfilling the needs of the user through working software. If the activity does not comply with this principle, the execution of this activity should be re-evaluated.
- *Travel light*. Every model one decides to keep causes more reworking when changes occur. It is thus important to only keep the models that add the most value and to thereby limit the number of model as far as possible.

The supplementary principles are:

- *Content is more important than representation*. A model may be portrayed by multiple methods, for example white-board sketches, CRC cards or CASE tools. The underlying concept being illustrated should be the focus, however, still keeping in mind the purpose of the model.
- *Everyone can learn from everyone else*. No single person can know everything. However through interactions people are able to learn from one another. In an ever-changing world this ability is of utmost importance.

- *Know your models*. One needs to know the strengths and weaknesses of the diverse types of models to be able to select the most effective ones to use.

- *Know your tools*. Using the appropriate feature of a modeling tool at the correct time with the understanding of the result thereof is important for efficiency.

- *Local Adaptation*. The uniqueness of a project needs to be incorporated into the modeling methods selected for that specific project. The environment wherein the project is executed is one of the factors that influence the particular way of doing modeling. The environment is determined by factors such as the development process used and the organisational culture.

- *Open and honest communication*. Enabling people to speak their mind ensures that decisions are made based on correct and/ or valuable information.

- *Work with people's instincts*. As one gains experience one's instincts grow more accurate. These instincts come from one's subconscious and may be correct when there are no other facts on which to base decisions. Listening to one's instincts may force one to investigate a questionable scenario and thus in turn to provide proof to substantiate a decision. This principle requires courage, mentioned above as one of the values associated with AM.

Derived from the above principles are the following core practices:

- *Active stakeholder participation*. This is similar to XP's on-site customer practise but with a broader approach. AM seeks the active involvement of all the stakeholders – not only from the customer.

- *Apply the right artifact(s)*. Each model has a specific use and it is therefore necessary to select the appropriate artifact to produce to support the project's communication needs.

- *Collective ownership*. Allow any stakeholder to work on any of the models for the project.

- *Consider testability*. Always keep in mind how the design being modeled can be tested. Thinking about how the design will be tested helps the modelers to validate their models.

- *Create several models in parallel*. It might be more productive if one generates more than one model at the same time. It may also help gain a better overall understanding into the system being developed.

- *Create simple content*. Following the KISS (Keep It Simple Stupid) principle means that the representation of content should be as simple as possible by fulfilling only the needs of the project. No extra feature that is not part of the requirement specification should be represented or added into the model.

- *Depict models simply*. Models needs to be depicted at a level of detail that will satisfy the needs of the project and fulfil its purpose. For example, if one needs to see the relationship between classes there may be no need to also display all the methods available from each class.

- *Display models publicly*. For enhanced communication, the models generated should be displayed to the development team in a way that is easily accessible. This may be in the form of a "modeling wall" in the development team's office.

- *Iterate to another artifact*. Suspending work on a model that is 'stuck' and moving on to another one may prove to be helpful. Working on another model may help to gain a better understanding of the system and may resolve the problem associated with the model that 'got stuck'.

- *Model in small increments*. Breaking down development into smaller fragments enables more agility and provides stakeholders with quicker feedback.

- *Model with others*. Modeling is a communication aid for the team to gain an understanding of a problem. It is thus natural to try and use multiple members as inputs when defining the models. This enhance not only communication but also the quality of the models and in turn fosters a faster understanding my team members.

- *Prove it with code*. After one has created a model it is essential to verify that the model will actually work. This should be done by implementing the model and demonstrating the implementation to the customer. A good strategy is to take a diagram and write the test cases, then the implementation and then execute the test cases. It is worth noting that the practice of *modeling in small increments* tends to be more effective.

- *Use the simplest tools*. Make use of the simplest tool for the purpose of the model when modeling. It might be more feasible to use a white-board when modeling a throw-away diagram whose purpose is to gain a visual understanding of the current feature. In contrast, a software tool may be better suited when a model needs to be drawn for presentation to stakeholders.

The supplementary practices are:

- *Apply modeling standards*. As with XP's coding standards (discussed in Chapter 3), using modeling standards are deemed important to promote communication and unity in the team context. When modeling in a project context with multiple members involved it is necessary to use conventions to ensure consistency and better understanding. Selecting a modeling standard and enforcing it helps promote this.

- *Apply patterns gently*. When a modeller suspects the possible use of a design pattern but is still unsure, the modeller should design the system to be as simple as possible for today's needs but with the ability to be refactored to the full pattern when the pattern is deemed as the simplest solution. An example would be where a system uses two compression algorithms based on a certain selection process. This scenario leads to the possibility of using the *Strategy* pattern. However it may be less complicated and faster to use an if statement to choose between the two algorithms, with the two algorithms implemented in a similar structural way. Later on when the need arises for a third algorithm, the design may be refactored to the full Strategy pattern implementation.

- *Discard temporary models*. A model that has outlived its usefulness should immediately be thrown away. Most models usefulness is limited to gaining an understanding of a specific issue when developing the system. As soon as the developers have gained an understanding and implemented it in code there may be no further need for the model because the synchronisation between code and the model might be lost very quickly.

- *Formalize contract models*. Because contract models[7] imply a contract between parties it is necessary to standardise these models. More effort and ceremony should be spent on developing and maintaining these models. However to conform with the travel-light principle, one needs to keep the number of contract models for a project as low as possible.

- *Model to communicate*. Models are used to exchange information between different parties. It is important to realise who the audience of the models are going to be; modeling accordingly will maximise the communication between the parties.

- *Model to understand*. Modeling is used to explore the problem space. Through modeling one can design solutions. Multiple solutions can be compared to select the most appropriate one for implementation.

- *Reuse existing resources*. When possible one needs to investigate models that are already available in the problem domain. These may include existing business process models, data models and requirement models. Applying design patterns gently is also a means of reuse.

- *Update only when it hurts*. Models should only be updated when the updated model will provide more value to the development effort than not updating it. Having to update a model requires resources that could be better utilised for other development needs. Models may still be useful enough without updating them, as with a street-map that still satisfies one's needs even though it is outdated.

Agile Modeling brings the benefits of modeling to agile software development in an agile manner. Although some of the agile methodologies de-emphasises extensive modeling, the usefulness of modeling in software development is acknowledged.

2.6 Conclusion

From the above description of the common ground between the agile methodologies and from the description of several of these methodologies, one may start to gain a greater understanding of what ASD entails.

The manifesto highlights the important role humans play in developing software both as the customer and the developer. From the brief introduction to some of the agile methodologies one may observe that they have the potential of providing the benefits advocated by their proponents.

Table 2.1 provides a short comparison of the methologies that have been described in the previous sections.

The following three chapters will provide more in-depth overviews of certain agile methodologies. The agile methodologies to be investigated are XP (Chapter 3), Crystal (Chapter 4) and FDD (Chapter 5).

[7]Models representing API's (Application Programming Interfaces), XML DTD's etc.

Table 2.1: Agile Methodologies Comparison

Methodology	Distinctive Features
XP	• Highly disciplined • Project size limited to 20 people with the standard implementation • Requires an on-site customer • Uses a Test-First approach
Crystal	• Advocates a methodology-per-project philosophy
FDD	• Is a model driven approach • Includes a modeling-in-colour technique
SCRUM	• A managerial methodology that may be used in combination with other software development methodologies • Focusses on a daily, 15 minute stand-up meeting
DSDM	• Primary RAD framework in Europe • Developed and maintained through a consortium • Relies on a licensing scheme for its usage
Lean Programming	• Based on the Lean Manufacturing philosophy, originally developed for Toyota's assembly plant. This philosophy is used by the manufacturing industry

Chapter 3

A Critical Overview of Extreme Programming (XP)

3.1 Introduction

Extreme Programming (XP) is a 'light-weight' software development methodology (more correctly, an agile methodology). According to Beck [2000] it "is a discipline of software development".

This means that a software team has to comply with certain well-defined and well-circumscribed processes when developing software in order for their development methodology to be classifiable as XP. The most important of these processes will be spelled out below.

XP evolved out of the problems faced by software development, including: software not meeting customer expectations; reduced applicability of delivered software; and the inability to meet schedules[Beck, 2000; Cockburn, 2002a].

XP is summarised in the following sections by referring to the four variables of a project as specified by Beck [2000] as well as to the values, principles and practices defined for XP. These sections (i.e. 3.2 to 3.5) are mainly derived from publications by the original XP team members. (See Beck [2000]; Williams et al. [2000]; XP URL). Section 3.6 provides insights into the suggested physical working conditions and Section 3.7 describes the process. The last section will discuss issues regarding XP that are perhaps of a controversial nature: the feasibility of an on-site customer (Section 3.8.1), scalability of the methodology (Section 3.8.2), the impact of changeable requirements on a methodology (Section 3.8.3), the appropriateness of pair programming (Section 3.8.4), the accuracy of Beck's cost of change curve (Section 3.8.5) and the lack of documentation generated by XP (Section 3.8.6).

3.2 Four Variables of a project

Beck [2000] points to four interrelated variables that influence a project: scope; cost; quality and time. The management of these variables controls the output of the project. These variables are tightly coupled, thus changing one of the variables causes changes to the rest.

3.2.1 Scope

Thus, as the scope becomes larger, the system tends to become more complicated and therefore requires more money and time to complete. However, based on some scope metric such as function points[1]one should not simply assume a linear or near-linear relationship between scope and the other variables. In some applications, a scope enlargement may significantly add complexity, while in other cases, the added functionality may be relatively simple and loosely coupled to the remainder of the system.

3.2.2 Cost

Clearly if costs are constrained, then quality and/or scope and/or time have to be constrained as well. However, merely providing more money does not automatically guarantee a better balance between the variables but may have unexpected and/or opposite results. For example, it would be naï¿½e to imagine that one could merely provide more money in the hope that the scope and/or quality of the task would somehow "automatically" increase. Instead, the effect might merely be to take pressure off the developers, allowing them to complete the task at the same quality and scope levels, in somewhat longer time. (This is reminiscent of the "law" formulated by Parkinson [1958]: "Work expands to meet the time available for its completion.").

3.2.3 Quality

The level of quality required from the output has an important influence on the rest of the variables. The higher the quality demanded, the more money and time will be needed and *vice versa*.

Of course, the idea of software quality remains somewhat nebulous:

The quality of the software to some clients could mean an easy-to-use interface for users such as graphical interfaces with similar feeling as other applications; or a 24/7 uptime for services provided by the system; being able to manage x number of transactions per second; having a guaranteed 100% data integrity for input and/or output; the system provides the most features.

On the other hand to a developer, quality might mean delivering a product that is based on the best architecture possible; that has the cleanest code; or that provides the optimal performance.

From the above we can deduce that quality may reflect three dimensions:

- The requirements of the system. The system requirements state what the end products should look like from the user's point of view. Comparing the actual product developed against the stated system requirements therefore gives an indication of the systems quality from the users's perspective. It should also be noted that the system requirements directly relates to the scope (Section 3.2.1) of the system.

- The resourcefulness of the development team. This include the motivational and incentive factors as well as experience, training and the working environment (See Section 3.6).

[1]ISO/IEC/JTC1/SC7 Standard #14143 definition for Functional Size: "A size of software derived by quantifying the *functional user requirements*"

- The development process used. This statement raises a lot of debate in the software engineering community regarding which process is the best to use and resulting in the highest quality of software produced. This dissertation, and this chapter in particular, is part of this debate by providing a discussion on XP and how it achieves quality.

Although a full discussion on quality and its definition in software development is beyond the scope of this dissertation, how XP allegedly supports it will be mentioned throughout this chapter.

3.2.4 Time

Time is not necessarily related to cost in a linear fashion. This can be seen from the cost-of-change/time curve (Figure 3.2) introduced by Beck [2000]. This curve is somewhat controversial and a discussion is provided in Section 3.8.5.
In other contexts, too much time can be as crippling as too little time. If strict discipline and client interaction are not maintained, it may afford developers the opportunity of creating unneeded features. A balance needs to be struck between unrealistically tight time constraints on the one hand (which encourage cutting of corners and which tend to demotivate the developers) and too much time (which runs the obvious risk of promoting low productivity).

The interrelationship between scope, time, quality and cost that Beck has identified clearly applies to many contexts and is not limited to XP, nor indeed to software production. Furthermore, the question of determining values for these variables is non-trivial and is a general issue of concern in software engineering and project management. It is fairly easy to map time and costs to numeric values. The determination of scope is part of the 'traditional' requirements engineering process for which standards are being developed through ISO/IEC initiatives. Metrics for quantification of scope have been developed and standardised but are somewhat elusive. In any production context, these variables should be considered. Variables that can be quantified (e.g. time / cost) should be made explicit. If explicit metrics for other variables are not appropriate, they should be qualitatively articulated.

The distinctive part of XP is how agreement between contracting parties is to be reached on these values. The methodology requires that the "external forces (customers, managers)" may determine the values of any three of the variables. It is then the prerogative of the development team to propose "the resultant value of the fourth variable" [Beck, 2000]. Clearly, this is not meant to eliminate negotiations but to ensure a balanced and realistic agreement.

3.3 The values of XP

Values are the characteristics/qualities of something or someone that the valuer regards as desirable and important. Humans sometimes encapsulate these into a mission and/or vision statement. Values are adhered to either consciously or unconsciously.

Beck has highlighted four values that he believes ought to be part of any software development endeavour. In particular in the case of an XP initiative, these values are to be explicitly striven after. This does not mean that other development methodologies do not espouse these values, but rather that in XP these values are actively pursued.

The XP philosophy thus dictates that for a project to succeed, the project team needs to embrace four values: communication, simplicity, feedback and courage. Each of these values is now briefly reviewed.

3.3.1 Communication

Communication is widely recognised as important in successful human relationships, whether they be personal (marriage) relationships, work relationships, international (political) relationships, etc. In fact, in recent times, there has been renewed and growing interest in "knowledge management" as an important component in contributing to successful companies. In XP, communication between team members is strongly emphasised.

For a project to succeed, project team members need to be informed. A project consists of a team of individuals whose work influences one another. For this arrangement to work smoothly individuals and groups obviously need to communicate with the rest of the team, to keep them up to date and be kept up to date themselves.

XP recognises that successful projects heavily depend on the right communication flowing between stake-holders. Through the use of techniques like pair-programming, on-site customer, the environmental conditions and white-boards XP gives expression to this value.

3.3.2 Simplicity

The notion of simplicity has long been recognised as an important element in solving problems. This notion was expressed in the middle-ages through Occam's razor[2] and later by software development's KISS (Keep It Simple Stupid) principle.

XP uses the statement: "What is the simplest thing that could possibly work" [Beck, 2000], bringing to light its belief that implementing the simplest solution to solve today's problem is more important than using a complicated solution for tomorrow's **possible** requirements.

This value enhances communication because simple things are easier to communicate [Cockburn, 2002a].

In software development this means that one only thinks of the current requirement and NOT of *possible* future needs of the system; generalisation and extra functionality add more complexity to the implementation. This refers to the "You Aren't Gonna Need It" (YAGNI) principle.

3.3.3 Feedback

To have factual information on the current state of the system provides the best feedback and understanding to the development team. Providing feedback within the smallest possible time scale enhances decision making, thus management, on the project and ensures that the project moves in the correct direction.

Feedback ties directly in with the rest of the values, and influences each one directly: feedback is part of communication and increases courage (see next sub-section).

It is directly reflected in the practices of pair-programming (Section 3.5.7), short releases (Section 3.5.2) and automated testing (Section 3.5.5). It may also require courage.

[2]William of Occam (or Ockham) (1284-1347) stated "Entities should not be multiplied unnecessarily." which in turn can be restated/simplified as meaning "Keep things simple"

3.3.4 Courage

This refers to the ability to make difficult decisions and thus correct the project's direction.

The saying of "taking a step backward to take two steps forward", is an example of this.

XP relies on this value to enable agility and ensure the *embracing of change* principle.

The previously mentioned values help to *encourage* the team by giving them feedback on decisions; communicating strategies and using simple solutions.

Courage also refers to adhering to XP practices and not reverting back to traditional habits when the pressure starts to build up. Thus the opposite of courage is the fearfulness of risking and thus not giving feedback, the laziness of staying in a comfort zone where one does not express an opinion or communicate. Not expressing opinions may harm the team by providing the gap for individuals to dominate the group; reducing responsibility for actions and the reduction of synergy occurs.

These values are regarded as essential for a team to be able to practice XP. They should be part of the team's culture.

The above values forms the basis for the principles of XP. These principles are now discussed in the following section.

3.4 Principles

In this section the principles to use when developing software are discussed. These principles were derived from the values listed in Section 3.3 and are divided into the basic or fundamental principles and the secondary group of principles. These principles will form the basis for the practices used in XP. (See Section 3.5).

3.4.1 Fundamental Principles

These are the basis on which the practices are mainly built. They form the basic underlying 'rules' for XP.

3.4.1.1 Rapid feedback

From learning psychology we know that the time between action and feedback is critical for optimal learning [Robbins, 2001]. "... one of the principles is to get feedback, interpret it, and put what is learned back into the system as quickly as possible." [Beck, 2000].

This in turn enhances changeability and quicker fault correction. In XP this is implemented in various practices: having an on-site customer (Section 3.5.11); carrying out automated tests (Section 3.5.5); pair programming (Section 3.5.7) and ensuring that the environment is appropriate (Section 3.6).

3.4.1.2 Assume simplicity

Solve *today's* problem with the simplest solution that can work, without trying to solve future problems that might not even materialise. This again indicates the reliance on Occam's razor and KISS principle (see Section 3.3.2).

3.4.1.3 Incremental change

A series of small changes makes a big difference and is easier to manage. This can be compared to driving a car: using small adjustments to the direction a person is able to travel to the final destination [Beck, 2000].

Refactoring[3] in combination with automated testing enables this principle.

3.4.1.4 Embracing change

"The best strategy is the one that preserves the most options while actually solving your most pressing problem." [Beck, 2000].

Welcoming change during development is one of XP's main advantages [Beck, 2000] over other methodologies because it is well suited for developing systems with requirements that are in a constant flux. This ability differentiate it from most other methodologies that tries to prevent changes to specifications.

By allowing the customer to update the system requirements may also lead to more benefits, including customer satisfaction.

"Responding to change over following a plan"[Fowler and Highsmith, 2001; Agile Manifesto URL] is also one of the four values of agile methodologies.

3.4.1.5 Quality work

Everybody wants to do a good job, to ensure job satisfaction [Robbins, 2001]. When one does not enjoy one's work one does not give one's best and this reduces the quality of the product. This in turn means that every member of the team must be allowed to do his best.

Doing quality work enables the team to produce high quality products, leading to more positive feedback.

3.4.2 Secondary Principles

The more general principles that are encouraged and supplemental to the fundamental principles are as follows:

3.4.2.1 Teach learning

This principle involves eliminating doctrines with regard to learning, by teaching developers how to learn instead of learning in a predefined way (spoon-feeding). Beck uses the example of teaching testing in a specified way (a step-by-step approach) versus teaching strategies for testing (providing guidelines) [Beck, 2000]. By providing guidelines on how to test, the team are able to evolve their own techniques and customising them as needed.

This idea coincides with Cockburn's concept of three levels of listening and learning [Cockburn, 2002a]. The three levels are divided according to the detail of explanation required to understand a concept. As an example Cockburn refers to writing books on methodologies, where in a level one book the author would provide a lot of technical

[3]"When we remove redundancy, eliminate unused functionality, and rejuvenate obsolete designs we are refactoring" [XP URL] from the design and code. Refactoring thus means "A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality - simplicity, flexibility, understandability, performance" [Beck, 2000]

details including templates etc. for the methodology, whereas with a level three book the author would only give abstract ideas on what a methodology might contain.

The result of teaching learning will be that teams implementing XP will customise (Section 3.4.2.8) the practices and process to fit the project and to reduce the baggage[4] (Section 3.4.2.9) associated with their projects.

3.4.2.2 Small initial investment

Investing the correct amount of resources ensures that a project starts off on the right foot. By following a tight budget the stakeholders are forced to focus on what needs to be done [Beck, 2000].

This principle is illustrated in the following two instances:

- *human-resources*; The project should not start of with a full development team committed to it. Instead it should start of with say two developers and incrementally grow to accommodate more developers as the project progresses.
- *financial-resources*; The customer should not pay a fixed upfront amount of money for the system but rather commit a small initial amount for the exploration phase and then pay per release or per feature produced.

Doing this ensures that the business and developers are not over committed to the product's development, in the negative sense, so that they have the courage and ability to stop the project when it is not beneficial to the parties anymore.

3.4.2.3 Play to win

There is a difference in playing to win and playing not to lose. The latter means doing everything "by the book" [Beck, 2000] to be able to say they did the task as specified.

Sports is a good example to illustrate this: when a team is able to be relax (in the sense of being focused and playing by the rules without making mistakes) throughout the game, because of their mindset that they will win even if they suffer some setbacks, then they are playing to win. However, when a team shifts its goal from scoring points to only making sure the other team does not, then they are only playing not to loose.

This way of thinking seems to stem from the value of courage; being able to focus and continue following the practices and principles of XP when the going gets tough will enable the team to win.

Following this principle helps the team to focus on what is relevant to the project's success and not expending energy on the formalities, pretending to do what is expected.

3.4.2.4 Concrete experiments

For every decision that is made, it's validity should be tested to ensure it was the correct one. XP does this through communication, unit- and functional tests.

When a choice needs to be made regarding design, for example, a test should be created against which to validate the choice. This reduces the risk involved in guesses and hopes.

According to Beck, the result of a design session should be a group of experiments that address the questions raised. For example a decision regarding a data-structure

[4] Artifacts that the team need to keep and maintain over the course of the project. These includes documentation and restrictions to specific development tools.

that will support the functional and non-functional requirements should be based on experiments that tests the best options.

3.4.2.5 Open, honest communication

The stakeholders need to communicate with each other without punishment/rebuttal from the rest of the team.

As part of of the team culture and processes everybody needs to be able speak his/her mind without inhibitions.

3.4.2.6 Work with people's instincts, not against them

XP uses people's short-term self interest to serve the team's long-term interest. Beck recognises that people like to win; to learn; to interact with other people; to be part of a team; to be in control; to be trusted; to do a good job and to have working software [Beck, 2000].

The methodology utilises these instincts through its practices to ensure that it will and can be followed. This tends to differentiate XP from other methodologies where creativity/initiative is often inhibited by rules and prescriptive practices.

3.4.2.7 Accepted responsibility

Responsibility for a task should not be given to a specific person but should rather be accepted by him. It is assumed that in a team situation where a task needs to be completed, someone will eventually choose to do it, even if the task is deemed to be detestable [Beck, 2000; Gido and Clements, 1999].

3.4.2.8 Local adaptation

XP is not a rigid process that needs to be followed exactly. It is mainly a guideline. Every team should adapt and shape it to their needs [Beck, 2000].

This is also an approach that is promoted by Cockburn who recommends that a methodology should evolve to fit the circumstances. Of course, the adaptations made cannot be arbitrary. Rather the evolution should occur within the broad framework specified by XP. Practical experiences[5] have shown this to be true.

3.4.2.9 Travel light

To enable them to change direction easily (Embracing change) the team should not carry a lot of baggage.

Commitment to a specific set of tools should be avoided. Documentation for bureaucratic purposes (such as big upfront requirement specification that needs to be kept up to date) may also cause baggage to be dragged along, see section 3.8.6 for a discussion on documentation.

Being able to 'travel light' means the team is more *agile*.

Artifacts should be : few; simple and valuable [Beck, 2000].

[5]The Equinox team experienced increased success as they followed XP practices more strictly [Equinox Interview, 2002].

3.4.2.10 Honest measurement

The correct types and detail level of measurement needs to be used, that will satisfy the information needs to be gained. For example using ‘lines of code’ as measurement for productivity or quality is useless in modern programming languages in particular when refactoring and optimisation is done.

Measurements such as Feature Points (ISO/IEC/JTC1/SC7 workgroup [ISO/IEC JTC1-SC7 URL]) may be of more use and more realistic for example.

3.5 Practices

Striving for the values outlined in Section 3.3 and using the principles defined in Section 3.4 as guidance, an XP team still needs some concrete practices as reference to work by. XP practices are derived from the four basic software development activities: listening (What does the customer want? In essence, requirement extraction); designing (structuring the system); coding and testing. Most of the practices specified by XP are not new but were abandoned and or rejected by traditional methodologies because of contradicting beliefs by their authors. Some of the practices are even found in ‘traditional’ methodologies in some form. The XP practices are: the planning game; short releases; use of system metaphors; simple design; testing; refactoring; pair programming; collective ownership; continuous integration; 40-hour work week; having an on-site customer and using coding standards.

These practices are supportive of each other. The weaknesses of one practice are overcome by the strengths of the others [Beck, 2000].

3.5.1 The planning game

"A good plan today is better than a perfect plan tomorrow."
- General George Patton

This is where the two parties (business people and the developers/technical team) together make decisions on what should be done. Each party brings its own set of interests to the table. For the business it is: scope (which features needs to be part of the product); priority (selecting the most important features); release composition (when does a group of features provide value to the business) and release dates (when will the presence of the product provide optimal value). And for the developers: estimates (time-frame to implement a feature); consequences (providing feedback on business decisions that are influenced by technical factors); process (work and team organisation) and a detailed schedule (the order in which *user-stories*[6] will be implemented).

This means that decisions by business members influence the decisions made by developers and *vice versa*.

