

UNIVERSITY OF CALGARY

Adopting Iterative and Incremental Development

An empirical research study in an industrial setting

by

Caryna Accioly Pinheiro

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2009

© Caryna Accioly Pinheiro 2009

UNIVERSITY OF CALGARY  
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Adopting Iterative and Incremental Development

An empirical research study in an industrial setting” submitted by Caryna Accioly Pinheiro in partial fulfillment of the requirements for the degree of Master Degree in Computer Science.

---

Supervisor, Dr. Frank Maurer  
Department of Computer Science

---

Co-supervisor, Dr. Jonathan Sillito  
Department of Computer Science

---

Dr. Guenther Ruhe  
Department of Software Engineering

---

Ron Murch  
Haskayne School of Business  
Senior Instructor

---

Date

## Preface

I would like to take this section to clarify the use of the word “*we*” in this thesis. I, Caryna Pinheiro, am the student eagerly pursuing my Masters Degree and the author of this work. However from here on I will use the word “*we*” to describe my work. I felt that the use of “*I*” to convey my research would not pay the well-deserved dues to my supervisors Dr. Frank Maurer and Dr. Jonathan Sillito. Their guidance and knowledge have been an active and inseparable part of my research. As such, I have chosen to describe my work in the context of the true team work a post graduate project entails between students and supervisors.

# Abstract

Though Iterative and Incremental Development (IID) practices have been around since 1950, many traditional managers see IID as the innovative response to the single-pass Waterfall process. Much of the literature about IID adoption in medium-to-large companies comes from its most well-known subset, Agile Methods. And this literature has focused on companies developing software products and/or providing software consultancy, or from sources that have been decontextualized. Understanding how IID practices work in other contexts is important for many IT managers. This thesis provides an empirical study of a group of projects in a large bureaucratic government agency that adopted IID practices. The primary goal of this research is to investigate how effectively the adoption of IID practices dealt with business concerns related to: poor software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays. To this end, we conducted two case studies with projects that migrated to IID practices and with a new project that followed IID since inception. In both cases the effects of these changes are considered quantitatively and qualitatively. This thesis also presents an action research project, with the goal of addressing two issues unresolved by the IID adoption: staging environment instability and lack of support for test automation. Based on the analysis of the data from these case studies, this research makes three key contributions: (1) an addition to the body of knowledge about the effects of introducing an IID approach into an organization, (2) documentation of the challenges and lessons learned in making those changes, and (3) a description of a test strategy that successfully introduced test automation solutions to legacy applications and configuration management issues.

## Publications

Some of the materials, ideas, and figures in this thesis have previously appeared in the following publications:

Caryna Pinheiro, Frank Maurer, and Jonathan Sillito. Improving quality, one process change at a time. In 31<sup>th</sup> International Conference on Software Engineering (ICSE 2009), volume 31, ICSE Companion 2009, pages 81-90. IEEE, 2009.

Caryna Pinheiro, Frank Maurer and Jonathan Sillito: Adopting iterative development: the perceived business value. In Proceedings of the conference on Agile Processes and Extreme Programming, pp. 185-189, 2008.

Caryna Pinheiro, Frank Maurer and Jonathan Sillito: Moving towards agility in a bureaucratic environment RUP as a bridge between Waterfall and Agile processes, Agile Conference 2008 (Research-in-Progress Workshop), Toronto.

Celio Santana, Cristine Gusmo, Liana Soares, Caryna Pinheiro, Teresa M. Maciel, Alexandre Marcos Lins de Vasconcelos, Ana Cristina Rouiller: Agile Software Development and CMMI: What We Do Not Know about Dancing with Elephants. In Proceedings of the conference on Agile Processes and Extreme Programming, pp. 124-129, 2009.

## Acknowledgements

I thank our Lord for giving me the strength to complete this journey. I owe to His grace all that I have and all that I am in this life.

I thank my family for being there for me. Marcus, your birth has filled our lives with such joy and love. With you I have learned what it means to love someone unconditionally. Fabricio, your support and understanding through the hard times has been a blessing that I hope to one day be able to retribute. Sorry for the late nights and lack of family time. Thank you for seeing this thesis as a “future investment.” I feel truly honoured to have you in my life. Mother, thank you for your babysitting help and for raising me to understand the importance of education. Grandma Hilda, I have learned so much from you.

I thank my supervisors. To Dr. Maurer and Dr. Sillito, thanks for all your help, guidance and support over the past few years. Thank you for giving me the chance to experience this journey. I feel truly honoured to have worked with you.

A warm thanks goes to a group of very special individuals that have helped me in many different ways: Trudy Kamphuis, Jim King, Bryan Schultz, Perry McKenzie, Dorina Tamas, and Guing Panillo.

## Dedication

To My Lord Jesus Christ, my deepest gratitude for all His blessing. Specially, thank you Lord for blessing me with a partner like Fabricio, and for our beautiful son Marcus.