Beck divides the planning game into three phases: exploration; commitment and steering.

A project should start off with a simple plan which is then refined in iterations as the project progresses. This is possible by enabling the customers to update the plan themselves with estimates from the programmers.

This technique is implemented with different granularity during the process, see Section 3.7 for a more technical description. Instances of the game includes release

[6]The customer writes a story describing what the system needs to do. It consists of about three sentences in customer terminology [Beck, 2000; XP URL]

planning (Section 3.7.1) and iteration planning (Section 3.7.2). Where release planning deals with the story level and iteration planning with tasks. Pairs may even use the planning game when implementing a task.

3.5.2 Short releases

Releases should be as short as possible containing the features that will give the most value to the business.

Releasing the product after only a few months or even weeks is possible when the planning game is used to implement the most valuable stories first and by using continuous testing and integration to ensure correctness throughout development.

This in turn enhances faster feedback from the users.

3.5.3 Metaphor[7]

The goal of using metaphors is to create a view/understanding of the system that everyone can relate to. This enables: a common vision of the system; mutual vocabulary for describing the system; generalising the problem and highlighting the possible architecture of the system [Wake, 2002; XP123 URL].

An example would be to describe pair programming as tag-team wrestling [XP123 URL] or the C3 payroll system[8] as lines, buckets, and bins.

When a good metaphor is used it may increase communication and understanding of the system.

From the survey reported in [Rumpe and Schöder, 2002] it seems that this is the most difficult practice to follow because most people have difficulty understanding how it should be applied. It is also the least used practice according to the respondents. This seems to suggest that practitioners need to get better acquainted with what this practice means and how they should apply it.

3.5.4 Simple Design

Always keep the design of the system as simple as possible. This is done by only adding functionality when required and by simplifying complex code with continual refactoring. This practice also requires that no duplication in logic exists and that code should be as readable as possible.

The rule of KISS (Keep It Simple Stupid) is of utmost importance. See Section 3.3.2 for a discussion on simplicity.

3.5.5 Testing

This is one of the more important practices, with the other practices relying heavily on it.

XP uses automated tests to build confidence in the program being developed. By running all the tests when a change (new feature or refactoring) has been made one not only confirms that the change did not break the system but also boosts the developers confidence and morale. Through testing the team gets courage to change the code

[7]"A story that everyone - customers, programmers and managers - can tell about how the system works" [Beck, 2000]

[8]The C3 (Chrysler Comprehensive Compensation) payroll system was the first project using XP [Beck, 2000].

as needed with the knowledge that problems will be caught as soon as the change is complete and the tests are run. Having tests enables the program/system to become "more capable of accepting change..."[Beck, 2000].

By writing tests first, requirements can be better understood before implementation is started.

The programmers write unit tests to validate their implementations and check their understanding of the requirements.

Customers specify and write, with aid from the programmers, functionality/acceptance tests for the system according to the user-stories.

Tests need to be automated to ensure quick execution on a regular basis.

The use of automated testing was also heavily sought after by Netscape and Microsoft for their project development since the middle 1990's [Cusumano and Yoffie, 2000]. However it was only partly successful during that time due to a lack of tools and technology, since then tools such as XUnit and JUnit with its extensions have become widely and successfully used.

3.5.6 Refactoring

The use of coding standards, collective ownership, pair programming and automated testing provides the possibility for continuous refactoring. Code is continuously refactored when and as needed by any pair during development.

After a new feature is implemented and all the tests succeed, developers may be able to simplify the implementation because of retrospect, insights and they should do so. This is only one example of the many instances where refactoring may occur.

A lot of research has been done on this topic. Some publication include Fowler [1999]; Roberts [1999]; and Opdyke [1992] and a detailed discussion is beyond the scope of this dissertation.

3.5.7 Pair programming

Two programmers work together on one machine. One uses the machine to implement the current feature, while his partner thinks strategically, thus globally about the proposed solution. Together they develop the tests and implementation, refactoring as needed. Pairs are assigned as needed by the current problem, enabling knowledge exchange between developers.

This practice is one of the fundamental elements of XP and also one of the more controversial ideas, causing a lot of debate between traditional and agile methodologists and their followers. See section 3.8.4 for discussion on this controversy.

3.5.8 Collective ownership

All the developers share responsibility of the system (the code), enabling a pair to improve it immediately when they need to. This also increases understanding of the system by the whole team and in turn increases communication.

The practice of allowing any developer access to any code is a controversial idea that generates a lot of opinions in the software engineering/development field, even between the different agile methodologies.

The opposing methods include no-ownership and individual ownership. Both are rejected by XP because of their negative effect on communication, refactoring and changeability. Restricting access to code causes bottlenecks during development if mutual exclusive functionality needs to be added to a single unit of code, for example a class in a object-oriented language.

3.5.9 Continuous integration

Changes and updates to code made by pairs should be integrated regularly, after a few hours and not later than at the end of a day of development. This confirms that local changes did not break the system as the integrated system should always run tests with a 100% success rate.

One of the effects of this is that there is always a stable and current version of the system available to be released if needed.

Another result is that the different pairs of developers can update their local working copy of the system as soon as a pair has integrated their changes, thus reducing conflicts and ensuring easier integration when it is their turn.

Regular integration is also practiced by Microsoft with their nightly builds practice [Cusumano and Yoffie, 2000].

3.5.10 40-hour week

Limiting the working hours helps to ensure that developers are focused and creative, leading to better morale and productivity. “Overtime is a symptom of a serious problem on the project. The XP rule is simple - you can’t work a second week of overtime.” [Beck, 2000].

The “40-hours” is not a limit. This can be customised to fit the team’s tolerance, because some people are able to work 45 hours a week and some only 35.

An advantage of this rule may be that domestic problems due to “working-late” are reduced, helping team members build better relationships with the organisation. This brings the saying of “All work and no play makes Jack a dull boy” to mind.

3.5.11 On-site customer

Having at hand a customer who will be using the system and who understands the domain, to answer questions and to validate functionality is important to an XP project. This speeds up development and helps with quick correcting of deviation from the real requirements.

This practice reduces misunderstandings and invalid assumptions by developers.

The business, those that have not experienced the benefits, is mostly against this practice because of the cost and time associated with committing a user who might be needed for the organisation’s primary work. This debate is discussed in Section 3.8.1.

This concept is not unique to XP but is based on business thinking that states that a business should have close relationships with its customers. This is also a practice encouraged by project management schools [Gido and Clements, 1999].

In the end this is probably the most difficult requirement of XP and it will take a lot of explanation and motivation to convince the customer to follow this practice.

3.5.12 Coding standards

For optimal collective code ownership and pairing, the team needs to adopt a set of rules to dictate the practice of writing code for the system. Adoption needs to be voluntarily by all members and should enhance communication. Thus it should be a team decision.

Sticking to a common style of coding is important because when there is collective code ownership (Section 3.5.8), pair programming (Section 3.5.7) with constant pair swopping and refactoring (Section 3.5.6); the code should always look the same without the ability to distinguish who in the team wrote it. Having a consistent style for the code enhances readability and productivity.

The standard, as with all practices, should comply with the XP values, thus: enhance the communication in the team and be as simple as possible and in turn simplify the code.

3.6 Environment

An important consideration that helps XP to work, is to ensure that the environmental condition wherein the XP team functions is appropriate. In simple terms this refers to the working area of the development team.

This form part of a specialised field in human science, where the question is: 'What are the perfect working conditions for optimal productivity?'. For intellectually intensive work the emphasis lies with increasing communication and knowledge management [Robbins, 2001].

Cockburn gives a detailed discussion on communication and the working environment in [Cockburn, 2002a]. The conclusion is that the closer team member are to each other, the higher the level of information flow is between them and this in turn increases productivity.

In the XP context the working environment is setup for ultimate flow of information in the shortest time possible.

Beck's guideline for the workspace layout is:

> The best setup is an open bullpen, with little cubbies around the outside of the space. The team members can keep their personal items in these cubbies, ... and spend time at them when they don't want to be interrupted. ... Put the biggest, fastest development machines on tables in the middle of the space ... This way, if someone wants to program, they will naturally be drawn to the open, public space. From here everyone can see what is happening, pairs can form easily, and each pair can draw from the energy of the other pairs who are also developing at the same time.[Beck, 2000]

Beck also suggest a common place for team members where they can relax on couches with coffee and toys to help them step away for a break from development when they get stuck with a problem.

Another feature of the environment should be the use of white boards to enhance communication. It may be used for quick brainstorming sessions and the displaying of tasks, stories and project status.

Once again all the values, principles and practices of XP and mostly the local situation should be taken into account when setting up the environment for development, because it should be customised to fit with the development team's culture, keeping in mind the organisational culture and regulations.

3.7 The process flow

Figure 3.1 gives a visual summary of the process used by XP as described by [Beck, 2000].

As previously stated, XP is a highly iterative process thus the whole process can be repeated as many times as required to satisfy the customer. Keeping this in mind and the fact that the same rule applies to the sub parts, then the process flow can be described as starting off with the Release *Planning Game* (Section 3.7.1) which produces a small release as output that has been validated against *Acceptance Tests*. This process is repeated iteratively for the life span of the project.

3.7.1 The Release Planning

The duration of a release is usually between one and three months, where the planning part may take between one to three weeks.

Being an instance of the *planning game* means that this phase consists of the three sub-phases exploration, commitment and steering (see Section 3.5.1). These phases are discussed below:

3.7.1.1 Exploration

During this sub-phase both parties try to get an understanding of what the system is required to do. This is done by writing *user-stories* describing each feature of the system.

The customer does this using natural language and index cards.

The development team takes these *user-stories* and assigns *estimates* to each one. However, if the developers feel that the story is not specific enough or that it is too broad they should ask the customer to *split* the story up into more specific stories. Developers may also use *spikes*[9] to help determine estimates when they are unsure.

3.7.1.2 Commitment

Here the scope and date for the release is determined. The scope is determined by the list of stories that need to be completed for this release. To decide on which stories to include in the list the customer first needs to sort the stories by their *value* to the business. A value of 1 (high), 2 (medium) or 3(low) should be assigned to each story and the stories are then grouped accordingly.

Developers then sort these stories according to the level of *risk* involved using the values 1 (high), 2 (medium) or 3(low).

The result of the above should be matrix that categorises the stories according to the level of value to the business and the level of risk involved in implementing them.

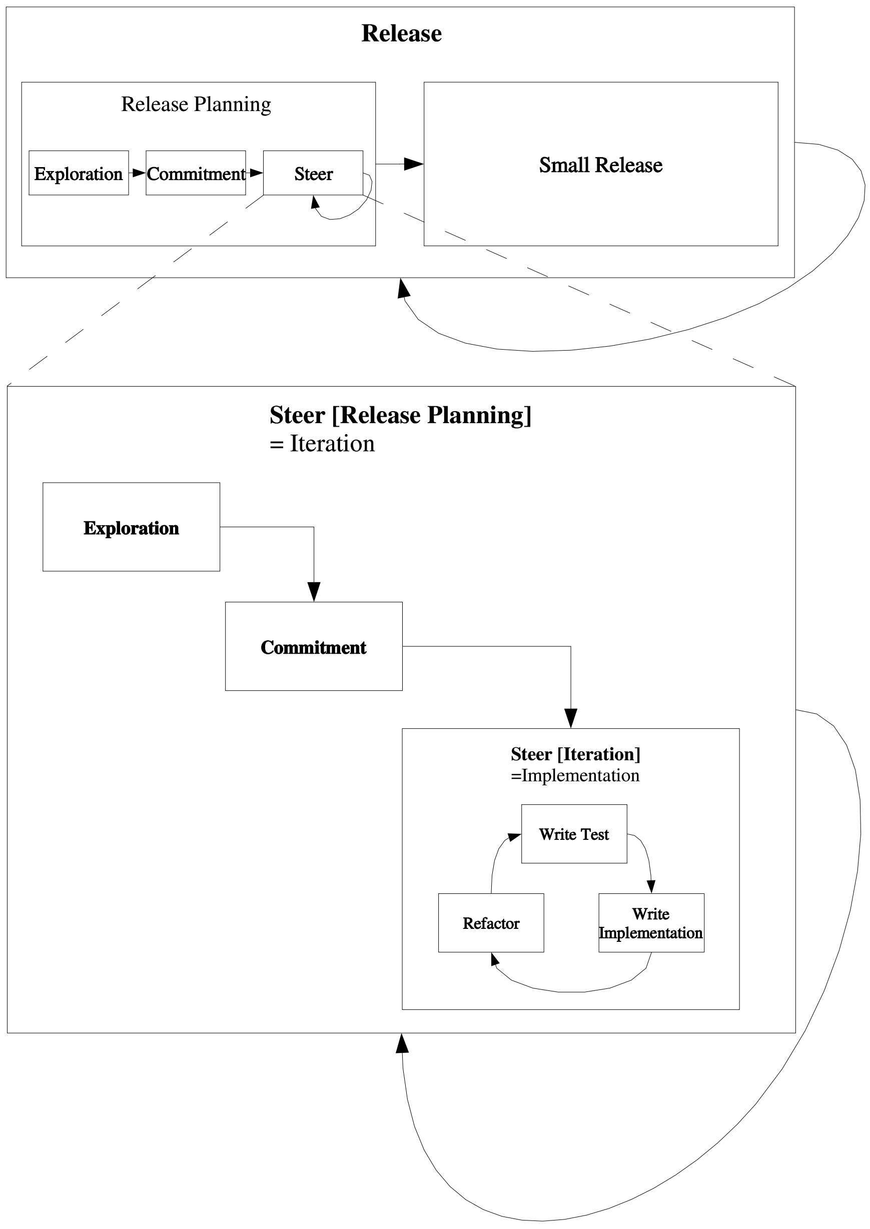
The next step is for the developers to estimate the current *velocity*[10] of the team.

The business/customer finally chooses the *scope*, either by selecting the story-cards that will fit into the current release time-frame or by selecting all the cards and then calculating the release date using the *velocity* and *estimates*.

[9]This is a quick programming or literature exploration of an issue that developers are unsure of.

[10]How fast the team can program in Ideal Engineering Time per calendar month. This is indicated by specifying the number of story-points the team can implement during a iteration.

Figure 3.1: XP Project Process



3.7.1.3 Steer

The purpose of this phase is to enable the team to update the plan using the knowledge gained thus far. Figure 3.1 shows this sub-process as looking similar to Release Planning.

The four possible actions that may be taken during this stage are:

- *Execute an Iteration*. The team implements an iteration worth of stories (see Section 3.7.2).
- *Recover*. If the development team realises that they have overestimated their velocity, they may request the customer to re-select from the remaining stories (for the current iteration) those that should remain in the current release, based on the newly calculated velocity and/or estimates.
- *Adding of new stories*. Business needs may change and this will require the ability to add new stories to the current iteration. When this occurs the customer has to replace stories with these new ones, keeping in mind the time estimates for each.
- *Re-estimate*. The development team may feel that they need to re-estimate the time to complete the remaining stories and reset their velocity, if they discover that the estimates provided previously do not reflect their current understanding of the system.

3.7.2 Iteration

An iteration's time scale is approximately one to three weeks and the *planning game* (see section 3.5.1) practice is followed using tasks as the detail level.

The phases specified by the *planning game* and used during an iteration are:

3.7.2.1 Exploration

Developers take the stories and break them down into *tasks*. The task descriptions are written on index cards. These tasks should be spitted or combined to form tasks that can be implemented in a few days, in a similar way as with stories.

3.7.2.2 Commitment

Responsibility for a task is *accepted* by a programmer and he/she provides an *estimate* on how many ideal engineering days it will take to implement. It should be noted that the effect of pair programming is not taken into consideration because its effect is reflected in the *load factor* that each programmer are able to handle. The *load factor* is a calculated value that reflects the relationship between ideal programming days and calendar days; it is usually based on historical data.

After the above steps the team should do some *balancing*. Each programmer sums the time estimates of his/her list of tasks and multiplies the sum with his/her load factor. Over-committed programmers should give up some of their tasks to under-committed programmers. However if the team is over-committed then the steering phase from release planning (Section 3.7.1) should be followed to correct this.

3.7.2.3 Steering

Once again the direction of the project can be changed during an iteration by allowing the following actions:

- *Implement a task*. This is where a task card is taken and its contents is implemented. See Section 3.7.3 for a more detailed description.
- *Recording the progress*. Every few days a team member should go around and find out from each programmer how much time they have spent on each of their accepted tasks and how many days they have left.
- *Recovery*. If a programmer becomes over-committed he/she may request assistance using the following options:
  1. Request that the scope of his tasks be reduced;
  2. Ask the customer to reduce the scope of certain stories;
  3. Discard non-vital tasks;
  4. Ask for more or better assistance from other members;
  5. Ask the customer to postpone certain stories to later iterations.
- *Story verification*. The functional tests, as specified by the customer, are run to verify that the story has been correctly implemented.

3.7.3 Implementation

A programmer takes one of his/her tasks and finds a partner with whom to pair program. This may be someone who is more familiar with a certain part of the possible solution or technology to be used.

The pair will start off by discussing the task and then write test cases for the task. As questions regarding the task arises they should discuss them with the on-site customer. As previously stated, writing the test cases first helps the programmers with building a better understanding of the requirements for the task.

When all the necessary test cases have been written they proceed to write the implementation. The implementation is then verified by running the test cases.

The pair may refactor the code base as needed during their implementation of the task.

When they are satisfied with the implementation and all the test cases runs at 100% they proceed to *integrate* their code into the system.

From the above one can see that XP is actually a highly disciplined process [Beck, 2000; Cockburn, 2002a]. This may come as something of a surprise to those who regard XP as too informal or undisciplined in regard to processes that are emphasised by more ‘traditional’ methods (such as production of documentation). The testimony of Equinox is that departure from the discipline has a negative impact [Equinox Interview, 2002].

In the next section, other criticisms raised against XP are briefly considered.

3.8 Common concerns about XP

Not surprisingly, because of the fairly radical break that has been made with 'traditional' development methodologies various questions have been raised regarding the practices used by XP. Several of these issues are discussed in this section.

3.8.1 On-Site customer

Managers are inclined to argue that a customer is too valuable to the company in doing his job for the person to simply be re-assigned to the development team as a customer representative.

This means that a trade-off has to be made by the client between acquiring a working system earlier on the one hand; and having the normal output of the customer that would be reassigned to the development team on the other [Beck, 2000]. Beck states that "If having the system doesn't bring more value to the business than having one more person working, perhaps the system shouldn't be built" [Beck, 2000].

Another statement made by XP practitioners is:

> This may seem like a lot of the customer's time at first but we should remember that the customer's time is spared initially by not requiring a detailed requirement specification and saved later by not delivering an uncooperative system.

Another compromising view is that although a customer sits with the development team, the programmers will not be able to ask 40-hours (Section 3.5.10) worth of questions; thus allowing him/her to still do some of his/her work while sitting with the team.

The survey by Rumpe & Schöder (reported in [Rumpe and Schöder, 2002]) indicates that XP practitioners regard the on-site customer as:

- the XP practice whose neglect is likely to put the project at highest risk;
- the second most difficult XP practice to implement, only preceded by Metaphor;
- the second least used practice.

Although the first point in combination with the last point might suggest that XP does not work in practice, it seems the project teams surveyed have overcome this difficulty by using substituting techniques such as part-time on-site customers to try and fill the gap. With these compromises they reported a zero failure for projects.

The above survey seems to confirm that requiring an on-site customer as part of an XP process is a very controversial area. Its implementation and acceptance will require a widespread change of culture in industry requiring a paradigm shift for managers and software acquirers.

3.8.2 Lack of scalability

XP's dependence on verbal communication limits the number of people that can effectively partake in a team. XP also requires that the team is centrally located (same room). This is in direct contradiction to the trend of de-centralised development (diverse geographical sites in multiple time zones). Note, however, that there are indications that this trend is being reversed [Paulson, 2001].

Although Beck specifies a maximum of 20 team members, successful projects with up to 50 members have been reported [Taber and Fowler, 2000] and the survey [Rumpe and Schöder, 2002] reported that 4.4% of respondents projects consists of more than 40 members.

The above reports suggest that while XP can indeed scale up, further study is needed to determine critical success factors in doing so successfully.

3.8.3 Changing requirements

Although big upfront design methodologies have mechanisms to support changing the requirements, they tend to discourage changes. Changes to requirements also tend to have high costs involved in these methodologies due to the large effort devoted to requirement solicitation, analysis and design at the start of the project. The reasoning behind this approach is that the authors assume that the requirements are mostly fixed and determinable at the start of the project; thus changes are unlikely to occur [Beck, 2000; Cockburn, 2002a].

This view is valid for well known domains that have been implemented before. These domains tend to be well defined and static, for example accounting systems.

However for domains where the business needs are dynamic and ever changing this view point is flawed. Web based systems are an example [Reifer, 2002; Equinox Interview, 2002; Rumpe and Schöder, 2002]. The same problem occurs for example when developing software and hardware concurrently in embedded products, due to the fact that the hardware specifications are not fixed until the hardware development is finished [Grenning, 2002; Balbes and Button, 2002].

3.8.4 Pair programming

This is a controversial practice for all software development processes, even between agile practitioners.

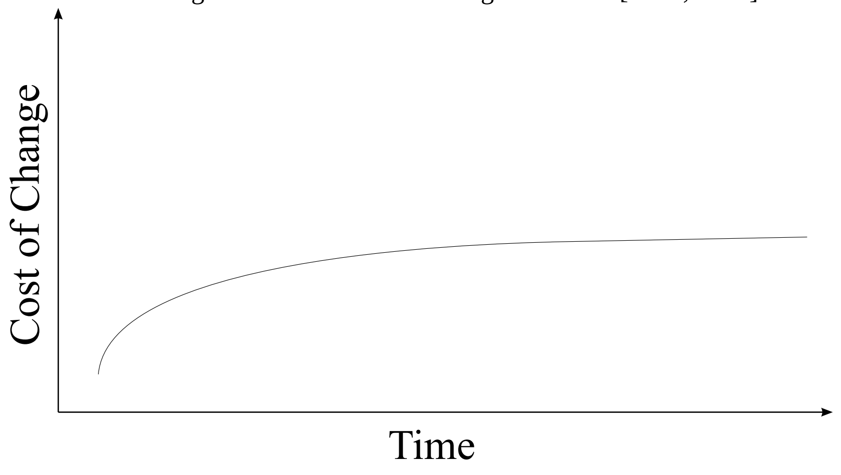
There are two main objections commonly raised against pair programming:

- **Using twice the resources to execute a task is wasteful** - a manager tends to be reluctant to assign two programmers to solve a problem that a single programmer seems capable of completing.

- **Programmers are not sociable personalities** - Programmers are stereotypically regarded as possessing weak communication and social skills, thus preferring to work on their own. This is only a half truth; programmers prefer communicating and interacting with their peers [DeMarco and Lister, 1987; Beck, 2000; Cockburn, 2002a].

Although both these claims may have some merit, they should be balanced against the contrary evidence of research done on the effectiveness of pair programming. These studies are described in [Williams et al., 2000]. In the study, students were split into a pair-programming group and individual programming group and four consecutive assignments were given to them. The result was that the pairs produced more correct solutions than their individual counterparts and as they progressed from assignment to assignment the failure rates decreased for the pairs whereas the individuals showed no pattern. In terms of time, the pairs finished their assignments 40% to 50% faster. The study states that similar results were found between the student and industrial experiments.

Figure 3.2: XP's cost of change over time [Beck, 2000].



The author's personal experience in pair programming, indicated that logic and semantic mistakes are reduced, implementation time is shortened and debugging becomes easier and less time consuming.

3.8.5 Cost of Change over Time

Beck [2000] suggests that using XP produces a flattened cost of change over time curve that is log(n) (see Figure 3.2). Beck cites an experience where after the system had been in production for two years, it took them thirty minutes to make a logic change to it (see [Beck, 2000]), indicating a flattened cost of change because if the change would have been made two year earlier it would have cost the same.

Cockburn provides a mathematical prove [Cockburn, 2000], based on experience scenarios, that XP still conform to the traditional exponential cost curve. The difference according to Cockburn lies in the scale of the curves from big upfront design methodologies and that of XP. Thus in essence when XP reaches it's production phase other methodologies may still be in their requirement specification phase, the time scale between the approaches are thus different.

Both Beck's and Cockburn's curve representations are based on their own experiences and not on generalised scientific research. As a result the alleged flattening of cost of change over time proclaimed by XP cannot be confirmed.

3.8.6 Lack of documentation

The agile manifesto states "Working software over comprehensive documentation" as the first value [Fowler and Highsmith, 2001; Agile Manifesto URL]. XP does not generate any formal documentation accept when the customer request it as one of the requirements. When this happens the documentation becomes a story and is treated as such allowing the customer to prioritise its generation. This arrangement means that documentation does not form part of the process but part of the requirement specification of the system.

Big upfront specification and design is the main culprit for generating a lot of formal documentation in software development. Due to XP's nature of small incremental development (Section 3.5.2) and having on-site customers (Section 3.5.11) the need for formal big upfront design is removed and the documentation needed in the process reduced to only story and task cards.

For the question 'What should one put into documentation?' from a developers point of view in light of the process, Cockburn [2002a] provides the following answer: "That which helps the next programmer build an adequate theory of the program". From this we see that XP uses the intimate communication in the team to accomplish this goal. A development manager on one of Becks' XP projects made the remark "anyone who couldn't find the rest of what they needed to know from the code and the tests had no business touching the code" [Beck, 2000].

The only concern left is, what documentation will be available when the project is terminated to use if the project is revived later on? If this situation occurs the last requirement stated by the customer and executed by the development team should be to write a final report to augment the test cases and code.

3.9 Summary

XP has brought some controversial issues to light and re-emphasised some practices in the software development community. These practices were mentioned and the controversies discussed. Being the agile methodology with the most publicity has made XP the reference methodology for debate between big upfront design (traditional) processes and agile processes. The adoption of, and experimentation with XP, has increased over the last few years, which in turn has fuelled the debate over whether XP can work. Initial surveys seem to indicate that XP produces a zero failure rate, keeping in mind that XP should only be used on projects that are suitable in regard of the advantages that XP claims to provide. These restrictions on applicability to projects have been defined in [Beck, 2000], with some restrictions being successfully stretched on some recorded projects.

Academic interest has grown over the past years but there is still a lack of sufficient scientific studies to validate XP on a scientific bases. This chapter has tried to provide a critical overview on XP. However, there are still issues that need to be researched in more depth, including the range of applicability and how XP influences the cost of change curve.

Chapter 4

A Critical Overview of the Crystal Family of Methodologies

4.1 Introduction

Based on his methodology-per-project (Section 4.2.8) belief, Cockburn defined a framework (Figure 4.1) to classify projects and methodologies. This classification is intended to enable project managers to select a methodology that is appropriate for the project's specific characteristics. These characteristics include scale and criticality. From experience Cockburn has designed methodologies that fit into his framework and that have been used successfully on industry projects. These methodologies, collectively known as the Crystal family of methodologies, are based on a common set of principles and beliefs (Section 4.2 and 4.3.1) and are summarised in Section 4.3.

4.2 A Philosophy of Software Development

Cockburn describes his philosophy of software development in [Cockburn, 2002a]. This section will provide a summary of his ideas. Cockburn bases his outlook on software development on five pillars. These pillars are: the learning process; the cooperative game metaphor; the human factor; communication; and the goals of software development. The aforementioned pillars provide the bases for the principles defined and used by Cockburn's methodologies.

How he defines the learning process is summarised in Section 4.2.1. In Section 4.2.2 the notion of software development being a cooperative game is explored. The influence that people and communication have on the development process is described in sections 4.2.3 and 4.2.4 respectively, followed by his views of the goal of software development in Section 4.2.5. Cockburn and Highsmith's ideas on what principles should be used when designing a methodology are mentioned in Section 4.2.6 and the need for agile and self-adapting characteristics is explained in Section 4.2.7.

4.2.1 Three Levels of Listening

Cockburn defines three levels of listening to portray the way in which a methodology should be defined and taught [Cockburn, 2002a]. The three levels/stages of listening are *following*, *detaching* and *fluent*. Each of these levels are briefly described in the following subsections. It should be noted that the levels apply to both listening and learning, and are treated as such throughout this chapter.

4.2.1.1 Following (Level 1)

This is the first stage of listening and is applicable to a new comer with no or little knowledge or skill in the applicable domain. To learn a new skill the beginner needs to take an example illustrating the skill and duplicate this example. The result is thus that when the beginner has duplicated the example he/she can determine whether it works and how the process is carried out.

An example of level 1 learning is a first year university student learning to program for the first time. The lecturer will explain the condition statement using a simple numerical comparison. The student is then asked to implement a conditional statement in a program using different values for comparisons.

In essence people in this stage require detailed step-by-step instructions on how to accomplish a task.

In a software development context this is where the developer uses a manual that describes a specific process for building a software system in detail, including templates and examples of all the artifacts that are expected to be produced.

4.2.1.2 Detaching (Level 2)

During this stage a person will start to discover limitations to the processes learnt. Alternatives to accomplishing the task with more sufficiency are required and the person has gained enough skill to be able to learn these alternatives. In the example of learning to program, a student will learn that nested if statements may be replaced by switch statements.

For software development this is the same as a developer learning different methodologies/processes for developing software.

4.2.1.3 Fluent (Level 3)

By this stage a person has gained enough knowledge to be able to produce the desired end product without having to follow a specific process. When asked which process the person is following she may not even be able to confirm which specific process is being used because an *ad hoc* one is used. This is the stage in a software developer's life where she uses elements from a diverse range of processes and even self derived ones to accomplish a task according to the current need/situation.

It is not difficult to recognise these stages in many parts of life. Another example where these levels of learning are evident is learning to do ballroom dancing. First the dancer is taught the basic steps of a dance and she practices these steps until she is proficient enough to move on to learn the advance steps with a set sequence. Only after mastering the sequence of steps is the dancer 'allowed' to dance her own sequence.

This breakdown of the different levels of learning bring some insight into how the skill level of a developer may influence development and implementation of a methodology.

After reading Section 4.3 on how the family of methodologies function, one should notice that the methodologies are introduced on level 3 (fluent) of learning. Doing this makes the methodology more agile and encourages the notion of a methodology-per-project.

4.2.2 Software Development as a Cooperative Game of Invention and Communication

> "[A]lthough programming is a solitary, inspiration-based, logical activity, it is also a group engineering activity. It is paradoxical, because it is not the case, and at the same time it is very much the case that software development is:

- Mathematical, as C. A. R. Hoare has often said
- Engineering, as Bertrand Meyer has often said
- A craft, as many programmers say
- A mystical act of creation, as some programmers claim" [Cockburn, 2002a]

The above is a compromise view of what software development is. Every developer has his/her own view on what software development is and against which other profession it may be compared.
The central characteristics of software development will be discussed in this section with a focus on Cockburn's interpretation.

4.2.2.1 Engineering

Does programming involve processes that are similar to engineering? Engineering usually follows a fixed process which starts of with a requirement specification phase followed by an analysis phase, a design phase and then a building phase. Thus, a kind of static process leads to a fixed and final physical product. Examples include the engineering of bridges, roads, washing machines, etc. But software is not the same. It is usually dynamic in the sense that it evolves and grows iteratively. This statement is true for new systems whose requirements are not fully known beforehand. Thus, for example, the user requirements of a website tends to evolve incrementally as the site is developed and as the site's 'life-time' progress. But for known domains where the requirements are well established by similar projects that have been done before, the use of engineering principles is feasible. For example, accounting software has been produced many times before and is based on a centuries old practice.

However not withstanding the foregoing, engineering practices should indeed be used in software development; but used "in the small". For example a feature that forms part of a system is derived from a requirement of the system. The requirement should be analysed and some informal design thinking should take place before implementing the feature. This should be done informally and in a small scale to keep the project agile.

4.2.2.2 Innovation

When software is to be developed, it is because a problem is to be resolved. Clearly some innovation is required to solve the problem for if nothing new was required then there would not have been a problem in the first place. This statement thus identifies one of the characteristics of software development: the requirement for innovative, creative thinking to provide a software solution to some problem.

4.2.2.3 Communication

Programmers have been stereotyped as being loners who are separated from society, preferring not to socialise with other people. But is this stereotyping accurate? Whatever the truth about the general case, most programmers would attest that they communicate well with their peers, exchanging information on development quite easily.

As with any project, in particular projects requiring highly intellectual input, a high degree of communication is needed. This high level of communication is required to coordinate and assist members within the team.

4.2.2.4 Cooperative game

Cockburn characterises software development as a group *game*, where a group game is defined as goal seeking, finite and cooperative. It is *goal seeking* due to the fact that development needs to produce an artifact which, in this context means a software system. The game is *finite* due to the fact that at some point the project will either be suspended or it will reach completion when the goal is reached. The game is *cooperative* because all the parties taking part in the project work together to reach the goal.

These characteristics as being central to software development, clearly influence the methodologies that Cockburn propose, as will be seen in Section 4.3.

4.2.3 People Centric

In the past few years there has been a growing realisation that people are the most valuable resource that a business has. This realisation has had a dramatic impact on the management of workers by organisations, leading to dedicated departments to manage and enhance the potential employees. The value of people to software development organisations has been expressed in for example [DeMarco and Lister, 1987]. An associated idea is that the more capable the developers, the less developers are needed to accomplish a task. Cockburn cites a project where it was initially estimate that 6 good programmers would be needed. This was not possible so they had to constitute a team consisting instead of 24 mixed skill programmers [Cockburn, 2002a].

All the above leads to the conclusion that any process that makes use of human resources needs to take the human characteristics into account and be able to adapt to and support these characteristics. The more supportive the process is to human nature and needs the more satisfied the people will be which in turn may lead to higher productivity. The characteristics of humans are too diverse and as such, a complete list is beyond the scope of this dissertation. However some examples that are relevant to the discussion are presented in the following paragraphs.

One characteristic of human beings is that they are spontaneous and their behaviour varies from day to day even when faced with similar situations. One day a developer will be able to produce 400 lines of code for example and the next only 50 due, for example to changes in emotional state or to being physically indisposed.

A second characteristic is that individuals also play other "games" in parallel with the cooperative game of software development. These include the career game and the game of life. People will normally place higher priority on their careers and personal lives than on a software project. Where these priorities conflict it would be natural for a person to harm the project rather than sacrifice a his/her career or personal life. This situation needs to be kept in mind by managers as well as by methodology designers.

Another factor that influences a person is the motivation for working. Most organisations base their motivation on some reward system, usually monetary, but this may only be a short term solution. The long term reward for an individual lies in pride-in-work, pride-in-accomplishment and pride-in-contribution. The detailed issues regarding reward systems are beyond the scope of this dissertation and may be found in many business management literature sources as well as in [Cockburn, 2002a; DeMarco and Lister, 1987].

From the foregoing, one would agree that humans cannot be treated as a simple constant resource associated with a project. Management needs to understand the characteristics of the human resource and to realise that people are the most important factor in software development. This realisation should promote a people-centric software development approach.

4.2.4 Communication - Cooperative Teams

In a situation where the progress of a project depends on acquiring information, it is of utmost importance to shorten the communication delay. For example, if person A has information that person B needs, then the progress of the project depends firstly on how long it takes person B to discover that person A has some valuable input and secondly, on how much energy is required to exchange the knowledge from person A to person B.

Four different scenarios below illustrate the cost (both time and monetary) involved in discovering that useful information may be transfered. They are now provided in ascending order of cost. Assume that the cost for a minute a programmer is a arbitrary amount, say N.

Scenario 1: Persons A and B are pair-programming, enabling communication to be instantaneous. Thus the cost is a fraction of two times the cost of a programmer per minute, thus less than 2N.

Scenario 2: Persons A and B are located in the same open plan room next to each other but at separate work stations. Given their close proximity, they may notice when one or the other is searching for information or they may ask one another. The time laps may be somewhere between 1 to 10 minutes. Thus costing between approximately N to 10N.

Scenario 3: Persons A and B are located on the same floor but in separate offices. There is no chance for person A to discover person B's needs and voluntarily contribute information. Instead person B will need walk to person A's office, establish A's availability and only then ask the question to find out if person A can contribute. It is quite likely that person B will first try to solve the problem independently before approaching person A in this situation, thus adding say another 10 to 30 minutes to the task. This causes a cost of approximately 12 to 35 minutes with a monetary value of between 12N to 35N.

Scenario 4: Persons A and B are located on different floors or in different buildings. In this situation there is an increased possibility of failed attempts to communicate because of the increased chance that the required person may be unavailable. For example person B walks to person A's office to find that A has just left for a meeting or a coffee break, B therefore needs to repeat the attempt to communicate later on. An estimated 10 to 70 minutes or even more might be needed to eventually establish communication. The cost has increased once again and the calculations of the monetary cost becomes very variable and perhaps excessively high.

The use of telephones might ameliorate these costs slightly, although many people are familiar with the frustration of engaged or unanswered calls. In some cases, short rapid email might cut down on costs. However email can also be excessively time-consuming. Moving away from face-to-face communication introduces the quality of communication factor. Research has shown that 55% of communication consists of body language [Mehrabian, 1981].

When looking at the above scenarios the cost might not at first seem that high. However, considering that these scenarios will occur multiple times during a project, the accumulative impact of the cost differences may result in huge time and monetary waste.

The above illustrates how important effective communication is for a project to execute efficiently and in within budget constraints.

There has been much research on how to increase the efficiency of communication as well as on techniques and tools to enable this [Cockburn, 2002a; DeMarco and Lister, 1987]. A full discussion on communication in relation to cooperating teams is beyond the scope of this dissertation. Nevertheless, the matter is considered important by Cockburn and plays an integral role in his proposals as discussed in Section 4.3.

4.2.5 Goals of Software Development

Cockburn considers that the primary goal of software development should be to deliver working software products that satisfy the users' needs. A secondary goal is to leave sufficient 'markers' behind for others to follow, for them to get a understanding of how the system works and for them to be able to extend it.

This clear and explicit goal hierarchy is apparent in the methodologies that he proposes and that are discussed in Section 4.3.

4.2.6 Methodology Concepts and Design Principles

Cockburn [2002a] describes a methodology in the software development context as

> "... everything you regularly do to get your software out. It includes who you hire, what you hire them for, how they work together, what they produce, and how they share. It is the combined job descriptions, procedures, and conventions of everyone on your team. It is the product of your particular ecosystem and is therefore a unique construction of your organization."

Some terms that he uses to describes a methodology are listed below:

Control elements refer to any deliverable, standard, activity or technique that is used or required by a methodology.

Methodology size indicates the number of control elements in the methodology.

Ceremony refers to the amount of precision required from activities. The detail given and the method used to write a use-case, for example, may range from white-board designs to a five page template that has to be completed.

The *Methodology weight* is the product of the methodology size and the ceremony. Thus it is the number of control elements multiplied by the ceremony needed for each element. The result is conceptual but still relevant.

Project size is the number of people who participate in the project and who need to be coordinated.

System criticality describes the degree to which an undetected defect may have an influence or may have repercussions.

Cockburn [2002a] defines seven principles that are useful when designing or evaluating methodologies. They are briefly presented below.

1. "Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information". Sections 4.2.2 and 4.2.4 have suggested that this principle has substance.

2. "Excess methodology weight is costly". By requiring the project team to create more artifacts than is needed to develop software (which is seen as the primary goal) the methodology size increases. More ceremony to compile these artifacts is also required. Thus a higher weight is added to the methodology when such artifacts are required. These higher costs may not provide important value to the project.

3. "Larger teams need heavier methodologies". As the number of people associated with a project increases, the need for control elements also increases accordingly to meet the communication needs of the project and to manage the cooperation between the team members. This increase in communication requirements also leads to more ceremony required for the control elements that are used. Thus, once again, the weight of the methodology rises.

4. "Greater ceremony is appropriate for projects with greater criticality". As the risk of potential cost (monetary, loss of life, discomfort, etc.) grows, more ceremony is needed to help absorb the risk.

5. "Increasing feedback and communication reduces the need for intermediate deliverables". Thus, for example, a working piece of the software system should be delivered as soon as possible to illustrate the developers' understanding of the requirements that were specified by the acquirer.

6. "Discipline, skills, and understanding counter process, formality, and documentation". Software projects depends heavily on tacit knowledge possessed by the team in general and by specific individuals in the team. The knowledge is not easily transferable. It is infeasible to capture all the knowledge that exists in a project due to its tacit and wide range properties. Trying to capture the intellectual thinking that lead to **evolving** the system can only result in reduced productivity and efficiency. The goal should be to produce markers that will assist other people in building their own knowledge base on how the system was developed when they need to extend the project. Highsmith is quoted as stating that "Process is not discipline" [Cockburn, 2002a], but that discipline is a result of a person who chooses to execute a task in a specific manner consistently, whereas process is the execution of a step-by-step list of instructions.

7. "Efficiency is expendable in non-bottleneck activities". For activities that do not form bottlenecks in the development process, the emphasis on efficiency might be reduced while ensuring that activities that are bottlenecked are carried out more efficiently. This means that the project manager should have a holistic view of the process and project before trying to streamline a particular activity. An example, as discussed in [Cockburn, 2002a], would be a project that has five programmers and one database administrator (DBA). If all the programmers are feeding the DBA with database design requirements that need to be implemented, then this will result in a bottleneck at the DBA who may only manage to implement the input of, say, two programmers. In this situation, the manager may either reduce the number of programmers or else 'reduce' their efficiency by not streamlining their activities. This can be accomplished by imposing increased design and rework requirements on the programmers before conveying requests to the DBA. In essence, streamlining the process as a whole is not necessarily the sum of streamlining the individual elements.

These principles are based on experience gained from projects conducted by Cockburn and Highsmith over many years. They are offered as a list of plausible principles to consider when designing a methodology.

4.2.7 Agility and Self-Adaptation

From the previous sections it may be concluded that many factors influence a software development project and the methodology that should be followed. These factors also determine the agility of the methodology. As the ceremony and control elements increase, so the agility decreases. To try and stay as agile as possible the manager of the project should keep in mind what the goals of software developments are. As stated previously, Cockburn defines the primary goal as delivering working software and the secondary goal as preparing for the following game. For the manager this leads to the question: what is "light but sufficient" to attain these goals? The sufficiency of markers for the project should be determined by the project's specific needs. The developers and their managers need to learn to recognise when the generation of documentation passes the point of being useful to the point of being wasteful of resources. Cockburn [2002a] provides various guidelines on how to produce sufficient documentation.

Factors that may maximise agility as stated by Cockburn include:

- *Two to eight people in one room*. This enables effective communication with the lightest ceremony.
- *On-site usage experts*. This is based on the advantages associated with rapid feedback. An on-site usage expert enables the team to ask questions and get answers almost instantaneously. These experts also provide the opportunity to create a product that truly addresses the acquirer's needs.
- *One-month increments*. This is based on the idea of rapid feedback and its advantages. Using short increments of one to three months enables the development team to repair the process and the requirements. The one to three month range is based on feedback provided by projects surveyed by Cockburn and Highsmith.
- *Fully automated regression tests*. Cockburn states that having automated regression tests (unit and/or functional) improves the system design quality and the

programmer's quality of life. The system design quality is increased by enabling the programmers to revise code then and retest the system by a simple command execution (i.e. by a simple press of a button). The programmer's quality of life is enhanced since one is able to confirm that the system is working according to one's current understanding by simply running the automated tests.

These factors are similar to the practices proclaimed by XP, with a slight deviation in magnitude.

A self adapting methodology is one which explicitly advocates that the implementors adapt the processes and practices of the methodology if and when circumstances change as the project progress.

Cockburn believes that software methodologies should be self adapting. He argues that the result of an activity may not be the same from project to project, due to the uniqueness of each project. An activity that may have increased the efficiency on one project, may decrease the efficiency on another project. In his view this alleged unpredictable nature of methodology implementation requires the team to adjust the methodology, processes and activities throughout the project's life time. Methodology reviews should be conducted at different stages during the project. Suggested points are at the start of the project; in the middle of the first increment; after each increment and/ or in the middle of increments. The team should determine which practices worked well and which did not. They should also consider practices that have not been tried before and which might work.

4.2.8 Methodology-per-project

The fact that each project is unique with regard to the inputs provided and the fact that the output required is never the same, suggests that the methodology used for executing each project needs to be specifically tailored for that project. This notion of a methodology-per-project is what has inspired the Crystal Family of methodologies.

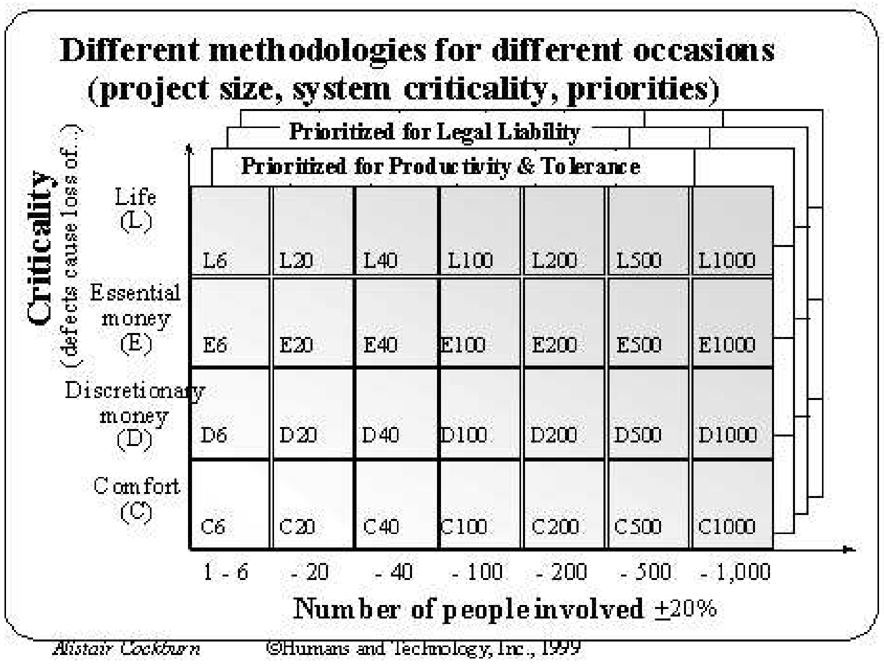
4.3 The Crystal Family

Cockburn constructed a *family* of methodologies and gave them the 'family name' Crystal. These methodologies are meant to be samples of methodologies that need to be adjusted to each individual project. The methodologies are based on Cockburn's philosophy of software development (see Section 4.2) in which the focus lies on development being people-centric, communication-rich and adaptable.

The methodologies in the family are arranged in a two dimensional matrix (refer to Figure 4.1). The x-axis or horizontal dimension measures the project size (or number of people involved), and the y-axis or vertical dimension measures the system criticality (see Section 4.2.6). The indexed criticality values are: Life (loss of life is possible if a problem occurs in the system); Essential money (loss of essential money might cause bankruptcy); Discretionary money (some money might be lost due to faults in the system); Comfort (merely causes discomfort for the user if a problem occurs). The project size starts at 1 to 6 people and goes on to 7 to 20, 21 to 40 etc. Cells of the matrix are thus codified as, for example, E20, indicating "essential money/ 7 to 20 people; D500 indicating "discretionary money/ 201 to 500 people; etc.

In addition the matrix is colour-coded. The colours darkens as one moves from right to left and bottom to top. They range from clear up to violet. The colours are intended

Figure 4.1: Cockburn's Methodology Matrix [Cockburn, 2002a].


Different methodologies for different occasions
(project size, system criticality, priorities)
Prioritized for Legal Liability
Prioritized for Productivity & Tolerance
Criticality
(defects cause loss of...)
Life (L)
Essential money (E)
Discretionary money (D)
Comfort (C)
L6 L20 L40 L100 L200 L500 L1000
E6 E20 E40 E100 E200 E500 E1000
D6 D20 D40 D100 D200 D500 D1000
C6 C20 C40 C100 C200 C500 C1000
1 - 6 - 20 - 40 - 100 - 200 - 500 - 1,000
Number of people involved ±20%
Alistair Cockburn
©Humans and Technology, Inc., 1999

to suggest the increase in "hardness" of projects as they require the co-ordination of more people and compliance with increasing criticality requirements. Cockburn, somewhat romantically, associates various areas of the matrix with different crystals, such as clear quartz for C6 and topaz for D20. This is the origin of the term "Crystal Family".

Using the matrix one is able to select the appropriate methodology for a specific project.

Thus, for example, consider NASA's next generation shuttle guidance system as a project and suppose it has five developers assigned to it. This is a life critical system (L on the y-axis) with less than 6 people, resulting in an L6 project. This classification is a 'diamond-style' project requiring Crystal Clear practices with more emphasis on 'ceremony' , such as documentation, validation etc.

Section 4.3.1 states the common features for all of the members of the Crystal family. The remaining sections will be devoted to discussions on three of the sample methodologies that have been developed and used in practice. The first one is Crystal Clear (Section 4.3.2) for a D6 category project. The second is Crystal Orange (Section 4.3.3) a D40 instance. This is followed by the specially developed Crystal Orange Web (Section 4.3.4) . A full description of each member of the Crystal family is to be found in [Cockburn, 2002a]. Each example will be discussed under the subsections of: roles required, policy standards needed, deliverables and tools suggestions.

4.3.1 Family Commonalities

Members of the Crystal family share a common set of values and principles as well as the practice of on-the-fly methodology tuning (i.e. they are self adapting). The common values shared by the family members are the following.

- They are people and communication oriented. This means that tools, processes and artifacts are supportive of the persons, as opposed to them dictating the development efforts. This is also the first value stated in the *Agile Manifesto* [Fowler and Highsmith, 2001; Agile Manifesto URL].
- They are highly versatile. This means that the project team is not restricted to working with a specific process but may select parts from different processes, such as XP.

The Crystal family members are also based on the seven principles discussed in section 4.2.6.

Cockburn also defines two rules that need to be adhered to when creating a member methodology. Firstly, incremental development should be used. Cockburn suggests one- to three-month increments. Secondly, retrospective meetings must be held before and after each iteration, and mid-iteration meetings are also suggested, though not prescribed.

4.3.2 Crystal Clear

This is an example methodology for a D6 (see Figure 4.1) type of project.

It consists of a single team comprising up to 6 members who are co-located, preferably in a single office. The system risk should be limited to essential money. The methodology is strong in communication that is rich and informal and light on deliverables, focusing on frequent deliveries.

Roles that should be assigned to separate people are: a sponsor; a senior designer-programmer; a designer-programmer and a user. Other roles that need to be assigned are a project coordinator, a business expert and a requirements gatherer.

A policy standard consisting of the following standards is mandatory:

- Software is to be incrementally delivered in cycles of two to three months.
- Progress is to be measured by the software delivered and not by the documentation produced.
- Automated testing of functionality is to take place.
- Direct user participation is to take place.
- Two viewings by user are to be held each release.
- As soon as the current task is stable for review, the following task should commence.
- Workshops are to be held at the start and middle of an increment to tune the methodology.

The above standards may be substituted by equivalent standards from other methodologies such as XP.

The deliverables as stated in [Cockburn, 2002a] are:

- Release sequence.
- Schedule of user viewings and deliveries.
- Annotated use cases of feature descriptions.
- Design sketches and notes as needed.
- Screen drafts.
- A common object model.
- Running code.
- Migration code.
- Test cases.
- User manual.

Crystal Clear only dictates that documentation should be generated for the afore mentioned deliverables, however the specifics for the format, detail level, content etc. is left for local tailoring and specification.

4.3.3 Crystal Orange

Designed for a D40 (see Figure 4.1) project, the orange D40 methodology requires more ceremony that Crystal Clear to help communications between the team members. A more thorough description of the methodology can be found in [Cockburn, 1998].

Roles that should be assigned to separate people are: a sponsor; a business expert; a usage expert; a technical facilitator; a business analyst/designer; a project manager; an architect; a design mentor; a lead designer-programmer; a designer-programmer; a UI designer; a reuse point; a writer and a tester.

The members are divided into teams to deal with: system planning; project monitoring; architecture; technology; functions; infrastructure and external testing.

Artifacts to produce, as stated in [Cockburn, 2002a] include:

- Requirements document.
- Release sequence.
- Schedule.
- Status reports.
- UI design document.
- Common object model.
- Inter-team specs.
- User manual.
- Source code.
- Test cases.

- Migration code.

The policy standards are the same as Crystal Clear (Section 4.3.2)

Crystal Orange has been successfully used in industry projects including the so-called Project Winifred. This was a D40 Smalltalk project with fixed-time and fixed price constraints. A full discussion on Project Winifred and Crystal Organge's implementation can be found in [Cockburn, 1998].

4.3.4 Crystal Orange Web

Crystal Orange Web was defined by Cockburn for eBucks.com, a joint initiative between the two South African companies: FirstRand Group and MTN [eBucks URL]. The difference between Crystal Orange and Orange Web lies in the idea that Orange Web does not consist of a single project. Instead it addresses the type of project that is in continuous development and that consists of different initiatives that need to be merged into a single code base. When this methodology was developed, the focus of the project was on defect reduction. It thus required a different emphasis to the standard Crystal Orange methodology as will be seen in this section [Cockburn, 2002a].

The team consisted of 50 people that included executives, business people, managers, analysts, programmers, and testers. Cockburn thus classified the project as an E50 instance.

Some of the conventions/rules used are as follows:

- A fixed length development cycle of two weeks, with the option of switching to a double-length cycle (four weeks), if needed. This allows all the different teams to synchronise on the same time line. These cycles are to be followed by a company-wide (thus all the eBucks project stakeholders) post-cycle reflection workshop.

- The basic process starts of with the business owner writing a business use case and a system use case brief. These are reviewed by the business executives for acceptance as a needed feature. When accepted, the detailed use cases are created by the business analysts. This detailed use case is then used to implement the feature. The integration testers test the feature and post the changes to the group and the call centre. Bug reports for features contained in live systems are sent to the 'SWAT team' by the call centre. The 'SWAT team' is then responsible for fixing these problems.

- The prioritised initiatives that provide the most value are posted for each cycle. These initiatives are broken down into work assignments that can be completed and tested within the cycle and are assigned to the developers. The developers in turn keep a status indication of their work on white boards that are mounted outside of their offices. Each morning a short status meeting is held between the business owner and the developers. This is the only time when the business owner is allowed to ask for the current status. Between 10:00 and 12:00 the company is on 'focus time', during which no meetings are held and phones are turned off. This is merely an *encouraged* practice.

- For testing, automated unit tests are written for each class. Code is only released for integration testing after a peer review. This means that the integration tester receives the code, test cases and a guarantee, in the form of a note, from another programmer. Business users use a special language to write sample transactions for testing purposes, while the call centre posts user interaction statistics in a visible area for everyone to see, thus allowing developers to see where improvements to usability of the system are needed.

- In this situation it might be better to physically separate functional teams in order to reduce communication on unrelated projects and to increase speciality conversations.

4.4 Conclusion

As can be seen from Section 4.3, the Crystal methodologies support the agile manifesto's principles. It should also be noted that both Cockburn and Highsmith are founding members of the Agile Alliance. Since the formation of the alliance Cockburn has addressed the question of how his methodologies are classifiable as agile and how some of the other agile methodologies fit into his matrix. He has done this by producing several publications. (see Cockburn [2002a,b, 2001])

Chapter 5

A Critical Overview of Feature Driven Development (FDD)

5.1 Introduction

Feature Driven Development (FDD) came into existence during a large and complex banking project for United Overseas Bank, Singapore (1997 to 1998). The process was developed by incorporating the techniques defined by Peter Coad for object modeling and the remainder of the process was derived from the experience of Jeff De Luca, the project manager and technical architect. The bank approached De Luca after the originally contracted firm abandoned the project, declaring the project undoable after 2 years. De Luca and Coad, with the aid of 50 people, resurrected the project and produced 2000 features in 15 months [SteP10 URL; DeLuca Biography URL; Highsmith, 2002]. FDD as a formal process was first published in [Coad and De Luca, 1999] with a focus on the 'modeling in colour' approach. Stephen Palmer and the other developers of the project fine tuned the process. A more generalised and practical description of FDD may be found in [Palmer and Felsing, 2002].

Kern, one of the contributers to [Coad and De Luca, 1999] and a practitioner of FDD, represented FDD at the formation of the Agile Alliance. This resulted in FDD being recognised as one of the official agile methodologies.

In an informal poll conducted by Software Development Magazine's People & Projects Newsletter, the use of FDD was rated as the number one defined practice used for "the project management practice that you turn to most often" [SDM's P&P, 2003].

As with most other agile methodologies, FDD addresses the problem space of providing software in the context of decreasing business cycles. As stated in Chapter 1, software developers are under increasing pressure to deliver working solutions in extremely short periods of time.

FDD consist of five processes, as will be described in Section 5.2. The best practices advocated and used by FDD are examined in Section 5.3. FDD specifies roles to be allocated to team members. These roles are listed in Section 5.4. A brief summary of the "modeling in colour" technique is provided as Section 5.5. Section 8.7 concludes with remarks by the author.

5.2 The Process

[Coad and De Luca, 1999] summarises FDD as "a model-driven short-iteration process. It begins with establishing an overall model shape. Then it continues with a series of two-week 'design by feature, build by feature' iterations."

As suggested by the process' name, the primary concept underlying FDD is the use of features. A *feature* is defined as "a client-valued function that can be implemented in two weeks or less" [Coad and De Luca, 1999]. The following template as defined in [Coad and De Luca, 1999] can be used to express a system's feature:

> <action> the <result> <by|for|of|to> a(n) <object>

For example: "e-mail the registration notification to a user" is a feature, where "e-mail" corresponds to the <action>, "registration" is the <result> and "user" is the <object>.

To enable implementing a feature in less than 2 weeks or even in a matter of days or hours a feature needs to be as small as possible. To build confidence that progress is being made a feature should also be showable to clients.

In turn, business-related features that are grouped together are collectively referred to as a *feature set*. The template used for a feature set is:

> <action><-ing> a(n) <object>

An example is: "registering a user." where "register" represent the <action> and "user" the <object>.

Feature sets in turn form part of a *major feature set*, expressed as:

> <object> management

Thus, in the previous example, "user management" might be an example of a major feature set.

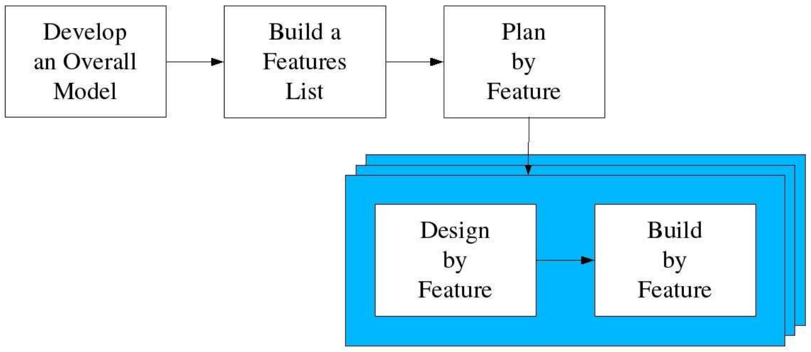
The five processes that form FDD are expressed visually in Figure 5.1. The processes are:

1. Develop an overall model.
2. Build a detailed, prioritised feature list.
3. Plan by feature.
4. Design by feature.
5. Build by feature

Processes 4 and 5 together are executed iteratively, as may be observed from Figure 5.1.

The authors of FDD intentionally summarised the processes for FDD to fit a single page for each process. This summary may be found in Appendix B. In the light of this comprehensive summary, it is unnecessary to do more than very briefly mention the different processes.

Figure 5.1: Visual representation of the five processes within FDD.



5.2.1 Process 1: Develop an Overall Model

The domain experts, in conjunction with developers, create an initial high-level ‘roadmap’ of the envisioned system. A skeleton model is then generated. This model is refined and extended using small sub-teams that are assigned portions of the model to investigate and report back on the details required.

The artifacts produced during the process are: class diagrams; informal sequence diagrams; informal features list and notes on alternative models.

The initial part of this process should only take up approximately 10% of the project’s time and another 4% on an ongoing basis.

5.2.2 Process 2: Build a Features List

Using the informal feature list compiled during process 1, the team decomposes the list into features; feature sets and major feature sets.

The identified feature sets and features are then prioritised into four categories: ‘must have’; ‘nice to have’; ‘add if we can’ or ‘future’.

The time allocations for this process is 4% initially and 1% on an ongoing basis.

5.2.3 Process 3: Plan by Feature

The sequence of how the features are to be implemented, and the time-frame for the implementation is planned out during this process. Responsibility (ownership) for the classes and features sets are assigned. The milestones to be met after each “design by feature, build by feature” iteration is defined.

2% of the project time is spend on the initial execution of this process and 2% on an ongoing basis.

5.2.4 Process 4: Design by Feature (DBF)

The next feature that needs to be implemented is analysed to identify the classes that contribute to the implementation. The class owners of the identified classes are called up to help compile the required sequence diagrams and the method specifications. The team concludes by holding a design inspection.

5.2.5 Process 5: Build by Feature (BBF)

From the DBF's output, each class owner implements the methods assigned to her including any test-cases required. After the team has inspected the code, the class owner is allowed to check-in the changes into the configuration management system.

The "design and build by feature" iterations should take up the remaining 77% of the project time. Each iteration of this "design and build by feature" should not last longer than two weeks.

5.3 Practices

As with any software development methodology, there exists a set of prescribed practices that characterise the specific methodology. In most cases it is not only the individual practices that are important, but more importantly the grouping of the practices. Not adhering to one of the prescribed practices may result in the inability to conform to the respective methodology. This is the case with FDD. The FDD authors acknowledge that some of the practices associated with FDD are not new. Rather, it is the way in which they are combined that is claimed to be new. As stated in [Palmer and Felsing, 2002], this combination results in a "whole that is greater than the sum of its parts". The rest of the section addresses each of the practices as subsections.

5.3.1 Domain Object Modeling

Through requirement solicitation from the domain experts, the developers are able to build a visual model of a possible solution to the system. This model should provide an overview of the system that can be decomposed into more detailed designs in each iteration as features are implemented.

The technique suggested by [Coad and De Luca, 1999; Palmer and Felsing, 2002] are the so-called "modeling in colour" technique. The basic idea behind the technique is to increase the visual presentation of a model by using colour to differentiate certain aspects of the model. A brief description is provided as Section 5.5.

5.3.2 Developing by Feature

As mentioned before, a feature is a *client-valued function* that expresses a requirement in a syntax that the client can understand. Following the strategy of implementing functionality on a per feature basis enables the developers to focus on implementing only the functionality that will provide value to the client. From the client's perspective this strategy provides a meaningful measure of the project's status. It also provides a means of prioritising the functionality to be implemented in the system, helping the client to steer the project more effectively.

5.3.3 Individual Class (Code) Ownership

In the object oriented world, classes acts as the primary unit of encapsulation. This means that when individual code ownership is used, the granularity of use is at the class level.

The authors of FDD cite three primary benefits for using class ownership. Firstly, as a means of providing a motivational force to developers. Having ownership of a part of the code builds pride in ones work. Secondly, it enables coherence at the class level of code. Thirdly, familiarity with the code is buildup, resulting in an expert for the inner working of certain pieces of code.

FDD also states that class ownership has better scalability than collective ownership.

However, the authors of FDD also acknowledge the potential drawbacks that individual ownership may incur. One of the drawbacks is reflected in the scenario were there are dependencies between classes. These dependencies may result in one developer having to wait for another developer to add a certain functionality before being able to complete his/her own functionality.

A second drawback is the potential loss of knowledge that may arise when one of the team members departs from the project. In some scenarios it may also result in developers gaining leveraging power over their superiors. Furthermore, the new class owner might have difficulty gaining the necessary understanding of the intricacies of the implementation.

Another drawback not explicitly mentioned, but theoretically possible and similar to the first drawback, is a scenario where there exists a primary class that multiple classes depend on. When multiple changes are needed to a single class it might result in a bottleneck for implementation. An example of this might be instances where the *Mediator* pattern [Gamma et al., 1995] occurs.

The drawbacks, mentioned above, can be managed by following the other practices prescribed by FDD. For example, making use of feature teams (see Subsections 5.3.4 and 5.4.3) introduces a dynamic team structure that enables class owners to be available to multiple feature teams and for different features. Inspections (Subsection 5.3.5) reduce the loss of knowledge in the event of a member leaving.

Nevertheless, it would seem that the above mentioned measures are not foolproof and that the likelihood of bottlenecks and loss of knowledge may still be high, depending on the unique characteristics of the project.

On the other side of the fence we have the collective code ownership paradigm. This paradigm takes the stand that any member of the development team should have access to the code base and should be able to change any part of the code as needed. This approach is pursued by methodologies such as XP and the Crystal family, surveyed in Chapters 3 and 4 respectively.

Collective ownership reduces the risks mentioned above that are associated with individual code ownership. However it may produce chaos if not appropriately managed. An example of this chaotic scenario would be if multiple developers were allowed to independently and simultaneously develop code for the same method. This would result in wasted time and resources. In turn these multiple instances of the same method may cause problems when the teams attempt to integrate their changes into the system. Thus as with class ownership, supporting practices are needed to ensure order when practicing collective code ownership. These supporting practices, as described and dictated by XP are: pair-programming; coding standards; automated unit testing and continuous integration.

5.3.4 Feature Teams

As mentioned before, a feature team is a group structure created and disbanded on a per feature basis. This highly dynamic structure supports the practice of class ownership by reducing the risk associated with it, such as bottlenecks.

Through the use of feature teams, the benefits are gained of more than one mind working together. Having a team of developers working together to produce a solution for a feature enables the selection of the most appropriate solution from a larger possible solution base.

It should be noted that feature teams should generally be as small as possible, having between three to six members per team.

5.3.5 Inspections

This is a well-known technique that has been used for the past few decades. The goals of the inspections include: error reduction; knowledge transfer; and standard conformance. The aforementioned goals in turn support an increase in quality. Inspections is a delicate technique that needs to be implemented in the correct manner to ensure that it does not produce negative effects on the developers. Pride in work may be broken by insensitive reviews. And as pride-in-work is a primary force for class ownership, the loss of pride and morale may strangle the project.

5.3.6 Regular Builds

Making builds of the complete system at regular intervals is a common agile practice that is also prescribed by FDD. This practice enables the project team to verify correct integration and to rectify any integration problems as soon as possible before they become unmanageable.

Having regular builds also provides the ability to demonstrate a working system that is as up-to-date as possible to the customer.

Added benefits include the ability to run automated functional and regression tests on a regular basis.

This practice is also advocated by other methodologies including XP (continuous integration) and within Microsoft (their nightly build practice).

5.3.7 Configuration Management

As with most software development efforts, a good configuration management (CM) system should be used to track changes in the system being developed. The type of CM system to use is dependent on the project's characteristics. A good practice suggested in [Palmer and Felsing, 2002] is to not only place the source code under CM but to also include documentation such as requirement specification and designs.

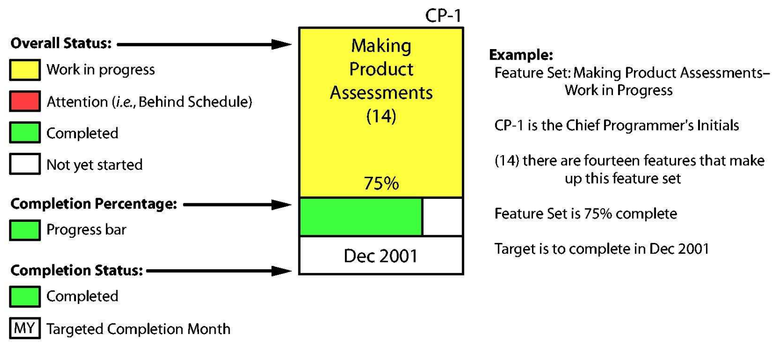
5.3.8 Reporting/Visibility of Results

The FDD authors claim that FDD's ability to provide visual status reporting is one of its strong points.

All the chief programmers meet as a group with the release manager on a weekly basis. During the meeting, the chief programmers provide verbal status reports on their assigned features. This activity enable the rest of the chief programmers to gain an understanding of the progress in other feature teams. During the meeting, the release manager records the status for capturing in the database. The status reports are provided in a visual format for easy and fast understanding. These reports may be displayed to the development team in a communal area for enhanced communication. This practice is also advocated by Cockburn among others. (See [Cockburn, 2002a]).

The visual reports used take advantage of the impact that colour provides. This may be observed from Figure 5.2. Figure 5.2 provides a summary of the recommended report that may be generated for each feature set.

Figure 5.2: Progress Report Example Coad and De Luca [1999]


Overall Status:
Work in progress
Attention (i.e., Behind Schedule)
Completed
Not yet started
Completion Percentage:
Progress bar
Completion Status:
Completed
MY
Targeted Completion Month
CP-1
Making Product Assessments (14)
75%
Dec 2001
Example:
Feature Set: Making Product Assessments– Work in Progress
CP-1 is the Chief Programmer's Initials
(14) there are fourteen features that make up this feature set
Feature Set is 75% complete
Target is to complete in Dec 2001

5.4 Roles

As with any methodology, there are some specific roles that need to be represented for the process to function properly. In FDD, these roles are the Chief Programmer, Class Owner, Feature Teams and Release Manager. The aforementioned roles are summarised in the following subsections.

5.4.1 Chief Programmer

The responsibility of the chief programmer in FDD is to orchestrate the "design-build by feature" iterations. This is accomplished through mentoring and leading-by-example. The characteristics of a chief programmer is, as the title suggests, a highly skilled programmer with experience and greater productivity than normal programmers.

The authors of FDD claim that adding chief programmers to a team using an agile approach, may increase the project speed as opposed to slowing down the project as suggested by Brooks in [Brooks, 1995]. However the authors of FDD still agree with Brooks' assertion that adding a normal programmer to a project will slow down the project.

5.4.2 Class Owner

As mentioned in Subsection 5.3.3, FDD uses a class level code ownership model as opposed to the collective code ownership found in most of the other agile methodologies. This means that responsibility for design and implementation reside with a specific member of the team, called the class owner in FDD.

5.4.3 Feature Teams

For every iteration, features are assigned to the different chief programmers. Each chief programmer analysis the feature and request the class owners who's classes are impacted by the feature to join his feature team. It should be noted that a class owner may be part of multiple feature teams at any point in time.

5.4.4 Release Manager

The release manager is responsible for compiling status reports for use by the development team, the customer and management. Reporting in FDD is discussed in section 5.3.8.

5.5 Modeling in Colour

A technique that is recommended, but not required, by the authors is the so called "modeling in colour" technique. The idea was conceived primarily by Coad during the Singapore project and based on his and Mayfield's concept of *archetypes*, first described in [Coad and Mayfield, 1992]. An archetype is similar to a stereotype, as used in UML. However the term archetype is deemed to be more descriptive than "stereotype".

Merriam Webster defines archetype as:

> "The original pattern or model of which all things of the same type are representations or copies." [M-W URL]

The archetypes that Coad and Mayfield defined are:

The moment-interval archetype. It is something that has an occurrence in time or that may have a lifespan over a time period. An example would be a order transaction. An order may start of with a initialisation time, continue to an accept status and end with a delivery date.

The role archetype. Such an archetype refers to a person, organisation, place or thing that partakes in something. An example is a sales clerk role.

The description archetype. This is a collection of values that can be reused. A book is an example of this. A book contains a title, an author and an ISBN number.

The party, place, or thing archetype. This refers to the situation in which someone or something may play multiple roles. A person may play both a clerk and a union member role.

Originally the archetypes were applied using UML's stereotype text elements. However the indications were lost in the abundance of text on the diagrams and the monotonous nature of these diagrams. In September 1997, Coad *et al.* started using different coloured Post-it^{TM}Notes to aid in the visual representation of the models [Coad and De Luca, 1999]. As UML and modeling in general is meant to be a visual aid for gaining an understanding of the system, it was just a natural step to use colour to enhance the visual representation.

The colour assignment is as follow: yellow for roles; green for things; blue for descriptions and pink for moment-intervals.

The use of colours to aid modeling has been found highly useful by the Singapore project members. Since the first usage, it has been successfully used in other projects and is being promoted by the FDD authors and others.

See [Coad and De Luca, 1999] for a more detailed description on the "modeling in colour" technique.

5.6 Conclusion

Most of the ideas promoted by FDD are not new. However, as the FDD authors noted, the packaging of the ideas is new and together it forms the solution presented here. Some of these ideas come from as far back as the 1960's and 1970's.

In regard to the role of modeling FDD has a different outlook in comparison to some of the other agile methodologies. In FDD, modeling forms an important role in the process. However it still retains agility when developing these models.

Chapter 6

Development Standards and Agile Software Development

6.1 Introduction

Having established the underlying principles and practices associated with agile software development, one may be inclined to contemplate whether agile methodologies are able to comply with established software engineering standards.

The apparent lack of documentation generated by agile methodologies may be at the root of concern when trying to address compliance. This is partly due to the concept of generating documentation as proof of compliance to the specified standard.

However, it should be noted that the agile movement is not against documentation *per se* but rather, against the overemphasis on writing documentation that provides no real value to a project's main goal of delivering effective software.

Fortunately agile methodologies are, by definition, highly adaptable and are thus able to comply with standards when required. However, it would seem that there are almost no guidelines for mapping these standards onto agile methodologies that ensure their compliance with specified standards. This chapter suggests a few such guidelines, based on an analysis of currently used ISO software standards. Since XP is perhaps the best known and most widely used agile methodology, the discussion below will focus on it as a representative of the various agile methodologies. This means that whenever there is a need to refer to some specific instance of an agile feature or practice, then the way this feature is realised in XP will be cited. Other agile methodologies such as Crystal can be adapted in a similar manner.

Section 6.2 will highlight ISO standards that are of interest to software developers. A deeper investigation of some of these standards, and guidelines for using the relevant standards are provided in Section 6.3.

6.2 Standards that are of interest

The most important ISO standards applicable to software development are ISO/IEC 12207:1995 and its replacement ISO/IEC 15288:2002, both referring to the *Software life cycle processes*. This chapter focuses on the ISO/IEC 12207:1995 standard, since this is currently used in industry. The ISO/IEC 15288:2002 standard was only approved by ISO in October 2002 and is still under consideration by local standards bodies such as the South African Bureau of Standards (SABS). At the time of writing, the standard was not generally available to the public.

Other standards that are also of interest to software development are ISO/IEC 15939:2002 (Software measurement process) and ISO/IEC 14143 (Software measurement - Functional size measurement). Although these standards are used in support of software development they are not directly relevant to the present discussion and will not be further considered here.

6.2.1 ISO/IEC 12207:1995

It should be noted that this section will rely on definitions in ISO/IEC 12207:1995 when referring to certain terms. Where needed the definition of a term will be given in a footnote. ISO/IEC 12207:1995 defines a *software product* as "The set of computer programs, procedures, and possibly associated documentation and data."; It defines a *software service* as the "Performance of activities, work, or duties connected with a software product, such as its development, maintenance, and operation.". A *system* is defined as "An integrated composite that consists of one or more of the processes, hardware, software, facilities and people, that provides a capability to satisfy a stated need or objective."

The ISO/IEC 12207:1995 standard defines a framework for software life cycle processes, spanning all stages from the initiation stage through to the retirement stage of software. The framework is partitioned into three *areas*: primary life cycle processes; supporting life cycle processes and organizational life cycle processes. Relevant *subprocesses* in each of these areas are identified and various *activities* are specified for each subprocess. The subprocesses and their associated activities are given in Tables 6.1 to 6.3.

6.2.1.1 Compliance

Compliance with the ISO 12207:1995 standard "is defined as the performance of all the processes, activities, and tasks selected from this International Standard in the Tailoring Process ... for the software project." [National Committee TC 71.1 (Information technology), 1995].

This so-called tailoring process is discussed in an annex to the standard itself and is to be used to customise ISO 12207:1995 to a specific project. The process starts off by identifying the characteristics of the project environment. These may include the team size, organisational policy and project criticality. Next, the process requires that all the affected stakeholders of the project should be consulted on the way in which the ISO 12207:1995 process should be tailored. Based on this consultation, the processes, activities and tasks that will be followed during the project should be selected. The selection should also take into consideration the processes, activities and tasks that are *not* specified in ISO 12207:1995 but which nevertheless form part of the contract. The selection activity should also document who will be responsible for each process, activity and task. Finally all the tailoring decisions should be documented, and explanations for the relevant decisions should be noted.

The foregoing implies that if an organisation prescribes conformance to ISO 12207:1995 as a requirement for trade, then that organisation holds the responsibility of specifying what will be considered as the minimum required in terms of processes, activities and tasks, in order to conform with the standard. What is to constitute compliance may

Primary Life Cycle Processes				
Acquisition subprocess	Supply subprocess	Development subprocess	Operation subprocess	Maintenance subprocess
responsibility of the *acquirer*[a]	responsibility of the *supplier*[b].	done by the *developer*[c].	done by the *operator*.	responsibility of the *maintainer.*
Activities: • initiation; • request-for-proposal [-tender] preparation; • contract preparation and update; • supplier monitoring; • acceptance and completion.	**Activities:** • initiation; • preparation of response; • contract; • planning; • execution and control; • review and evaluation; • delivery and completion.	**Activities:** • process implementation; • system requirement analysis; • system architectural design; • software requirements analysis; • software architectural design; • software detailed design; • software coding and testing; • software integration; • software qualification testing; • system integration; • system qualification testing; • software installation; • software acceptance support.	**Activities:** • process implementation; • operational testing; • system operation; • user support.	**Activities:** • process implementation; • problem and modification analysis; • modification implementation; • maintenance review/acceptance; • migration; • software retirement.

Table 6.1: Primary Life Cycle Processes

[a]"An organization that acquires or procures a system, software product or software service from a supplier." National Committee TC 71.1 (Information technology) [1995]

[b]"An organization that enters into a contract with the acquirer for the supply of a system, software product or software service under the terms of the contract." National Committee TC 71.1 (Information technology) [1995]

[c]"An organization that performs development activities (including requirements analysis, design, testing through acceptance) during the software life cycle process." National Committee TC 71.1 (Information technology) [1995]

Supporting Life Cycle Processes							
Documentation subprocess	Configuration management subprocess	Quality assurance subprocess	Verification subprocess	Validation subprocess	Joint review subprocess	Audit subprocess	Problem resolution subprocess
Specifies how information generated by the life cycle process should be recorded.	Addresses the administrative and technical procedures for the life cycle.	Ensures that the software conforms to the specifications as defined by the plans.	Used to confirm that the software products satisfy the requirements defined.	Confirms that the final system satisfy the intended use.	Used to assess the activities in a project.	Ensures conformity to the requirements, plans and contract for the system.	Used to analyse and resolving problems that are encountered during any process.
Activities: • process implementation; • design and development; • production; • maintenance.	**Activities:** • process implementation; • configuration identification; • configuration control; • configuration status accounting; • configuration evaluation; • release management and delivery.	**Activities:** • process implementation; • product assurance; • process assurance; • assurance of quality systems.	**Activities:** • process implementation; • verification.	**Activities:** • process implementation; • validation.	**Activities:** • process implementation; • project management reviews; • technical reviews.	**Activities:** • process implementation; • audit.	**Activities:** • process implementation; • problem resolution.

Table 6.2: Supporting Life Cycle Processes

Organisational Life Cycle Processes			
Management subprocess	Infrastructure subprocess	Improvement subprocess	Training subprocess
The activities that may be used by any party on any process to manage it.	Used to set and maintain the infrastructure required for the other processes.	Enables the ability of improving a software life cycle process.	Sustaining trained personnel in all parties.
Activities: • initiation and scope definition; planning; • execution and control; • review and evaluation; • closure.	**Activities:** • process implementation; • establishment of the infrastructure; • maintenance of the infrastructure.	**Activities:** • process establishment; • process assessment; • process improvement.	**Activities:** • process implementation; • training material development; • training plan implementation.

Table 6.3: Organisational Life Cycle Processes

be further refined and negotiated when defining the contract between the acquirer and supplier.

6.3 The agile angle

As previously stated, the focus here is on ISO 12207:1995 since it is the most relevant ISO standard in regard to software development. The present section motivates and provides guidelines for implementing the standard in an agile context. The discussion will focus on XP as a typical example of the agile methodologies. The following question is addressed: "Can agile methodologies be implemented or extended in such a way that they conform to ISO 12207:1995 but still retain their agile characteristics?" To the author's knowledge there is no literature that directly addresses this question. Nevertheless, the author takes the view that the question can be answered affirmatively. In support of this view, implementation guidelines are proposed that will ensure that an agile-based project conforms to ISO 12207:1995. These guidelines are derived from an analysis of the standard on the one hand, as well as from an analysis of the characteristics of the agile methodologies on the other.

The task of implementing an agile methodology in such a way that it conforms to ISO 12207:1995 can be viewed from two perspectives.

1. From the ISO standard's view the standard should first be *tailored* to meet the requirements of the project. This tailoring may involve all the parties but demands the special attention of the acquirer. The "tailored" standard should then be mapped to the development methods and processes that are used.

2. From the XP perspective, the methodology itself requires that its processes should be customised to comply with the project requirements. The same holds true for Crystal. This means the methodology inherently requires that it should be adapted to support the needs of the project. In the present context, this means that the methodology should be adapted to comply with the ISO standard. It should be noted that in both XP and Crystal, conforming to a standard is regarded as a system requirement specification in itself and is treated as such.

The discussion below refers to two of the three areas mentioned in the framework: the area dealing with primary life cycle processes (Table 6.1); and the area dealing with supporting life cycle processes (Table 6.2). Only the third of the various primary life cycle subprocess, namely the *development subprocess*, is relevant to the present discussion. Its activities are supplemented by the activities of each of eight supporting life cycle subprocesses in Table 6.2. The standard itself contains various clauses that elaborate in greater detail than provided by the tables above on what should happen in order for these various subprocesses to be realised.

Sections 1 to 4 of ISO 12207:1995 merely describes the standard and the document itself, whereas sections 5, 6 and 7 provides the prescriptions of the standard and are summarised in Tables 6.1, 6.2 and 6.3 respectively.

General comments that are broadly applicable to multiple clauses of the standard are first given below (Section 6.3.1). Then, Section 6.3.2 considers specific clauses and proposes guidelines to meet their requirements. Only clauses that relate to the development subprocess of Table 6.1, or that relate to relevant subprocesses in Table 6.2 are discussed. Finally, Section 6.3.3 addresses the matter of incremental documentation.

6.3.1 General comments

One broad approach for ensuring that an agile development team conforms to the ISO 12207:1995 standard is to assign the task of providing that assurance to one or more individuals. In essence, then, this person enables the team to produce the necessary artifacts in the manner required of the standard.

To ensure that the developers, the programmers specifically, are isolated from the burden of documentation and administrative tasks that are needed to conform to the standard, it is suggested that an organisational model similar to the one proposed by Brooks should be followed[Brooks, 1995]. This model of a so-called *Surgical Team* consists of a surgeon (an expert in performing the design); a copilot (who follows the design and knows the alternatives); an administrator (who manages all the administrative issues such as resources and legalities); an editor ('translates' the surgeon's documentation for general usage); an administrator's secretary; an editor's secretary; a program clerk(who maintains changing artifacts through version and configuration control); a tool smith (who is an expert on development tools); a tester; and a language lawyer (who is an expert on programming language usage and other specifications).

The above model is for the organisation of a development team and was proposed in the 1960s. Although it might seem inappropriate for some of the contemporary software development projects, it does suggest a few useful ideas. The particular point worth considering is the notion that the programmers should be isolated from administrative tasks and from generating documentation. In an agile project this means that the individuals assigned to ensure compliance with the selected ISO standards should generate the necessary documentation as required by these standards without burdening the developers. Thus the documentation sub-team should solicit the required information and data in a manner that is as non-intrusive as possible. As an example, when implementing this proposed model in an XP context, it is suggested that a 'standard-conformance' sub-team should be assigned as part of the development team. The members of this sub-team should be co-located in the same room as the programmers and the on-site customers in order to easily solicit information that is required for documentation. They may do so informally through normal everyday communication, and also more formally through attending XP required interactions such as regular white-board discussion sessions. Information should also be gathered from the test-cases developed and through the use of software tools that can extract relevant information directly from the source code that has been developed. Where possible, the documentation should be incorporated as part the source code to conform with the XP principles stating that the source code should be the main source of information regarding the development effort.

These personnel arrangements represent a general way to enable an agile development team to act in compliance with the ISO standards. However, also in general sense, an agile development team will quite naturally comply with a substantial part of the ISO standard's requirements, merely by virtue of following the agile methodology itself. In particular, consider the eight supporting life cycle subprocesses mentioned in Table 6.2. The next section will discuss a number of specific clauses relating to the first three subprocesses (documentation, configuration management and quality assurance) as well as to the last subprocess (problem resolution). But a cursory examination of the remaining four subprocesses in Table 6.2 will reveal that the issues which they address are, by and large, inherently built into the very substance of the agile methodology itself.

Thus, for example, *verification*[1] and *validation*[2] is inherently addressed by the agile practice of writing code specifically to meet test cases – i.e. test cases are set up prior to coding and code is not accepted until it has been verified against the test cases. Usually verification is done through unit test-cases and validation through functional test-cases.

The notion of *joint reviews* of project activity is strongly built into XP by virtue of requirements that enforce regular planning sessions, that insist on pair programming, that demand the rotation of coding tasks (such that code belongs to no particular person but is constantly reviewed by fresh pairs of developers), that ensure the availability of an on-site customer representative, etc. These kinds of arrangements ensure that continuous joint reviews of various levels of activity and at various levels of detail take place on an ongoing basis within an XP project as a natural outflow of the methodology itself.

In a broad sense, *auditing* can be regarded as a process whereby some independent agent reviews the activities of the audited party, reporting on the findings in a fairly formal fashion. The need for auditing typically arises in a context where activities are carried out in an independent and/or private fashion. But in an XP context, this is the very antithesis of the way in which code is developed. Instead, code is seen as belonging to the entire development team. Everyone has access to all code and the practice of rotating coding tasks ensures that different team members are constantly exposed to, and indeed becoming intimately familiar with, code developed by others. Of course, if the client has additional auditing requirements (e.g. stemming from concerns about possible collusion in the development team), then there is nothing to prevent arrangements amongst the various parties to have code independently audited at various times, or indeed, to periodically insert a code auditor into the development team at different stages of the development.

6.3.2 Clause specific proposals

This section provides guidelines for ensuring that agile teams adhere to specific clauses from ISO 12207:1995.

The clauses that will be examined are from Sections 5.3 and 6 of ISO 12207:1995. The former section relates to the development subprocess of the primary life cycle processes) and Section 6 deals with various supporting life cycle processes. Clauses in ISO 12207:1995 that are not specifically mentioned here mainly relate to the business and organisational activities – for example, the various activities implied in Table 6.3. They fall beyond the scope of the development activities discussed here.

The various clauses that are to be examined are shown as framed inserts. In each case the insert is followed by a discussion of the clause and suggested guidelines for bringing an agile methodology into line with the clause. Note that the clauses are examined from the development organisation's perspective – i.e. from the perspective of the suppliers of the software product. Also note that whenever appropriate, documentation in regard to the results from and outputs of each of these clauses should be produced on the basis of the model proposed in Section 6.3.1.

[1]"Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled" [National Committee TC 71.1 (Information technology), 1995]. Thus, verification checks that the system executes correctly.

[2]"Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled" [National Committee TC 71.1 (Information technology), 1995]. In essence, validation checks that the system provide the functionality required to fulfil the needs of the acquirer.

5.3.4.2 The developer shall evaluate the software requirements considering the criteria listed below. The results of the evaluations shall be documented.

a) Traceability to system requirements and system design;

b) External consistency with system requirements;

c) Internal consistency;

d) Testability;

e) Feasibility of software design;

f) Feasibility of operation and maintenance.

a) Traceability to system requirements and system design;

b) External consistency with system requirements;

c) Internal consistency;

d) Testability;

e) Feasibility of software design;

f) Feasibility of operation and maintenance.

XP explicitly takes the above into consideration during the *planning game* with the aid of *spikes* and in collaboration with the *on-site customers*. During the *planning game* the development team verifies the *user-stories* (the system requirements), ensuring not only their mutual consistency but also their overall feasibility. If the developers are unsure about a *user-story*'s characteristics, then they use a *spike* to discover in greater detail what the user-story entails. The feasibility of the software requirements may therefore also be determined through spikes.

During implementation, the practice of *pair programming* obliges the non-typing programmer to evaluate the above mentioned criteria. In fact, one of the explicit roles of the non-typing programmer is to think strategically. By its very nature, such strategic thinking will invariably relate the current coding task to the overall system's requirements in terms of the foregoing criteria.

Furthermore, testability is addressed explicitly in terms of the test-driven development principle that is applied in XP.

Any other issues to emerge during development and that relates to the above clauses (e.g. in respect of traceability, external consistency etc.) should be discussed in the team context and, if necessary written as comments in the source code for other members to read and/or to be extracted by software tools used by the documentation subteam.

5.3.5 Software architectural design. For each software item (or software configuration item, if identified), this activity consists of the following tasks:

5.3.5.1 The developer shall transform the requirements for the software item into an architecture that describes its top-level structure and identifies the software components. It shall be ensured that all the requirements for the software item are allocated to its software components and further refined to facilitate detailed design. The architecture of the software item shall be documented.

5.3.5.2 The developer shall develop and document a top-level design for the interfaces external to the software item and between the software components of the software item.

XP relies on the use of *metaphors* to facilitate the architectural visioning of the system. Using metaphors helps the developers and customers to bridge the gap of understanding between the technical jargon of the developers and the business jargon of the customer. It does this by providing a medium to describe an unknown domain using a known domain as comparison, thus using a known domain's architecture as reference. For more concrete architectural design information, the documenter should capture information from the *planning game* and from *white-board* discussions.

Given that XP propounds the idea that the source code should be the documentation, it is natural that it would be well-disposed towards the use of technologies that extract user friendly documentation from source code. Examples of such technologies include Javadoc and model extraction from code. Of course, the deployment of such technologies will usually impose certain coding standards on the development team. For example, to use Javadoc effectively means to reach some agreement about where Javadoc comments will be inserted into the code (e.g. before all public methods), what the nature of those comments should be (e.g. a statement of all preconditions in terms of public methods and public instance fields), the Javadoc parameters to be provided, etc.

Architectural documentation may be complemented by the information generated from source code whereas design documentation can be mostly derived from the source code. One of the advantages of generating documentation from the code is that the documentation is up to date and a true reflection of the system.

6.1 **Documentation process**
The Documentation Process is a process for recording information produced by a life cycle process or activity. The process contains the set of activities, which plan, design, develop, produce, edit, distribute, and maintain those documents needed by all concerned such as managers, engineers, and users of the system or software product.
List of activities. This process consists of the following activities:

a) Process implementation;

b) Design and development;

c) Production;

d) Maintenance.

a) Process implementation;

b) Design and development;

c) Production;

d) Maintenance.

The standard "only tasks the development process to invoke the documentation process"[IEEE & EIA, 1998b]. This means that each of the organisations (the acquirers; suppliers and/ or other 3rd parties) should decide on how to use and implement the documentation process. Being able to define how to implement the process enables the development organisation (supplier) to use agile methodologies and adapt them as needed.

The documentation needed is agreed upon and defined during the tailoring process. What is to be documented is therefore contracted between the acquirer and supplier. This acquirer-required documentation, together with documentation that the development organisation requires for internal purposes, jointly constitute the set of documentation that is to be generated.

6.2.3 Configuration control. This activity consists of the following task:

6.2.3.1 The following shall be performed: identification and recording of change requests; analysis and evaluation of the changes; approval or disapproval of the request; and implementation, verification, and release of the modified software item. An audit trail shall exist, whereby each modification, the reason for the modification, and authorization of the modification can be traced. Control and audit of all accesses to the controlled software items that handle safety or security critical functions shall be performed.

Nowadays there is a diverse range of configuration management software tools that enable development teams to meet the standard. They include Rational's ClearCase® and Visible's Razor. The full list of available products is too extensive to state here. Configuration control is an integral part of most development organisations and is a widely accepted practice. It should be noted that using these tools does not automatically ensure conformance to ISO 12207:1995 – the tool should be used in such a way that the outcome specifically conforms to the requirements of ISO 12207:1995.

6.3 Quality assurance process

The Quality Assurance Process is a process for providing adequate assurance that the software products and processes in the project life cycle conform to their specified requirements and adhere to their established plans. To be unbiased, quality assurance needs to have organizational freedom and authority from persons directly responsible for developing the software product or executing the process in the project. Quality assurance may be internal or external depending on whether evidence of product or process quality is demonstrated to the management of the supplier or the acquirer. Quality assurance may make use of the results of other supporting processes, such as Verification, Validation, Joint Reviews, Audits, and Problem Resolution.

List of activities. This process consists of the following activities:

a) Process implementation;

b) Product assurance;

c) Process assurance;

d) Assurance of quality systems.

a) Process implementation;

b) Product assurance;

c) Process assurance;

d) Assurance of quality systems.

Quality Assurance (QA) is built into XP through the use of functional-, acceptance- and unit tests as well as through the presence of an on-site customer. It is explicitly part of the role description of the on-site customer to ensure that the development team delivers software that meets the requirements of the acquirer by defining and carrying out acceptance tests and by doing reviews during development.

It would appear, therefore, that XP already does much to ensure quality. Although the developers on the team could hardly be regarded as unbiased (and therefore as agents for formally carrying out QA), it may sometimes be appropriate to consider the on-site customer as an appropriate “external” authority. In some contexts, the extent to which on-site customer may be considered unbiased may however be limited. For example, it would be quite natural for a congenial relationship between the on-site customer and the development team to be built up over time. In as much as it may be considered necessary to do more than merely provide for an on-site customer in order to conform to the standard, a person or sub-team could be assigned to coordinate the QA in conjunction with the on-site customer. For an even greater assurance of impartiality, a totally independent team may be assigned to verify the quality, without any reference at all to the on-site customer.

6.8 Problem resolution process

The Problem Resolution Process is a process for analyzing and resolving the problems (including non-conformances), whatever their nature or source, that are discovered during the execution of development, operation, maintenance, or other processes. The objective is to provide a timely, responsible, and documented means to ensure that all discovered problems are analyzed and resolved and trends are recognized.

List of activities. This process consists of the following activities:

a) Process implementation;

b) Problem resolution.

a) Process implementation;

b) Problem resolution.

Although XP does not have a specific process labelled “Problem Resolution", its short iterative development cycles and its high-intensity team interaction processes naturally lead to early problem detection and resolution. Furthermore, it is a natural feature of XP to support change, including changes required to resolve problems. For example, accepting changes to the requirements is part of the *steering* phase of the planning game. However, XP does not make explicit provision for documenting the problems encountered nor for documenting the changes made to resolve those problems. Such documentation, if required by the acquirer, should be referred to the documentation sub-team as proposed in Section 6.3.1.

6.3.3 Incremental Documentation

The guidelines [IEEE & EIA, 1998a,b] to ISO 12207:1995 state that when development is incremental or evolutionary the documentation generation may also be incremental or evolutionary. This statement is important to agile methodologies in general, and to XP in particular, since the system evolves over the duration of development without the big upfront design that many other methodologies use. Generating documentation incrementally also supports the idea of generating the documentation from the source code, since the source code itself is generated incrementally. Thus, the use of tools to generate documentation from source code has the benefit of ensuring that the documentation is up to date and of therefore allowing the documentation to reflect the true nature of the software.

6.4 Conclusion

The guidelines provided here are not intended to be exhaustive. Rather, they provide a starting point for agile developers who are required to comply with ISO 12207:1995.

The guidelines should of course be tested in practice and should be further customised for specific project needs.

It has been argued above that agile methodologies can indeed be adapted to ensure compliance with a standard such as ISO 12207:1995. In doing so, care should be taken to ensure that the development organisation abides by the agile manifesto principle of "working software over comprehensive documentation". The previous two statements might seem to contradict one another, but herein lies the heart of the tension that is under discussion. The agile principle of stressing the development of working software rather than huge volumes of documentation need not be a rejection of producing documentation *per se*. Rather, it represents a shift of focus to what software development really is all about – the production of software. In situations where documentation is required, this requirement then becomes a system requirement and should be treated as such. The acquirer needs to understand that extensive documentation increases resource utilisation. This translates into higher cost and perhaps slower delivery of the system. Where possible, the development team should use tools that automate the generation of documentation to reduce the resource utilisation. These tools should rely on source code as input, because source code is the most accurate representation of the software product. However, it should be realised that the effective use of such tools may well impose certain coding standards upon the development team, and these standards will have to be agreed upon and followed.

The guidelines and conclusions presented here were presented at the annual *South African Institute of Computer Scientists and Information Technologists* Conference of 2003 [Theunissen et al., 2003].

Of further note is that the *Software Engineering Institute* (SEI) is conducting research into the ability of agile methodologies to conform to the *Software Capability Maturity Model* (SW-CMM).

Paulk [2001] provides some findings regarding SW-CMM and XP. He states that XP is able to reach a level 3 grading with only minor 'key process areas' missing that are of a managerial nature. This shortcoming is due to XP being a technical process and the focus of CMM being on managerial issues. He concludes that "We can consider CMM and XP complementary" and "XP practices can be compatible with CMM practices ..."

Achieving ISO 9001 certification for an organisation that uses XP is discussed by Wright [2003]. Wright describes how his company gained ISO 9001 accreditation while following XP. He addresses the necessary practices that were needed to comply with the requirements of the standard. See [Wright, 2003] for a more detailed description.

Having established that agile methodologies are able to comply with standards, the next question to address is "how do practitioners experience developing software using agile approaches." In the next chapter a case study on an XP development team is provided to address the aforementioned question.

Chapter 7

The Equinox Case Study

7.1 Introduction

This chapter reports on an evaluation of the practical implementation of Extreme Programming (XP) in industry. The evaluation was conducted at the South African financial company Equinox Financial Solutions (Pty.) Ltd. Equinox is a LISP[1] (Linked Investment Service Provider) that provides a web-based interface for their customers on which to transact.

The goal of the evaluation was to determine how a group of South African developers with limited access to physical XP training were able to use XP as the development approach. Another goal was to gain an understanding of how the developers experience XP in practice and if they thought it to have any short comings.

Section 8.2 provides a description of the approach used to conduct the study. Another study on XP carried out by other researchers is mentioned in Section 7.3, followed by a question-by-question discussion on the responses to the study in Section 7.4. Section 7.5 highlights some general observations and the chapter ends with concluding comments in Section 8.7.

7.2 Methodology

Most members of the Equinox development team have a long association with the Computer Science Department at the University of Pretoria. They have developed a reputation of being very serious implementors of XP. These members each hold four to five year degrees and each has over ten years of experience. In recent years, the author had made their acquaintance and had confirmed their critical but dedicated pursuit of the XP methodology. As a result, they seemed to constitute an excellent test-bed of XP expertise whose experience and insights on XP could be probed.

As such, the developers at Equinox were invited to answer a questionnaire (individually) on how they experienced XP. This questionnaire is provided as Appendix C. The development team is rather small, consisting of six developers. From these four positively responded to the invitation, including the manager who did not complete the questionnaire since he felt that he did not do any programming and was therefore unable to answer the questions as such. Instead he provided his own summary of how he

[1]A financial service company that buys unit trusts at wholesale rates and resells them to the retail market.

experienced XP. The sample set was nevertheless considered adequate to fulfil the goal of the case study, namely to verify the results of Rumpe and Schröder's survey [Rumpe and Schöder, 2002] and to compare it to a South African context.

The experiences and opinions provided by the developers are described in the following chapters.

7.3 Rumpe and Schröder's Survey

In a survey conducted by Rumpe and Schröder [Rumpe and Schöder, 2002], the authors tried to gain quantitative evidence to substantiate XP's claim of being more beneficial for software development than traditional methodologies.

Even though the authors acknowledge that it was an initial and limited survey, their results suggest that "XP is superior to some of the traditional approaches at least in the domains it was used." They also cautioned that this conclusion is based on feedback from supporters of XP and should be seen in this light.

This case study has several questions that correspond to questions in the aforementioned survey. These overlaps will be addressed where appropriate in the following section as each question is discussed.

7.4 The results

This section collates the feedback of the respondents, provides some comparisons with Rumpe and Schröder's survey and suggests a number of inferences.

7.4.1 Question 1, 3 and 4 – Respondent's experience.

The methodology experience of the developers is rather diverse. It includes experience in following the waterfall approach and some of its derivatives. Two out of three respondents have RUP experience. Some experience in a military environment was also mentioned.

All the respondents have over ten years of software development experience and more than two years of following the XP approach.

7.4.2 Question 2 – Are there any bottleneck activities in the way you implement XP?

One of the respondents declared testing as a bottleneck activity in their implementation. The part of testing that results in the bottleneck is the time it takes to execute the automated tests.

For release and acceptance testing the automation of the customer tests sometimes caused problems due to unforeseen issues and/or a misalignment between the practical and envisaged system.

In later stages of the project the tests became a bottleneck due to the long time (approximately 25 minutes) they took to run. This is a problem in the case where committing changes to the system (thus integration) are only allowed after passing all the tests. Adhering to 'continuous integration' and achieving rapid development becomes difficult in such a situation.

7.4.3 Question 5 – How would you rate XP as a factor in the success of Equinox' development effort?

Two-thirds of the respondents gave unqualified support to the view that XP contributes as a major factor to the success of the development effort. However one of the respondent's opinion was: "We made a lot of mistakes and the net outcome was that the company could not bring in enough business to justify the development effort." This respondent nevertheless concluded that "XP was better than anything else".

7.4.4 Question 6 – How do you experience collective code ownership?

All the respondents gave positive feedback on the collective code ownership practice. They all believed that it is more beneficial to the development effort than any other code ownership model.

7.4.5 Question 7 – How do you experience the single room concept?

The respondents are in agreement that the single room/"war room" concept is beneficial to the development effort. They noted however, that it should be done with discipline: only project related discussion should be allowed, no telephone calls should be allowed, developers should think about a problem before raising it with other pairs, etc. It was agreed that the benefits advocated by the XP authors are indeed gained by strict adherence to this practice.

7.4.6 Question 8 – How do you experience pair-programming?

It was unanimously agreed that pair-programming is more advantageous than individual programming. It helps the developers to keep to the discipline, including test-first programming and coding standards. The respondents noted that it provides a learning aid, not only for technology but also for social interaction and communication. However, it was thought that this intense communication and social interaction can result in some drawbacks. The developers sometimes find the requirement of communicating ideas to other developers difficult and intricate. Some developers found it tiresome to work in pairs. It was noted that the 40-hour work week countered this problem to an extent. However the developers indicated that they would like to have some individual coding sessions to explore parts of the system. This individual code may then be refactored by the other members before being accepted back into the system.

Having to work on the same project and with the same people over a long period of time (in this case approximately 2 years) was also indicated as a drawback. Some developers require new challenges in their careers. This need seems to stem from the characteristics of a developer. This problem is further addressed in Section 7.5.

7.4.7 Question 9 – Pair-programming as a limiting factor in learning/experimenting.

On the issue of how pair-programming influences learning and experimenting, the respondents provided two views. Firstly, in the case where a a new member is introduced into the team, the member is able to integrate with team in a very short period of time. The new member is able to quickly gain an understanding of the system.

Secondly, individuals are able to learn from each other, which increases knowledge sharing/distribution. However, as soon as a pair needs to investigate new topics it becomes more difficult to do so as a pair. The respondents all agree that in this case each individual should tackle the experimentation individually and re-join afterwards to solve the problem as a pair. This issue is discussed further in Section 7.5.

7.4.8 Question 10 – Rate your XP-project in terms of the extent to which the listed goals were reached.

The aim of this question was to determine the extent to which the major goals of software development set out by XP was met.

Deliver software on time. The consensus was that this goal was *partially achieved*. The reasons stated was as diverse as the number of respondents. An underlying theme was that the project was viewed as an ongoing process which was to continue for an undefined time. Thus the deadlines were not 'well-defined'. This view makes it difficult to give a judgement on this goal.

Let developers have fun in their work. Two of the respondents stated that this goals was *partially achieved*. The other respondent rated it as *worse* than previously used methodologies, since the development environment imposed a communication burden. As a result of the flat structure of the team there was a lack in the leadership needed to make decisions during conflicts.

Develop high quality software. This was rated as *fully achieved* by two out of three and *partially achieved* by the other.

Late changes don't incur high cost, because one can react quickly to changes. This was also described as *fully achieved* by two out of three and the remaining responder classified it as *partially achieved*. It was noted that after a while the changes to 'the fundamentals of the system' became more difficult to handle as the system grew.

In summary, it seems that the goals set out by XP were largely achieved.

7.4.9 Question 11 – The 'level of use' of XP elements and their contribution to success of project.

The respondents were ask to rate the level of usage of each of the XP practices.

The rating options were:

1. Not at all
2. Sometimes
3. Often
4. Continuously

The same question was raised in Rumpe and Schröder survey. However they used a scale 9 – 0 where 9 indicated 'fully used' and 0 'not at all'. For comparative purposes average values in both studies were computed as a percentage of the maximum score. As may be observed from the results provided as Table 7.1, most of the usage ratings were similar. Note in particular, the relatively low level of "metaphor" usage in both cases. One should however keep in mind that the case study only represents the views of three people in relation to one project's implementation whereas the survey takes multiple projects into account. The particular characteristics of the case study probably accounts for the relatively high rating with regard to the availability of an on-site customer, and the slightly lower rating with regard to having a simple design.

Table 7.1: The level of element usage.

Elements	Average		Rumpe and Schröder	
	(Scale: 4 – 1)	%	(Scale: 9 – 0)	%
1. Planning Game	4.00	100%	6.03	67.00%
2. Short Releases	3.67	88.89%	6.86	76.22%
3. Metaphor	2.00	33.33%	3.19	35.44%
4. Simple Design	3.00	66.67%	6.98	77.56%
5. Testing	4.00	100%	7.27	80.78%
6. Refactoring	3.67	88.89%	7.77	86.33%
7. Pair Programming	3.67	88.89%	7.29	81.00%
8. Common Code Ownership	4.00	100%	8.01	89.00%
9. Continuous Integration	4.00	100%	7.56	84.00%
10. 40-Hour-Week	3.67	88.89%	7.17	79.67%
11. On-Site Customer	3.33	77.78%	4.09	45.44%
12. Coding Standards	3.33	77.78%	7.01	77.89%

As a secondary question, the respondents were ask to rate the level of contribution of each of the XP practices to the success of development.

The rating options were:

1. Negative contribution
2. No contribution
3. Helpful
4. Indispensable

Table 7.2: Each elements' contribution to the success of development.

Elements	Average	
	(Scale: 4 – 1)	%
1. Planning Game	4.00	100%
2. Short Releases	4.00	100%
3. Metaphor	3.33	83.33%
4. Simple Design	3.33	83.33%
5. Testing	4.00	100%
6. Refactoring	4.00	100%
7. Pair Programming	3.67	91.67%
8. Common Code Ownership	4.00	100%
9. Continuous Integration	4.00	100%
10. 40-Hour-Week	3.67	91.67%
11. On-Site Customer	3.00	75.00%
12. Coding Standards	4.00	100%

Rumpe and Schröder' survey had a similar question to the above, however the rating and the resulting data was found incompatible to provide a comprehensive comparison. The result from the survey is represented in a graphical fashion providing no definite values for comparison.

Even though the sample size of the case study is small compared to Rumpe and Schröder' survey, the results on how useful the developers found the practices to be seem to correlate. However the *metaphor* practice seems to be more useful to the Equinox team than to the respondents of the survey. This conclusion seems to be contradicted when considering that the respondents rated the usage of the *metaphor* practice at only 33.33% (see Table 7.1). This may be attributed to an uncertainty in regard to the correct usage of this practice. Another deviation seems to be with how the respondents experienced the *on-site customer* practice. The Equinox team found that having an on-site customer was 75% beneficial whereas Rumpe and Schröder' respondents had mixed opinions in this regard, when taking into account the written responses and the approximately 56% rating. Practices such as the planning game, short release cycles, simple design and continuous integration seem to correlate tightly between the two studies.

7.4.10 Question 12 – List the documentation that is generated.

The respondents stated that they generated documentation mostly for inter-group communication. This documentation was basically for visual communication in the team and included:

- class diagrams;
- package dependency diagrams;
- process diagrams;
- diagrams illustrating how transactions flow through the accounting system;
- instance diagrams.

The goal of the documentation was to enable the developers to understand how far they have progressed with the system and to illustrate some of the fundamental parts of the system. The diagrams where placed on the walls in the 'war-room' for easy reference.

7.4.11 Question 13 – How much value is gained from the documentation?

The respondents found the 'light-weight' documentation that they generated invaluable. The cost of generating this documentation was justifiable in terms of the value that it provided. The visual communication gained through this 'documentation' was regarded as indispensable.

7.4.12 Question 14 – How comprehensive is the documentation?

Most of the documents that was generated were high level overviews that merely provided a guide of what the system entails.

7.4.13 Question 15 – Do you capture/archive the information generated on white-boards? How?

Some of the information is captured. Initially the white-board's contents was captured using a digital camera. However these images where very seldom used. A storyboard was built as a website to capture the stories, but this was also not really used.

7.4.14 Question 16 – How do you measure your performance/productivity?

The only measurement made by the development team was the number of *stories* completed and the project *velocity* per iteration.

7.4.15 Question 17 – How would you rate your productivity when practicing XP compared to traditional processes?

All the respondents agree that their XP productivity is **better** compared to the 'traditional' processes.

7.4.16 Question 18 – Would you advocate the use of XP to others?

The respondents declared unanimously that they would advocate the use of XP to other developers. This is directly aligned with Rumpe and Schröder's results of 100% of respondents willing to actively advocate XP.

7.4.17 Question 19 – Do you have any suggestions for improving any of the XP elements?

One of the respondents felt the lack of a centralised authority to step in and make decisions when the development team was in disagreement. Although XP advocates the evolution of the design, the respondent suggested that a person with authority in the team should have a design and vision of the system in his head. This person should guide the evolution of the system (as derived from the vision) through communication facilitation.

The rest of the respondents preferred to first ensure that their current implementation process is fully aligned to the prescriptions of the defined XP discipline before attempting to change XP.

7.4.18 Question 20 – Any additional comments?

It was noted that due to XP's particular approach, a newcomer aught to freely embrace XP, rather than have it forcefully imposed on him/her. For this reason, one should rather ensure that the team is kept focused on the XP practices and not try too hard to change the individual.

The issue of how fast to grow the number of developers on a team was addressed by one of the respondents. This respondent suggested that the team should 'grow slowly', adding not more than one person per iteration. This ensures that the value of adding more developers is not 'diluted' by paying penalties due to the learning overhead.

7.5 General Comments

Some of the possible dangers of implementing XP emerged from the feedback of the respondents. As mentioned in Subsection 7.4.6, these dangers seem to stem from the intense social interaction between developers – a group that has been traditionally stereotyped as anti-social and loners. The traditional culture of software development has been to give a developer a task and send him/her off to do this task on his/her own. An opposite method is followed by XP. A pair is assigned a task and together they need to analyse the problem, approach other pairs and/or customers for more input and together come up with the solution. This high level of interaction and need for cohesion may become strenuous on the individual. As stated earlier, one of the respondents noted that the 40-hour week practice helps to manage this strain. The evidence would suggest that the manager/coach should be aware of this possible problem and try to manage it pro-actively when needed. Another suggestion is to abide by the XP office layout (open bullpen/ cave and commons) where individuals have private offices to go to when the need arises.

This intense social interaction brings another potential problem to the surface. Having to constantly work with the same people on the same project over an extended period of time may affect the morale of some of the developers. The need to address challenges is part of the characteristics of most developers. These developers long for changes in their work, a chance to learn and experience new things. When projects become monotonous, the risk of developers withdrawing from the project increases. This problem may be ameliorated if developers are able to switch between projects. However, this is only a solution for bigger companies with multiple concurrent projects. This is not a viable solution in Equinox's case .

In regard to the individual vs pair-programming learning, it is the opinion of the respondents that deeper understanding is gained when developers are able to explore topics on their own. The author agrees with this opinion and builds on it by indicating that individual learning helps sustain the idea of bringing individualistic and fresh ideas into the pair-programming effort. This not only increases the notion of two heads being better than one, but also decreases the risk of the developers becoming too attuned with one another, so much so that the quality of the solution degrades.

7.6 Conclusion

The study indicated that XP lived up to its expectations in the case of Equinox. However, as with most things in life, there exists a learning curve to follow before one feels comfortable with the new process. When implementing XP as the development methodology, one needs to understand that mistakes will be made along the way. The developers at Equinox experienced this and acknowledged that the benefits gained from XP increased as they became more disciplined and skilled in XP. Through continual improvement in the way the practices were implemented and by abiding with the rules as stated in the literature, the team was able to reap the benefits of XP.

Overall the respondents were quite positive about XP. There were some suggestions and issues raised that need to be addressed. These issues seem to relate more to managerial matters and not a methodological short coming.

Chapter 8

The Telkom Case Study

8.1 Background

To determine the applicability of agile methodologies in the telecommunication industry, an investigation of software development activities undertaken by a telecommunication company has been carried out. The company used in the case study was Telkom SA Ltd., currently the only South African fixed-line operator. As a first step, the various sections dealing with software within Telkom were identified. Then, the most prominent of these sections was selected for further study. This particular section, the so-called Telkom Development Laboratory (TDL), undertakes in-house software and hardware development to meet various telecommunication needs. One of the reasons for selecting it, is the fact that it takes part in a diverse range of telecommunication projects. The diversity relates to scale (as measured by both cost and time) as well as to breadth of technologies. Another reason for selecting the section is that some of the projects undertaken tend to be rather more complex than those in other sections within Telkom. Some of these projects may not only require the development of software, but – concurrently – also the development of interdependent hardware [1].

This chapter reports on the investigation into the software development processes undertaken by TDL. Section 8.2 explains the methodology used to conduct the investigation. The general findings that resulted from the investigation are then discussed in Section 8.3 and 8.4. Certain problems identified during the investigation and potential solutions to these problems are discussed in Section 8.5. In Section 8.6, guidelines to make RUP – the main development methodology followed by TDL – more agile in character are provided. A few concluding remarks are given in Section 8.7.

8.2 Methodology

The investigation took place in three phases.

Phase one consisted of the identification of software development initiatives within Telkom. Software development personnel from different sections were approached to

[1]Although this raises an interesting question in regard to the appropriateness of agile software practices in scenarios that rely on traditional engineering practices to develop hardware, the matter will not be further considered in this chapter. The author note in passing however, that hardware often has to be developed to meet high quality demands (perhaps even as high as zero error tolerance) and its development simultaneously subject to the constraint that, once produced, it cannot be changed.

ascertain whether they could meaningfully contribute to a study about current software development processes and policies followed in Telkom. From the feedback provided, the TDL section was deemed to be the most appropriate telecommunication-specific software development representative. There did not seem to be any compelling reason to include more than this one division in the subsequent investigation.

Phase two involved information gathering about the software development processes in force in TDL. This was done primarily by means of interviewing and observing, both of which took place *in situ*. Initially, an interview was conducted with one of the senior developers, and consisted mainly of a walk through the software development process followed by the section. Demonstrations of various software tools used to assist in the development process were observed. In addition some time was spent observing the activities that take place in the development environment. During the observations some informal discussion with team members was instantiated. The information thus gathered stimulated a number of further questions which were raised and discussed. Some pre-compiled questions were presented during the interview to the senior developer acting as liaison. The questions and their answers are discussed in Section 8.4

Phase three of the investigation then focused on analysing the information that the interviews and observations had brought to the fore, and on drawing conclusions. Sections 8.3 to 8.5 discusses the most important findings.
The interviews and observations provided a good insight into the development process followed by the TDL. This process is described in Section 8.3. In addition, Section 8.4 discusses both the various pre-formulated questions that had been designed to acquire background information about TDL and the answers received during the interviews. Finally, Subsection 8.5 discusses a key problem area experienced by TDL and suggests a number of possible remedies.

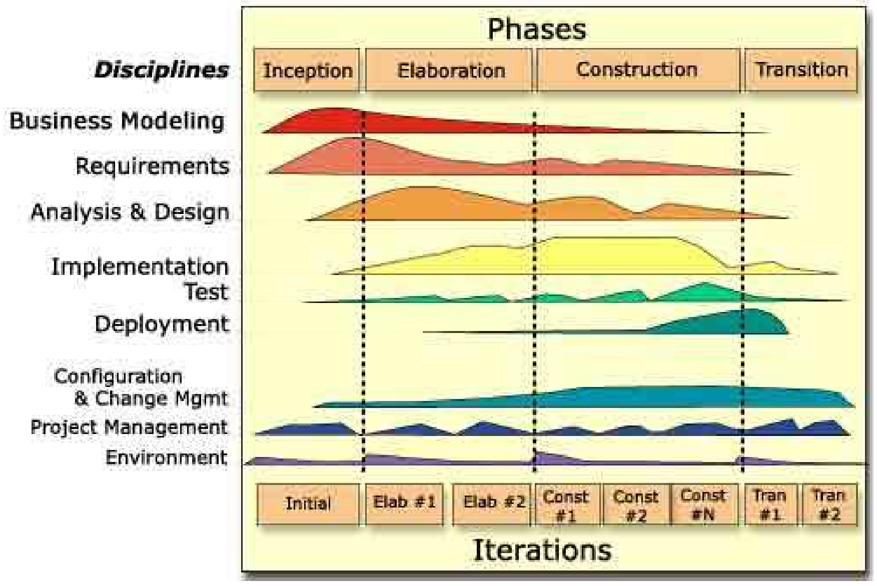
8.3 TDL Software Development Process

TDL implements a tailored version of the well-known Rational Unified Process ® (abbreviated to RUP®) of Rational Software Corporation to develop its software. Figure 8.1 gives a summary of RUP. Various so-called *disciplines* are depicted on the vertical axis, each discipline consisting of a number of activities that logically belong together. The horizontal axis represents the time line. It shows how the life-cycle phases of the process unfolds over various iterations within four phases, termed the inception, elaboration, construction and transition phases respectively.

RUP is a *framework* for software development projects. It is based on over 16 years of experience of Rational Software Corporation as well as the experience of other industry players. RUP defines more than a 100 artifacts, over 40 roles and 9 disciplines [Kruchten, 2001; Hirsch, 2002]. Because it is a framework, its implementation requires that the appropriate roles and artifacts have to be selected from the framework's available items. However, in-house practices, roles and artifacts may also be incorporated.

In order to comply with local requirements, tailoring generally needs to be carried out. Removing the *Business Modeling* discipline from certain projects undertaken by TDL is an example of such tailoring. This removal is appropriate, because most of the projects undertaken by TDL do not require an investigation of the business processes. Instead, the clients commonly approach the development team with most of the requirements already having been specified. This enables the developers to focus on restating and verifying the provided requirements during the *inception* phase.

Figure 8.1: Summary of the elements of the Rational Unified Process [Rational URL]



Much effort and time goes into the *inception* and *elaboration* phases of the project. The intention is to ensure that a high quality architecture of the system is developed. There is a strong belief that if a very good architecture is designed during the *inception* and *elaboration* phases, then the resulting architecture will facilitate the incorporation of later extensions to the system.

Thus the development teams adhere to the following process:

A team starts off by interacting with the client to compile the requirements of the system. This will typically include several planning sessions between all the stakeholders of the project. In addition, research has to be undertaken into the technologies that could be used during the project. This research may include investigations into network protocols, hardware that is to form part of the system and solutions to similar problems that are provided by third parties or by equipment manufacturers.

The requirements are modeled and compiled using use-cases. Once the planning sessions have produced sufficient information, the information is captured and modeled into Rational Rose®, the software development tool created by Rational. As soon as the requirements have been 'finalised' and stabilised, a requirement specification document is generated for the client, to be signed off and used as the basis for further development.

After the requirement specification has been produced, an architectural team (typically consisting of two to three architects) begins the task of analysing these requirements. Pairing during the design is preferred, since experience suggests that pair-development tends to increase the quality of the models produced. The use-cases are used to derive the architecture. Consistent with the RUP philosophy, an effort is made to determine the most significant use-cases, especially those that appear to carry the highest risk. These high risk uses-cases have the most influence on the architecture and are the first to be tackled during analysis. Thus, as a matter of principle, the team tries to eliminate risk as early as possible during development by implementing these high-risk use-cases first. The result of the analysis is reflected in the Analysis Model.

The model consists of high level class- and sequence diagrams.

Through the aid of Rational Rose the a Design Model is derived from the Analysis Model. The model provides a high level description of the code that will be generated during the Implementation discipline. It may include detailed class-, sequence- and state- diagrams, all of which are supported by Rational Rose. Object oriented design patterns are consider during this phase and are recommended as a basis for implementation if considered appropriate.

At the beginning of the Implementation discipline the base code of the system is generated by Rational Rose, using the models created through the previous disciplines. The primary models used are from the Design Model. The source code generated are usually Java and/or C++ depending on the project. This skeleton code is then filled in and extended to produce the required implementation. During the implementation the developers make use of the following practices:

- Pair-programming is used for writing code. Experience by the team has shown that pair-programming increases the quality of the code.
- The team members are required to abide by the coding convention specified for the TDL unit. The convention is based on the guidelines provided by Rational.
- Developers write the unit test cases themselves. These test cases are designed to validate class functionality. They are to be carried out by the testing sub-team, as described below. The testing sub-team may extend these tests as they deem necessary.
- Using Rational Rose, the developers are required to reverse engineer the Design Model so that it remains consistent with the code which they develop.

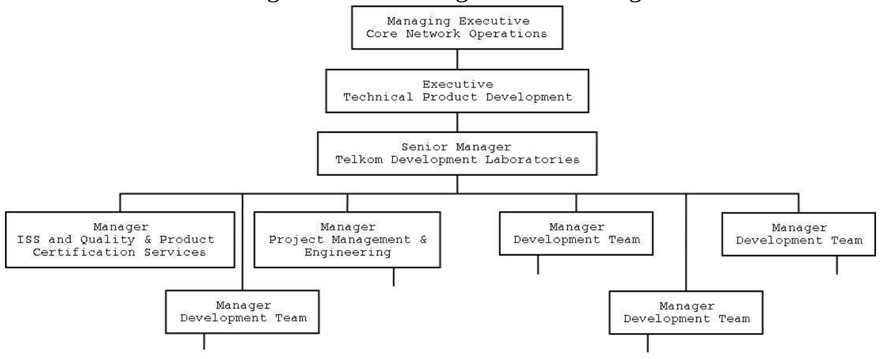
The Testing discipline starts as soon as any part of the system becomes testable. The majority of the testing is done by a testing sub-team that is dedicated to writing and running test-cases on the system. Much of the testing relies on automated testing tools provided by Rational Rose. The tests may include unit tests, integration tests, functional tests, user interface tests, business cycle tests, performance profiling tests, load testing, configuration testing and installation testing.

- In the case of unit tests, these are mostly written by the developers for each Class/ code unit.
- The business cycle tests involve the simulation of normal business activities over different periods of times. In this simulated mode, time is compressed and the objective is to validate system usage in the field.
- Performance profile testing relies on Rational Quantify® and Purify®. These tests are used to help with memory error and leak detection, code coverage and application performance analysis.

RUP is an iterative process as Figure 8.1 clearly suggests. This means that all the disciplines described above are repeatedly carried out so that each iteration of a discipline builds on the previous one in an evolutionary fashion.

As mentioned previously, a key component of the overall TDL strategy is to develop a very good architecture that will be extendable. TDL believes that this strategy enables its developers to incorporate changes with little effort. The liaison noted that experience had shown that if they had a good architecture then they would be able to incorporate most of the changes and updates to a system with ease.

Figure 8.2: TDL Organisational Diagram



8.4 Questions discussed

In this section, seventeen questions are listed. The questions were intended to assist in acquiring a better understanding of the development efforts by TDL. They were submitted to the senior developer acting as the representative of TDL during the interview. In each case, a summary of the responses to the question is provided. The questions pertain to general software development issues including managerial, process and environmental factors. The answers provided reflect mainly on how development is done in TDL.

8.4.1 What is the general organisational position and layout of TDL?

TDL forms part of the "Technical Product Development" department, which in turn is part of the "Core Network Operations" unit. TDL is subdivided into six groups. One group (ISS and Quality & Product Certification Services) is responsible for testing, calibration and validation of hardware measurement tools and thus has nothing to do with software development. A second group consists of project managers that are assigned to each project. The remaining four groups are development teams, each headed by an internal manager and assisted by one of the project managers. The representative group interviewed consisted of eleven members including senior-, junior developers and testers. Figure 8.2 provides a organisational diagram to illustrate TDL's position.

8.4.2 What is the type/range of software development projects undertaken by Telkom and TDL?

The projects are rather diverse and not easily classified under a specific heading. However, client/server applications are quite common, as well as projects that involve the development of both hardware and software which are combined into a single product. Finally, a number of projects center around the integration of multiple systems. These projects are mostly for internal use and are based on telecommunication-specific needs.

8.4.3 What software development process/policy is used?

A diverse set of software development processes are used by the various development sections in Telkom. These range across a variety of processes such as the custom-built process dubbed the “Solution Value Chain” (an end-to-end development methodology based on the waterfall model). Some sections use *ad-hoc* methods while others rely on Rapid Application Development (RAD) methodologies. The TDL section itself uses a tailored version of the Rational Unified Process (RUP) as described in Subsection 8.3 above.

8.4.4 What are the exact standards to which TDL has to comply?

The TDL teams do not need to comply with any specific standard *per se*. These teams are merely required to follow the RUP guidelines.

8.4.5 What is the success rate of projects in TDL?

From a technical and developer’s viewpoint the success ratio of projects is very high indeed, almost reaching 100%. This is attributed to the way in which the RUP process is implemented. Spending a lot of effort in early understanding of the problem domain and in defining a good architecture as soon as possible enables the team to determine the feasibility of the project early on. A project can then be halted, if necessary, before the implementation/construction stage starts.

Some projects may be described as failures from a business perspective, due to the fact that they have yielded a poor return-on-investment. The reasons for these failures are thus usually non-technical and relate more to political or market factors. Political factors dictate that the business users and/or managers decide against using the system after it has been delivered. Market factors – such as supplier policies denying the use of custom software solutions – may cause the customer to buy a solution from the hardware manufacturer instead. It should thus be noted that the business users needs to have policies in place to dictate that a thorough investigation needs to be undertaken before approaching TDL. Management should also stick to their decision of using TDL’s solution after committing them to a project.

8.4.6 Do you do project reviews (retrospective evaluations) as part of the development process?

A project debriefing is held after each project. The efficiency of practices and techniques used are evaluated. Changes are considered and noted for future projects. These changes may include tuning of practices and also configuration changes required to tools such as Rational Rose with regard to forms and reports used. The author noted the lack of process reviews during the process for example between iterations. This may lead to the loss of the advantages gain-able from fine tuning the process on a per project basis.

8.4.7 What is the current code ownership policy?

The whole team has access to the code and is able to change it as needed (i.e there is collective code ownership). This is managed by Rational Rose’s version control tools. Code sharing between teams is rare, however code sharing between projects assigned to the same team are done when feasible.

8.4.8 What are the characteristics of the physical development environment?

Each team is co-located on the same floor and within the same general area. Each member has her/his own cubical or office. The testers are grouped together enabling them to work closely together. The different teams are located in the same building.

8.4.9 To what extent do you make use of white-boards?

The use of white-boards is limited to the inception phase. Preliminary planning and modeling is done on the white-boards. Thereafter information is captured in Rational Rose.

8.4.10 Do requirements change after requirement analysis has been completed and the project is already in the implementation phase?

The team noted that requirements do sometimes change during the implementation phase (as is commonly the case in software projects). However they claim that user-interfaces changes are the most common, whereas the rest of the requirements usually are stable. This stability is attributed to the effort put into designing a comprehensive model and architecture through multiple iterations with the client.

The large number of changes to user-interface specification was regarded as a major problem in most of the projects. This problem and possible solutions are the specific theme of Section 8.5.

8.4.11 How strongly is reuse advocated in Telkom and how regularly is it carried out?

There is no policy or official strategy to promote reuse in Telkom as a whole. TDL tries to reuse code across teams and projects under its control. There is an active effort to reuse code at team level. This is done within projects as well as between projects wherever possible.

8.4.12 What is the level of interaction with customers?

The development team tries to involve the customers as much as possible in the development process. A high level of interaction and communication is encouraged. This goal is actively pursued.

8.4.13 What statistics are kept of each project?

Through the use of the Rational Rose tools, a large amount of data is collected. This data can then be reflected in a diverse range of reports that are generated by Rational's tools. These reports include defect rates; time spent on tasks and test results. The usage of the reports is at the managers own discretion.

8.4.14 What is the time-line breakdown of the different RUP phases for a typical project?

The representative responded to this question by discussing a specific project that they regarded as typical. They estimated the time breakdown as approximately between 10% to 20% for the inception phase. This phase determines if the project will be undertaken. Approximately 30% to 40% time is spent on the elaboration phases. The construction phase consists of 40% and transition the remaining 10%. From this it is clear that a significant amount of effort is spent on the development of what is considered to be an effective design and architecture. It was restated that RUP is an iterative process and that the time-line breakdown should be viewed in this light.

8.4.15 How much time is spent on investigating new practices, tools, trends and innovation in software engineering?

Team members are sent on courses on a regular bases. The goal of this training is to equip the team with knowledge of new potential solutions to problems. The practice also enables the team members to keep up with trends in the industry.

Investigation of new technologies that may prove to be useful to current and future projects is also encouraged.

Where applicable new techniques and technologies are used in projects to discover the practical implications. During the debriefing sessions the effectiveness of the new technologies are evaluated and discussed.

If an unknown technology is to form part of a project, it will be explored at the beginning of the project.

8.4.16 Do projects often require that the developers have to become familiar with new domains and/or technology?

A major proportion of the projects introduces some or other new technology or new domain. This merely reflects the diverse range of domains and technologies in which the internal clients are involved. Another reason for this diversity may be that the projects often entail developing solutions to problems that third parties have not addresses. The cost associated with buying solutions from third parties may also be a reason for requesting TDL to do a project, since the international location of these third parties and their alleged expertise may lead to inflated prices.

8.4.17 Is any development maturity measurement (such as CMM or something similar) carried out?

The TDL section does not undergo any formal capability maturity measurement appraisals. This is partially due to the fact that the section mostly addresses internal development needs and therefore has no pressing need to impress external clients by a high CMM rating. The managers use the reports generated by Rational Rose for measuring the team’s performance.

8.5 The key TDL Challenge

Like most other software development teams, the interviewed team confirmed that they experience the familiar problem of trying to meet changing user-interface specifications. It seems that in many software development projects that contain a user-interface, the engineers are likely to experience this problem.

The problem is attributable to two factors. The first has to do with the fact that the customer's first impression and experience of the system is through the user-interface. The second factor is the fickle and individualistic nature of human beings. One's state of mind differs from day to day. What seemed like a good way of using an application yesterday may seem cumbersome and user-unfriendly today. In addition, every person has individual likes, dislikes and ways of accomplishing a task. As a result, when multiple people attempt to define the user-interface specifications, disagreement is inevitable.

Human Computer Interaction (HCI) theory dictates that the design of a user-interface should be carried out by a specialist and not by the user, see [Shneiderman, 1997]. Adhering to this rule thus means that the customer should not actually be able to define the interface. While it may be infeasible in practice to completely prevent the client from having a say in the interface design, it is suggested that this should be adhered to as far as possible.

Dealing with the problem of changing user-interface specifications seems to be a major stumble block, irrespective of the process and methodology followed. Some processes try to reduce this problem by freezing the specification as soon as possible. This is done by documenting the user-interface specification and requiring the customer sign off on it. Thereafter, changes to the specification are either entirely disallowed, or accompanied by heavy penalties. This practice runs counter to the business rule that proclaims that "The Customer is King". The customer should be satisfied with the system. If not, the system may not be used and the project will be deemed to be a failure. This brings us back to square one. How does one keep the customer satisfied while limiting changes to the user-interface specification?

One possible approach may be to specify a set of "project rules" at the start of the project. These rules should then contain the procedure for handling change in general and user-interface design changes in particular. The repercussion of changes to the user-interface design in respect of cost and time (project schedule) should be clearly stated. This approach corresponds to the idea of the "Problem Resolution Process" defined in the ISO/IEC 12207: 1995 standard. (See [National Committee TC 71.1 (Information technology), 1995]). It should help the customer to understand that changes to specifications may result in schedule changes and increases to monetary expenditures.

The approach dictated by agile methodologies such as XP, Crystal and SCRUM is to use specification logs. In this approach, any change is regarded as a new specification to the system and is introduced into the next iteration. At the start of an iteration the customer is allowed to choose which of the requirements needed should be delivered in the next release of the system. The selection of requirements that are to be delivered next should be based on the priority of the features with respect to the iteration length (the number of features that can be completed in one iteration). In this scenario the presence of on-site customers may also produce better results, especially if one or more of the on-site customers is an HCI specialist. Such on-site customers will also ensure that the development team will receive faster feedback on the user-interfaces as they are developed. In turn also reducing the effort required to accept changes.

8.6 Making RUP more agile

One of RUP's practices is that it should be tailored to conform with the project's needs. Through this practice, a team and/or organisation is able to adjust its implementation of RUP to be more agile. Being agile enables a team to reap the benefits of quickly adapting to change and of providing highly incremental delivery of solutions.

Several papers have been written to provide guidelines on how to adapt RUP to be more agile. These include [Kruchten, 2001; Hirsch, 2002; Pollice, 2001]. Some guidelines worth mentioning are:

Artifact usage should be considered. Artifacts to be generated should be selected on the basis of the value that they will provide. Limiting the number of artifacts helps to reduce the effort and expenditure on their maintenance. The detail level of the artifacts should also be considered especially in TDL's case.

The *level of detail* during planning should be as low as possible to be able to adapt to change. One should try to limit the scope of detailed plans to a single iteration at a time. Reducing the time spend on big upfront design and maximising the usage of iterative development may be applicable to TDL.

Customer collaboration should be maximised to ensure a constant and quick feedback environment. Although TDL tries to collaborate as much as possible with customers, this goal should always be actively pursued and maximisation be sought after.

Try using fixed *iteration cycles*, each of which concludes with the delivery of a functioning software artifact. A fixed iteration cycle of for example one month, resulting with either the internal or external release of a limited working software product, may be beneficial to TDL.

Iteration reviews should be held between iterations. These reviews should analyse the process and practices used. The investigation should produce ideas on how to fine tune the methodology used to be even more productive.

8.7 Conclusion

Through the investigation conducted, it has emerged that certain techniques and practices advocated by agile methodologies, are indeed being practiced by the development team interviewed. These techniques include pair development and automated testing.

However, the underlying principle of RUP is to do a significant amount of upfront design. This approach towards design (also known as “big upfront design" or BUD) is required or encouraged by most of the traditional and bureaucratic methodologies. However the approach is discouraged by the agile community, where the belief is that following a plan is less important than responding to change. (See principle 4 of the agile manifesto [Agile Manifesto URL].) The agile viewpoint is that heavy modeling of a system to produce software causes the development team to focus on the design and plan of the system and not the actual code of the system.

The agile approach thus considers that the use of intensive modeling to determine feasibility may not be the most efficient way of producing software. Instead, it considers that the feasibility of a system will become apparent much earlier if high risk features of requirements are selected to be implemented and practically tested first. If this approach is followed, the stakeholders are in a position to cancel the project before too much effort and time has been expended. For example, in applying a BUD methodology to a project, the feasibility models may typically take between 2 and 3 months to reach a stage of being sufficiently detailed to determine the project's feasibility. In contrast, an agile methodology such Extreme Programming may detect non-feasibility within the first or second iteration of attempting to implement the most critical high risk features. This could typically occur within 4 to 6 weeks.

In respect of the types of projects undertaken by TDL, the use of agile methodologies would seem to be a viable approach. However selection of an appropriate methodology should take into consideration the following aspects.

Firstly, how susceptible is the project's specification to change? Projects whose specifications are likely to remain stable from start to finish may not benefit very much from an agile methodology.

Secondly, is it feasible to deliver the system in an evolutionary way to the users? Can a system with limited functionality be introduced to the user as soon as possible, with extended versions being delivered on a regular basis thereafter? This approach may provide certain benefits to all the stakeholders, and may be of great importance to them. It is in the nature of agile methodologies to deliver working systems in an iterative fashion.

Thirdly, the project risk involved needs to be analysed, taking into consideration practices that may be used to reduce these risks.

In the end, the practice of having a methodology per project as proclaimed by Cockburn [2002a], appears to be a sensible approach. The focus should be on the project "with its constraints and its environment driving your choice in the most adequate process configuration." [Kruchten, 2001] and not on the process driving the project!

For future research the author would like to investigate the implementation of his recommendations in TDL.

This case study was presented at the annual *Southern African Telecommunication Networks and Applications Conference* of 2003 [Theunissen and Kourie, 2003].

Chapter 9

Conclusion

This research set out to determine what the whole agile approach to software development entailed. In this quest several questions were raised and investigated. To enable the author to investigate these questions he first had to gain a deep understanding of the underlying principles and practices associated with the methodologies that are classified as agile. This understanding was gained through an in-depth literature study. The result of this enquiry was reported in Chapters 2 to 5. Agile methodologies seem to build on and maximise the strengths of humans in a software development project, whereas the 'traditional' approach appears to focus on minimising the risk associated with human 'short comings' by establishing more rigid processes. These agile methodologies also take a different approach towards the customer. Fulfilling the need of the customer is of the utmost importance, leading to practices such as having an on-site customer. This also means that the solution is adapted to the changing requirements of the customers.

The practices advocated by the agile methodologies are not new. Some may even be described simply as common sense. However the implementation and combination thereof is different. This is reflected in the words of extreme programming advocates: "taking it to the extreme". Stressing face-to-face communication over comprehensive documentation is an example of this focus shift.

As with any methodology, the implementation of the methodology is usually unique to each environment where it is applied, causing different outlooks on the viability of the associate methodology. This point needs to be kept in mind when evaluating the feasibility of agile methodologies for a specific project. Other points to take note of are Cockburn's methodology-per-project philosophy and the policy of being open to adjust the practices used as the project progresses.

On the debate of whether agile software development will dominate development efforts – forcing 'traditional' methodologies to disappear – indications are that both approaches will co-exist and probably influence one other. This may lead to both sides adopting practices from one other. This opinion is shared by various researchers, such as Baskerville [Baskerville Communication, 2003].

In the end, when considering software development methodologies one needs to realise that there does not exist a 'silver bullet'. Every situation is unique.

Compliance with software engineering standards such as ISO 12207:1995 when using agile methodologies has been discussed in Chapter 6. Several guidelines enabling conformance were presented.

To determine whether or not the benefits of these agile methodologies have been exaggerated by their authors and advocates, a case study was conducted on the experiences of established XP practitioners. The case study consisted of a qualitative analysis of a software development team implementing XP. This team provides an in-house solution for a financial company that is located in South Africa. The members are experienced developers and have established academic standings. The analysis was conducted through interviews and individual questionnaires. The case study brought several issues to light regarding implementing an XP approach to software development. The overall conclusion drawn from the aforementioned study is that XP *largely provides the benefits advocated by its authors, provided that the practices and principles defined for XP are strictly adhered to*.

An investigation into the current software development practices in the telecommunication industry indicated that several agile practices were being used. This investigation was conducted primarily through an *in situ* examination of a software development unit from Telkom SA Ltd., the primary telecommunication company in South Africa. The study further indicated that *there is room for using agile methodologies in a variety of telecommunication industry specific projects*.

Through this study a number of additional questions have emerged for future research. The growth in the adoption of open source software and the increased utilisation of component based software development has brought new challenges to software development management. Several of these are noted in the following paragraphs.

The influence that open source development and agile software development exert on each other needs to be explored. Raymond, one of the open source advocates, provided some comments on this relationship in a recent article (see [Raymond, 2003]).

In Baskerville et al. [2003] the tendency of agile software developers "to build 'with reuse,' but less 'for reuse"' was uncovered. This tendency and the influence of component based software development on agile software development deserves to be investigated.

The guidelines proposed in Chapter 6 to aid in complying to ISO 12207:1995 needs to be practically tested and verified. Further examination into compliance to CMM and ISO 15288:2002 should also be undertaken.

> "As the water shapes itself to the vessel that contains it, so a wise man adapts himself to circumstances." – Confucius

Derived Publications

1. *A Case Study of Software Development Processes in the Telecommunication Industry*, WHM Theunissen and DG Kourie, Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC) 2003, September 2003, Fancourt, Southern Cape, South Africa.

2. *Standards and Agile Software Development*, WHM Theunissen, DG Kourie and BW Watson, Proceedings of SAICSIT 2003: Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, September 2003, Fourways, South Africa.

Bibliography

[Abrahamsson et al. 2002] ABRAHAMSSON, P , SALO, O , RONKAINEN, J , and WARSTA, J: *Agile software development methods: Review and analysis*. VTT Publication, 2002 (478). – URL `http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf`. – accessed: 2003/10/17

[Adaptive URL] : *Adaptive Software Development Homepage*. – URL `http://www.adaptivesd.com`. – accessed: 2003/06/30

[Agile History URL] HIGHSMITH, J: *History: The Agile Manifesto*. – URL `http://www.agilemanifesto.org/history.html`. – accessed: 2002/02/01

[Agile Manifesto URL] : *Manifesto for Agile Software Development*. – URL `http://www.agilemanifesto.org`. – accessed: 2003/05/20

[Agile Modeling URL] : *Agile Modeling Homepage*. – URL `http://www.agilemodeling.com`. – accessed: 2003/06/30

[Balbes and Button 2002] BALBES, M , and BUTTON, B: . *Embedded Systems Conference*. Chicago, 2002

[Baskerville et al. 2003] BASKERVILLE, R , RAMESH, B , LEVINE, L , PRIES-HEJE, J , and SLAUGHTER, S: Is Internet-Speed Software Development Different? : *IEEE Software* (2003), November/December

[Baskerville Communication 2003] : *Personal Communication with Richard Baskerville*. September 2003

[Beck 2000] BECK, K: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. – ISBN 201-61641-6

[Beck and Fowler 2000] BECK, K , and FOWLER, M: *Planning Extreme Programming*. 1st. Addison-Wesley Pub. Co, October 2000

[Boehm 1988] BOEHM, BW: A Spiral Model of Software Development and Enhancement. : *IEEE Computer* 21 (1988), May, No. 5, pp. 61–72

[Brooks 1995] BROOKS, Jr., FP: *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995

[Coad and De Luca 1999] COAD, P , and DE LUCA, J: *Java Modeling in Color with UML*. Prentice Hall, 1999

[Coad and Mayfield 1992] COAD, P , and MAYFIELD, M: Object-Oriented Patterns. : *Communications of the ACM* (1992), September

[Cockburn 1998] COCKBURN, A: *Surviving Object-Oriented Projects*. Addison-Wesley Pub Co, 1998. – ISBN 0-201-49834-0

[Cockburn 2000] COCKBURN, A: Reexamining the Cost of Change Curve. : *XP Magazine* (2000), September. – URL `http://www.xprogramming.com/xmag/cost\protect\T1\textunderscoreof\protect\T1\textunderscorechange.htm`. – accessed: 2002/04/01

[Cockburn 2001] COCKBURN, A: Crystal Light Methods. : *Cutter IT Journal* (2001)

[Cockburn 2002a] COCKBURN, A: *Agile Software Development*. Pearson Education, Inc, 2002. – ISBN 0-201-69969-9

[Cockburn 2002b] COCKBURN, A: Games Programmers Play. : *Software Development Magazine* (2002), February

[Cusumano and Yoffie 2000] CUSUMANO, MA , and YOFFIE, DB: *Competing on Internet Time: Lessons from Netscape and its battle with Microsoft*. Touchstone, 2000

[DeLuca Biography URL] : *Jeff De Luca's Biography*. – URL `http://www.nebulon.com/pr/bio.html`. – accessed: 2003/08/11

[DeMarco and Lister 1987] DEMARCO, T , and LISTER, T: *Peopleware: Productive Projects and Teams*. 1987

[DSDM URL] : *DSDM Homepage*. – URL `http://www.dsdm.org`. – accessed: 2003/05/15

[eBucks URL] : *eBucks Homepage*. – URL `http://www.ebucks.com`. – accessed: 2002/10/14

[Equinox Interview 2002] : *Personal communication with Equinox development team*. February 2002

[Fowler and Fowler 1964] FOWLER, HW , and FOWLER, FG: *The concise Oxford dictionary of current english*. London : Oxford University Press, 1964

[Fowler 1999] FOWLER, M: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

[Fowler and Highsmith 2001] FOWLER, M , and HIGHSMITH, J: The Agile Manifesto. : *Software Development Magazine* (2001), August

[Fusion Summary 1998] Engineering Process Summary (Fusion 2.0) / Hewlett-Packard Company. January 1998. – Research Report

[Gamma et al. 1995] GAMMA, E , HELM, R , JOHNSON, R , and VLISSIDES, J: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

[Gido and Clements 1999] GIDO, J , and CLEMENTS, JP: *Successful Project Management*. South-Western College Publishing, 1999

[Gough-Jones et al. 1993] GOUGH-JONES, VJ , JACOBS, SJ , ROS, NCJ , BRENKMAN, AMJ , PIETERSE, PHS , and VIELHAUER, AA: *Rekenaarstudie vir Vandag - Pascal st.8*. 2. Johannesburg : Lexicon Uitgewers, 1993

[Grenning 2002] GRENNING, J: . *Embedded Systems Conference*. Chicago, 2002

[Highsmith 2002] HIGHSMITH, J: Does Agility Work? : *Software Development Magazine* (2002), June

[Highsmith and Cockburn 2001] HIGHSMITH, J , and COCKBURN, A: Agile Software Development: The Business of Innvovation. : *IEEE Computer* (2001), September, pp. 120–122

[Highsmith 2000] HIGHSMITH, JA: *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, 2000

[Hirsch 2002] HIRSCH, M: . *Conference on Object Oriented Programming Systems Languages and Applications: Practitioners Reports*, 2002

[Hunt and Thomas 1999] HUNT, A , and THOMAS, D: *The Pragmatic Programmer*. Addison-Wesley, 1999

[IEEE & EIA 1998a] IEEE & EIA: IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995. April 1998. – Research Report

[IEEE & EIA 1998b] IEEE & EIA: IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995. April 1998. – Research Report

[ISO/IEC JTC1-SC7 URL] : *ISO/IEC JTC1-SC7 Homepage*. – URL `http://www.jtc1-sc7.org`. – accessed: 2002/12/01

[Jeffries et al. 2000] JEFFRIES, R , ANDERSON, A , and HENDRICKSON, C: *Extreme Programming Installed*. Addison-Wesley Pub Co, October 2000

[Kruchten 2000] KRUCHTEN, P: *The Rational Unified Process: An Introduction*. 2nd. Addison-Wesley, 2000

[Kruchten 2001] KRUCHTEN, P: Agility with the RUP. : *Cutter IT Journal* 14 (2001), December, No. 12

[M-W URL] : *Merriam-Webster Online*. – URL `http://www.m-w.com`. – accessed: 2003/10/17

[Mehrabian 1981] MEHRABIAN, A: *Silent messages: Implicit communication of emotions and attitudes*. 1981

[National Committee TC 71.1 (Information technology) 1995] NATIONAL COMMITTEE TC 71.1 (INFORMATION TECHNOLOGY): SABS ISO/IEC 12207:1995, Information technology - Software life cycle processes / South African Bureau of Standards. 1995. – Research Report

[Opdyke 1992] OPDYKE, WF: *Refactoring: Object-Oriented Frameworks*, University of Illinois, PhD Thesis, 1992

[O'Reilly Interview 1997] : *O'Reilly Interview with Tim Berners-Lee*. 1997

[Palmer and Felsing 2002] PALMER, S , and FELSING, M: *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002

[Parkinson 1958] PARKINSON, CN: *Parkinson's Law: The Pursuit of Progress*. John Murray, 1958

[Paulk 2001] PAULK, MC: Extreme Programming from a CMM Perspective. : *IEEE Software* (2001), November/December

[Paulson 2001] PAULSON, LD: The Consultancy Approach to Aligning Business and IT. : *IEEE IT Professional* 3 (2001), May/June, No. 3, pp. 12–15

[Pollice 2001] POLLICE, G: Using the Rational Unified Process for Small Projects: Expanding on eXtreme Programming / Rational Software. 2001. – Research Report

[Poppendieck 2001] POPPENDIECK, M: Lean Programming. : *Software Development Magazine* (2001), May/June

[Poppendieck and Poppendieck 2003] POPPENDIECK, M , and POPPENDIECK, T: *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003

[Pragmatic Programming URL] : *Pragmatic Programming Homepage*. – URL `http://www.pragmaticprogrammer.com`. – accessed: 2003/06/30

[Rational URL] : *Rational Software Corporation Homepage*. – URL `http://rational.com`. – accessed: 2003/06/01

[Raymond 2003] RAYMOND, ES: *Discovering the Obvious: Hacking and Refactoring*. June 2003. – URL `http://www.artima.com/weblogs/viewpost.jsp?thread=5342`. – accessed: 2003/10/01

[Reifer 2002] REIFER, R: Ten Deadly Risks in Internet and Intranet Software Development. : *IEEE Software* (2002), March/April, pp. 12–14

[Robbins 2001] ROBBINS, SP: *Organizational Behavior*. 9th. Prentice Hall, 2001

[Roberts 1999] ROBERTS, DB: *Practical Analysis for Refactoring*, University of Illinois, PhD Thesis, 1999

[Rumpe and Schöder 2002] RUMPE, B , and SCHÖDER, A: . *3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2002, pp. 95–100

[Schwaber 1995] SCHWABER, K: . *OOPLSA'95 Workshop on Business Object Design and Implementation*, 1995

[Schwaber and Beedle 2001] SCHWABER, K , and BEEDLE, M: *Agile Software Development with Scrum*. Prentice Hall, 2001

[SCRUM URL] : *SCRUM Homepage*. – URL `http://www.controlchaos.com`. – accessed: 2002/10/30

[SDM's P&P 2003] SDM'S P&P: *Software Development Magazine's People & Projects Newsletter*. August 2003

[Shneiderman 1997] SHNEIDERMAN, B: *Designing the User Interface*. 3rd. Addison-Wesley, 1997

[SteP10 URL] : *SteP10 Homepage*. – URL `http://www.step-10.com`. – accessed: 2002/11/04

[Taber and Fowler 2000] TABER, C , and FOWLER, M: An Iteration in the Life of an XP Project. : *Cutter IT Journal* 13 (2000), November, No. 11

[Team 2002] TEAM, MSF: MSF Process Model v. 3.1 / Microsoft. URL `http://www.microsoft.com/msf/`. – accessed: 2003/06/15, June 2002. – Research Report

[Theunissen and Kourie 2003] THEUNISSEN, WHM , and KOURIE, DG: . *Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC) 2003*. Fancourt, Southern Cape, South Africa, September 2003

[Theunissen et al. 2003] THEUNISSEN, WHM , KOURIE, DG , and WATSON, BW: *Standards and Agile Software Development*. : Editors: . ELOFF, J , ENGELBRECHT, A , KOTZE, P , and ELOFF, M: *Proceedings of SAICSIT 2003: Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*. Fourways, South Africa : ACM, September 2003, pp. 178–188

[Wake 2002] WAKE, WC: . *The Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering*, 2002

[Wilkes 1985] WILKES, MV: *Memoirs of a Computer Pioneer*. Cambridge MA : The MIT Press, 1985

[Williams et al. 2000] WILLIAMS, L , KESSLER, RR , CUNNINGHAM, W , and JEFFRIES, R: Strengthening the Case for Pair Programming. : *IEEE Software* (2000), July/August, pp. 19–23

[Wright 2003] WRIGHT, G: . *Proceedings of the XP Agile Universe 2003 Conference*. New Orleans, August 2003

[XP URL] : *eXtreme Programming Homepage*. – URL `www.extremeprogramming.org`. – accessed: 2003/02/01

[XP123 URL] : *XP123 Homepage*. – URL `http://xp123.com`. – accessed: 2003/01/14

Appendix A

The Agile Manifesto

Extract from *The Agile Manifesto* Fowler and Highsmith [2001]:

The Manifesto for Agile Software Development

Seventeen anarchists agree:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

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

We follow the following principles:

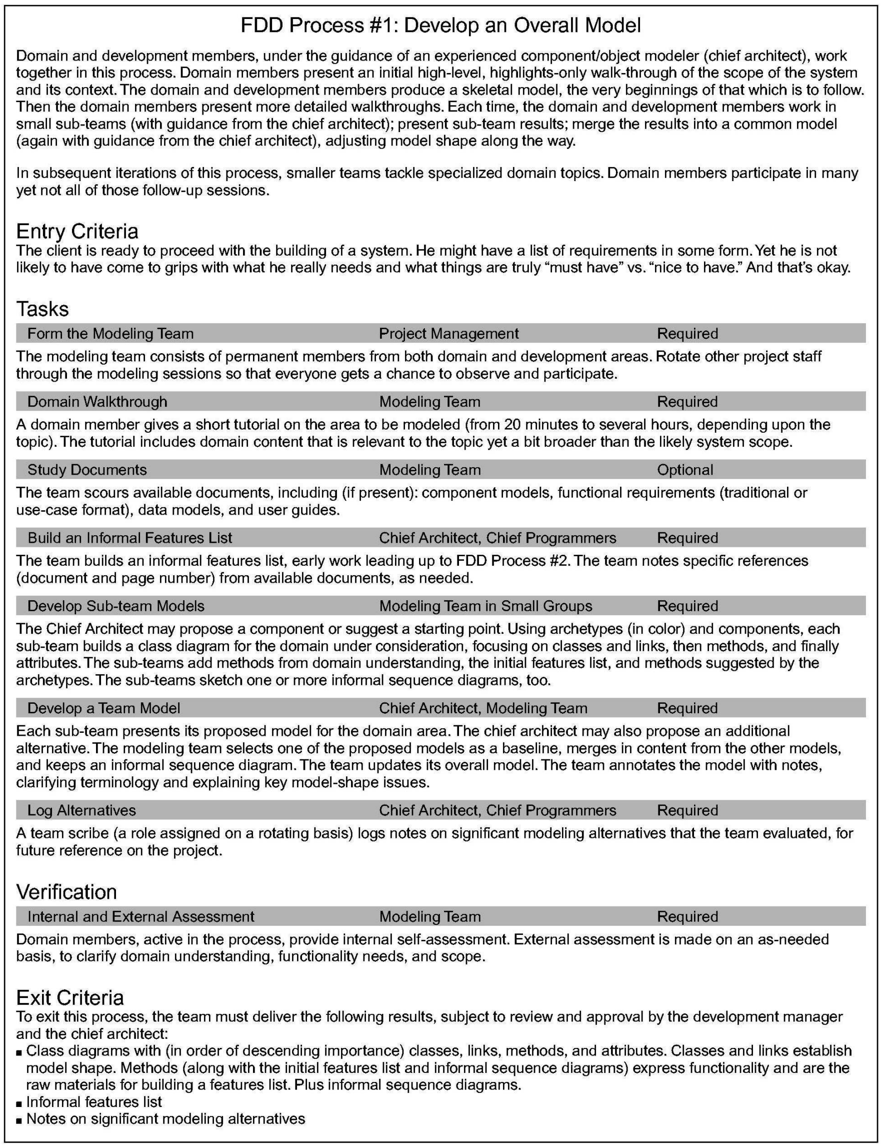
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity ‘the art of maximizing the amount of work not done’ is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

Appendix B

FDD Summarised



FDD Process #1: Develop an Overall Model

Domain and development members, under the guidance of an experienced component/object modeler (chief architect), work together in this process. Domain members present an initial high-level, highlights-only walk-through of the scope of the system and its context. The domain and development members produce a skeletal model, the very beginnings of that which is to follow. Then the domain members present more detailed walkthroughs. Each time, the domain and development members work in small sub-teams (with guidance from the chief architect); present sub-team results; merge the results into a common model (again with guidance from the chief architect), adjusting model shape along the way.

In subsequent iterations of this process, smaller teams tackle specialized domain topics. Domain members participate in many yet not all of those follow-up sessions.

Entry Criteria

The client is ready to proceed with the building of a system. He might have a list of requirements in some form. Yet he is not likely to have come to grips with what he really needs and what things are truly “must have” vs. “nice to have.” And that's okay.

Tasks

Form the Modeling Team	Project Management	Required

The modeling team consists of permanent members from both domain and development areas. Rotate other project staff through the modeling sessions so that everyone gets a chance to observe and participate.

Domain Walkthrough	Modeling Team	Required

A domain member gives a short tutorial on the area to be modeled (from 20 minutes to several hours, depending upon the topic). The tutorial includes domain content that is relevant to the topic yet a bit broader than the likely system scope.

Study Documents	Modeling Team	Optional

The team scours available documents, including (if present): component models, functional requirements (traditional or use-case format), data models, and user guides.

Build an Informal Features List	Chief Architect, Chief Programmers	Required

The team builds an informal features list, early work leading up to FDD Process #2. The team notes specific references (document and page number) from available documents, as needed.

Develop Sub-team Models	Modeling Team in Small Groups	Required

The Chief Architect may propose a component or suggest a starting point. Using archetypes (in color) and components, each sub-team builds a class diagram for the domain under consideration, focusing on classes and links, then methods, and finally attributes. The sub-teams add methods from domain understanding, the initial features list, and methods suggested by the archetypes. The sub-teams sketch one or more informal sequence diagrams, too.

Develop a Team Model	Chief Architect, Modeling Team	Required

Each sub-team presents its proposed model for the domain area. The chief architect may also propose an additional alternative. The modeling team selects one of the proposed models as a baseline, merges in content from the other models, and keeps an informal sequence diagram. The team updates its overall model. The team annotates the model with notes, clarifying terminology and explaining key model-shape issues.

Log Alternatives	Chief Architect, Chief Programmers	Required

A team scribe (a role assigned on a rotating basis) logs notes on significant modeling alternatives that the team evaluated, for future reference on the project.

Verification

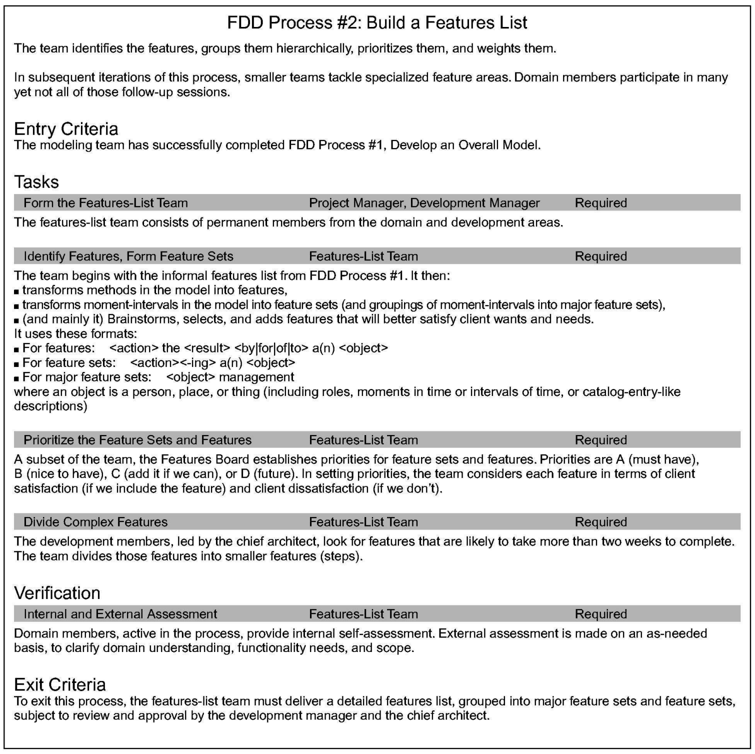
Internal and External Assessment	Modeling Team	Required

Domain members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, to clarify domain understanding, functionality needs, and scope.

Exit Criteria

To exit this process, the team must deliver the following results, subject to review and approval by the development manager and the chief architect:

- Class diagrams with (in order of descending importance) classes, links, methods, and attributes. Classes and links establish model shape. Methods (along with the initial features list and informal sequence diagrams) express functionality and are the raw materials for building a features list. Plus informal sequence diagrams.
- Informal features list
- Notes on significant modeling alternatives



FDD Process #2: Build a Features List

The team identifies the features, groups them hierarchically, prioritizes them, and weights them.

In subsequent iterations of this process, smaller teams tackle specialized feature areas. Domain members participate in many yet not all of those follow-up sessions.

Entry Criteria

The modeling team has successfully completed FDD Process #1, Develop an Overall Model.

Tasks

Form the Features-List Team	Project Manager, Development Manager	Required

The features-list team consists of permanent members from the domain and development areas.

Identify Features, Form Feature Sets	Features-List Team	Required

The team begins with the informal features list from FDD Process #1. It then:
- transforms methods in the model into features,
- transforms moment-intervals in the model into feature sets (and groupings of moment-intervals into major feature sets),
- (and mainly it) Brainstorms, selects, and adds features that will better satisfy client wants and needs.

It uses these formats:
- For features: <action> the <result> <by|for|of|to> a(n) <object>
- For feature sets: <action><-ing> a(n) <object>
- For major feature sets: <object> management

where an object is a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)

Prioritize the Feature Sets and Features	Features-List Team	Required

A subset of the team, the Features Board establishes priorities for feature sets and features. Priorities are A (must have), B (nice to have), C (add it if we can), or D (future). In setting priorities, the team considers each feature in terms of client satisfaction (if we include the feature) and client dissatisfaction (if we don't).

Divide Complex Features	Features-List Team	Required

The development members, led by the chief architect, look for features that are likely to take more than two weeks to complete. The team divides those features into smaller features (steps).

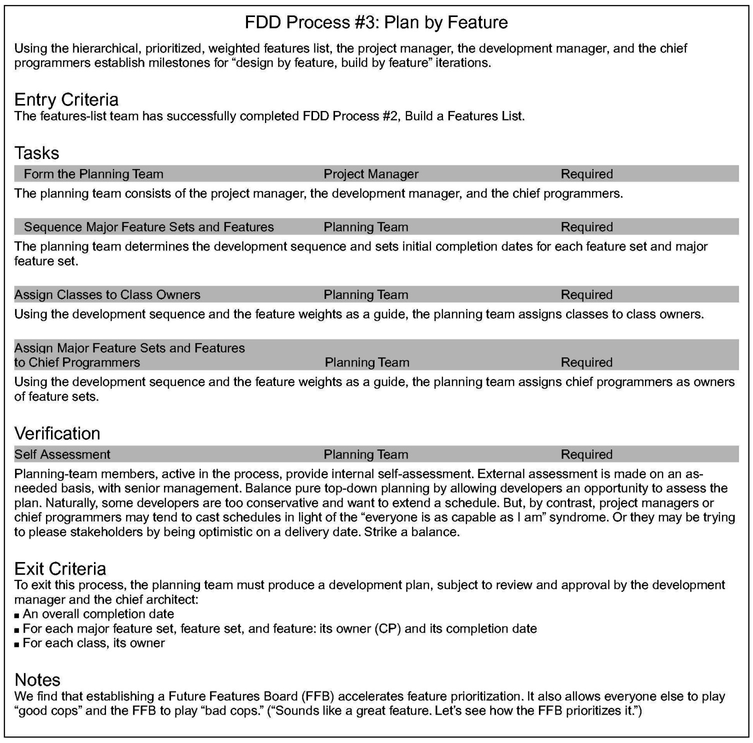
Verification

Internal and External Assessment	Features-List Team	Required

Domain members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, to clarify domain understanding, functionality needs, and scope.

Exit Criteria

To exit this process, the features-list team must deliver a detailed features list, grouped into major feature sets and feature sets, subject to review and approval by the development manager and the chief architect.



FDD Process #3: Plan by Feature

Using the hierarchical, prioritized, weighted features list, the project manager, the development manager, and the chief programmers establish milestones for "design by feature, build by feature" iterations.

Entry Criteria

The features-list team has successfully completed FDD Process #2, Build a Features List.

Tasks

Form the Planning Team	Project Manager	Required

The planning team consists of the project manager, the development manager, and the chief programmers.

Sequence Major Feature Sets and Features	Planning Team	Required

The planning team determines the development sequence and sets initial completion dates for each feature set and major feature set.

Assign Classes to Class Owners	Planning Team	Required

Using the development sequence and the feature weights as a guide, the planning team assigns classes to class owners.

Assign Major Feature Sets and Features to Chief Programmers	Planning Team	Required

Using the development sequence and the feature weights as a guide, the planning team assigns chief programmers as owners of feature sets.

Verification

Self Assessment	Planning Team	Required

Planning-team members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, with senior management. Balance pure top-down planning by allowing developers an opportunity to assess the plan. Naturally, some developers are too conservative and want to extend a schedule. But, by contrast, project managers or chief programmers may tend to cast schedules in light of the "everyone is as capable as I am" syndrome. Or they may be trying to please stakeholders by being optimistic on a delivery date. Strike a balance.

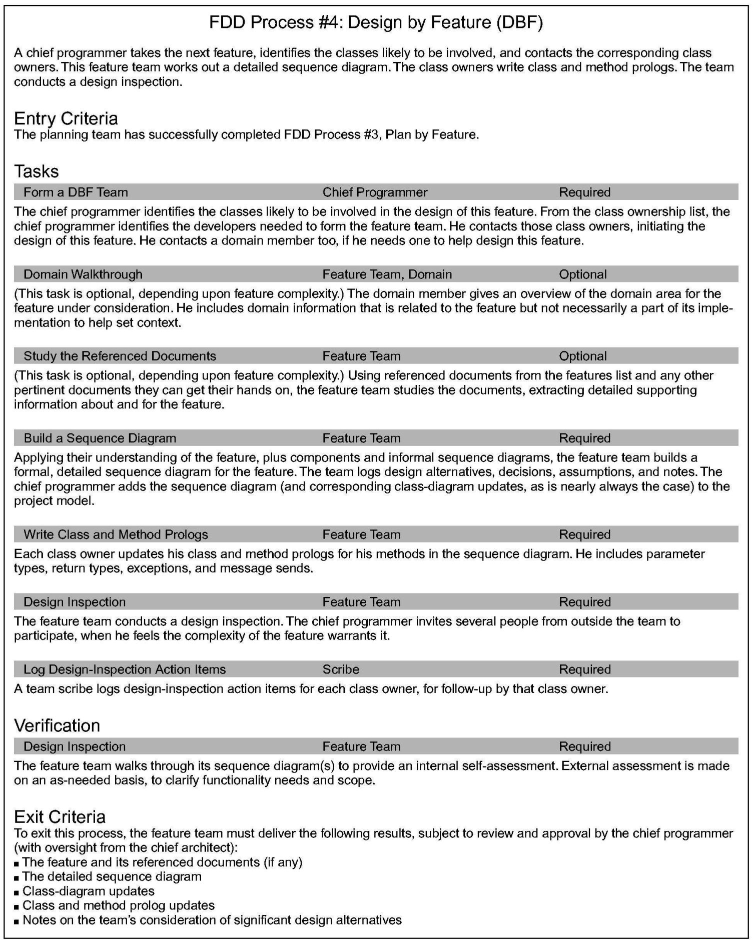
Exit Criteria

To exit this process, the planning team must produce a development plan, subject to review and approval by the development manager and the chief architect:

- An overall completion date
- For each major feature set, feature set, and feature: its owner (CP) and its completion date
- For each class, its owner

Notes

We find that establishing a Future Features Board (FFB) accelerates feature prioritization. It also allows everyone else to play "good cops" and the FFB to play "bad cops." ("Sounds like a great feature. Let's see how the FFB prioritizes it.")



FDD Process #4: Design by Feature (DBF)

A chief programmer takes the next feature, identifies the classes likely to be involved, and contacts the corresponding class owners. This feature team works out a detailed sequence diagram. The class owners write class and method prologs. The team conducts a design inspection.

Entry Criteria

The planning team has successfully completed FDD Process #3, Plan by Feature.

Tasks

Form a DBF Team	Chief Programmer	Required

The chief programmer identifies the classes likely to be involved in the design of this feature. From the class ownership list, the chief programmer identifies the developers needed to form the feature team. He contacts those class owners, initiating the design of this feature. He contacts a domain member too, if he needs one to help design this feature.

Domain Walkthrough	Feature Team, Domain	Optional

(This task is optional, depending upon feature complexity.) The domain member gives an overview of the domain area for the feature under consideration. He includes domain information that is related to the feature but not necessarily a part of its implementation to help set context.

Study the Referenced Documents	Feature Team	Optional

(This task is optional, depending upon feature complexity.) Using referenced documents from the features list and any other pertinent documents they can get their hands on, the feature team studies the documents, extracting detailed supporting information about and for the feature.

Build a Sequence Diagram	Feature Team	Required

Applying their understanding of the feature, plus components and informal sequence diagrams, the feature team builds a formal, detailed sequence diagram for the feature. The team logs design alternatives, decisions, assumptions, and notes. The chief programmer adds the sequence diagram (and corresponding class-diagram updates, as is nearly always the case) to the project model.

Write Class and Method Prologs	Feature Team	Required

Each class owner updates his class and method prologs for his methods in the sequence diagram. He includes parameter types, return types, exceptions, and message sends.

Design Inspection	Feature Team	Required

The feature team conducts a design inspection. The chief programmer invites several people from outside the team to participate, when he feels the complexity of the feature warrants it.

Log Design-Inspection Action Items	Scribe	Required

A team scribe logs design-inspection action items for each class owner, for follow-up by that class owner.

Verification

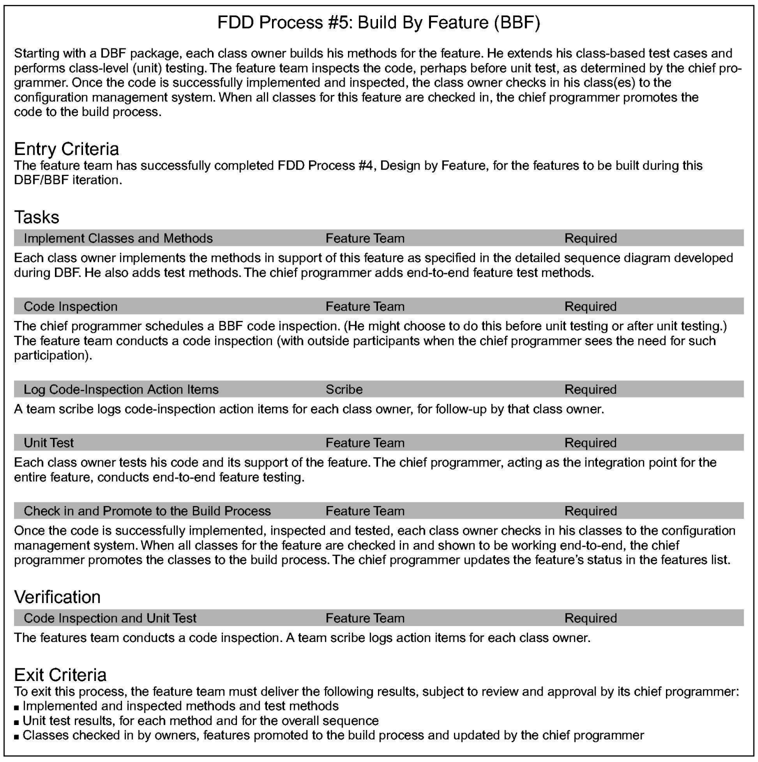
Design Inspection	Feature Team	Required

The feature team walks through its sequence diagram(s) to provide an internal self-assessment. External assessment is made on an as-needed basis, to clarify functionality needs and scope.

Exit Criteria

To exit this process, the feature team must deliver the following results, subject to review and approval by the chief programmer (with oversight from the chief architect):

- The feature and its referenced documents (if any)
- The detailed sequence diagram
- Class-diagram updates
- Class and method prolog updates
- Notes on the team's consideration of significant design alternatives



FDD Process #5: Build By Feature (BBF)

Starting with a DBF package, each class owner builds his methods for the feature. He extends his class-based test cases and performs class-level (unit) testing. The feature team inspects the code, perhaps before unit test, as determined by the chief programmer. Once the code is successfully implemented and inspected, the class owner checks in his class(es) to the configuration management system. When all classes for this feature are checked in, the chief programmer promotes the code to the build process.

Entry Criteria

The feature team has successfully completed FDD Process #4, Design by Feature, for the features to be built during this DBF/BBF iteration.

Tasks

Implement Classes and Methods	Feature Team	Required

Each class owner implements the methods in support of this feature as specified in the detailed sequence diagram developed during DBF. He also adds test methods. The chief programmer adds end-to-end feature test methods.

Code Inspection	Feature Team	Required

The chief programmer schedules a BBF code inspection. (He might choose to do this before unit testing or after unit testing.) The feature team conducts a code inspection (with outside participants when the chief programmer sees the need for such participation).

Log Code-Inspection Action Items	Scribe	Required

A team scribe logs code-inspection action items for each class owner, for follow-up by that class owner.

Unit Test	Feature Team	Required

Each class owner tests his code and its support of the feature. The chief programmer, acting as the integration point for the entire feature, conducts end-to-end feature testing.

Check in and Promote to the Build Process	Feature Team	Required

Once the code is successfully implemented, inspected and tested, each class owner checks in his classes to the configuration management system. When all classes for the feature are checked in and shown to be working end-to-end, the chief programmer promotes the classes to the build process. The chief programmer updates the feature's status in the features list.

Verification

Code Inspection and Unit Test	Feature Team	Required

The features team conducts a code inspection. A team scribe logs action items for each class owner.

Exit Criteria

To exit this process, the feature team must deliver the following results, subject to review and approval by its chief programmer:

- Implemented and inspected methods and test methods
- Unit test results, for each method and for the overall sequence
- Classes checked in by owners, features promoted to the build process and updated by the chief programmer

Appendix C

Equinox Questionnaire

Individual Questionnaire on the Software Development Approach at Equinox Financial Solutions (Pty.) Ltd.

1. Which software development processes/methodologies did you use before using XP?
2. Are there any bottleneck activities in the way you implement XP?
3. How long have you been doing software development?
4. How long have you been practicing XP?
5. How would you rate XP as a factor in the success of Equinox' development effort?

Not a factor	Minor factor	Major factor	Most influential factor

6. How do you experience collective code ownership?
7. How do you experience the single room concept?
8. How do you experience pair-programming?
9. Do you experience pair-programming as a limiting factor in learning/experimenting with new technologies and techniques? Explain?
10. XP was invented to make software development more successful. Some of its main goals are listed below. Rate your XP-project in terms of the extent to which these goals were reached. Explain any obstacles in reaching the goals.

	Fully Achieved	Partially Achieved	Same as Previous Methodology	Worse
10.1. Delivered software on time	1	2	3	4
10.2. Let developers have fun in their work	1	2	3	4
10.3. Develop high quality software (fewer bugs)	1	2	3	4
10.4. Late changes don't incur high costs, because one can react quickly to changes	1	2	3	4

11. What is the level of use of XP-elements in your project? Please specify to what extent each element contributed to the success of the project.

	Level of use				Contribution to success of development			
	Not at all	Sometimes	Often	Continuously	Negative Contribution	No Contribution	Helpful	Indispensable
11.1. Planning Game	1	2	3	4	1	2	3	4
11.2. Short Release Cycles	1	2	3	4	1	2	3	4
11.3. Metaphor	1	2	3	4	1	2	3	4
11.4. Simple Design	1	2	3	4	1	2	3	4
11.5. Testing	1	2	3	4	1	2	3	4
11.6. Refactoring	1	2	3	4	1	2	3	4
11.7. Pair Programming	1	2	3	4	1	2	3	4
11.8. Common Code Ownership	1	2	3	4	1	2	3	4
11.9. Continuous Integration	1	2	3	4	1	2	3	4
11.10. 40-Hour-Week	1	2	3	4	1	2	3	4
11.11. On-Site Customer	1	2	3	4	1	2	3	4
11.12. Coding Standards	1	2	3	4	1	2	3	4

12. List the documentation that is generated.

13. How much value is gained from the documentation?

14. How comprehensive is the documentation?

15. Do you capture/archive the information generated on white-boards? How?

Never	Some	Always
How:		

16. How do you measure your performance/productivity?

17. How would you rate your productivity when practising XP compared to traditional processes?

Beter	Same	Worse

18. Would you advocate the use of XP to others?

Yes	No

19. Do you have any suggestions for improving any of the XP elements?

20. Any additional comments?