# Table of Contents

Approval Page . . . . .	ii
Preface . . . . .	iii
Abstract . . . . .	iv
Publications . . . . .	v
Acknowledgements . . . . .	vi
Dedication . . . . .	vii
Table of Contents . . . . .	viii
List of Tables . . . . .	xi
List of Figures . . . . .	xii
1 Introduction . . . . .	1
1.1 Motivation for Studying the adoption of IID practices . . . . .	3
1.2 Overview of Research Goals and Methodological Approach . . . . .	6
1.3 Key contributions . . . . .	11
1.4 Thesis Overview . . . . .	13
2 Overall Organizational Context . . . . .	14
2.1 Inner Context . . . . .	17
2.2 Outer Context . . . . .	20
3 Literature Review . . . . .	21
3.1 Background and Context . . . . .	21
3.2 Iterative and Incremental Development Methodologies . . . . .	23
3.3 The Rational Unified Process . . . . .	25
3.4 Agile Methods . . . . .	29
3.5 Adoption strategies . . . . .	32
3.6 Related Experiences . . . . .	36
3.6.1 Experiencing transitioning to IID . . . . .	36
3.6.2 RUP and Agile methods comparisons . . . . .	39
4 Case Study 1 - Adoption path for process changes in a suite of existing applications . . . . .	43
4.1 The why of change - project specific context . . . . .	44
4.2 Data gathering - Available Project Metrics . . . . .	46
4.2.1 Quantitative Measurements . . . . .	47
4.2.2 Quantitative Measurements - Data Limitations . . . . .	50
4.2.3 Qualitative Measurements . . . . .	50
4.3 The how of change - process changes introduced . . . . .	52
4.3.1 Process change 1: Adopting Rational Tools and New Development/Architectural Approach (release IR1) . . . . .	52
4.3.2 Process change 2: Iteration Length Based on Tasks (releases: IR2 through IR6) . . . . .	53
4.3.3 Process change 3: Formal Manual Testing (release IR3) . . . . .	53
4.3.4 Process change 4: Code Refactoring (release IR7) . . . . .	54
4.3.5 Process change 5: Fixed 6 Week Iterations (release IR8) . . . . .	54



4.4	The content of change . . . . .	55
4.4.1	Discussion on the use of average and standard deviation . . . . .	55
4.4.2	IID, aided by code re-factoring and formal testing, improved software quality and software stability . . . . .	56
4.4.3	IID, aided by code re-factoring and formal testing, increased customer satisfaction . . . . .	59
4.5	Challenges . . . . .	65
4.6	Lessons-learned . . . . .	67
4.7	Limitations . . . . .	68
4.8	Research goals addressed . . . . .	69
5	Case Study 2. Adoption path for process changes in a greenfield project .	71
5.1	The why of change - project specific context . . . . .	72
5.2	Data gathering - available project metrics . . . . .	73
5.2.1	Quantitative measurements of success . . . . .	73
5.2.2	Quantitative measurements of success - data limitations . . . . .	75
5.2.3	Qualitative measurements of success . . . . .	76
5.3	The how of change - the new RUP execution state . . . . .	78
5.3.1	RUP Phases . . . . .	78
5.3.2	RUP work products (artifacts) . . . . .	82
5.3.3	Role set . . . . .	83
5.4	The content of change . . . . .	83
5.4.1	On-time Delivery . . . . .	83
5.4.2	On-budget . . . . .	85
5.4.3	Responsiveness . . . . .	88
5.5	Challenges . . . . .	92
5.6	Lessons-learned . . . . .	93
5.7	Limitations . . . . .	96
5.8	Research goals addressed . . . . .	96
6	A test automation strategy - an active research approach . . . . .	98
6.1	Diagnosis - Understanding the hurt areas . . . . .	100
6.1.1	Hurt area 1 - Environment configuration instability . . . . .	101
6.1.2	Hurt area 2 - Incomplete manual regression testing due to resource and time limitations . . . . .	103
6.2	Action planning - designing test strategies to address staging environment stability and lack of appropriate regression testing . . . . .	104
6.2.1	Technical hurt area 1: Environment Configuration . . . . .	105
6.2.2	Technical hurt area 2: lack of regression testing coverage . . . . .	107
6.3	Action taking and Evaluation (round 1) . . . . .	109
6.3.1	Non-functional Environment Configuration Testing . . . . .	109
6.3.2	Functional System Regression Testing . . . . .	112
6.4	Action taking and Evaluation (round 2) . . . . .	117
6.5	Reflection - Gaining visibility and support to implement technical automation techniques . . . . .	120
6.5.1	Technical impact of our testing strategy . . . . .	122

6.5.2	Organizational impact of our testing strategy . . . . .	125
6.6	Implications for practice . . . . .	129
6.6.1	Technical lessons-learned . . . . .	129
6.6.2	Contextual Organizational suggestions . . . . .	130
6.6.3	Generic Lessons-learned . . . . .	131
6.7	Limitations . . . . .	133
6.8	Research goals addressed . . . . .	133
7	Conclusion . . . . .	135
7.1	Research Goals . . . . .	135
7.2	Key Contributions . . . . .	137
7.2.1	An addition to the body of knowledge about the effectiveness of IID in environments with higher degrees of formality . . . . .	138
7.2.2	A description of the experience the organization had in making process changes . . . . .	139
7.2.3	A test strategy that successfully introduced test automation solutions to legacy applications and configuration management issues . . . . .	140
7.3	Future Work . . . . .	141
7.3.1	Bug root cause analysis . . . . .	141
7.3.2	An evaluation of RUP work products . . . . .	141
7.3.3	Longer term effects of the IID adoption . . . . .	142
A	Appendix . . . . .	144
A.1	Pettigrew’s Contextual Framework . . . . .	144
A.2	Canonical Action Research . . . . .	145
A.3	Case study 1 and 2 - Examples of Normality Test . . . . .	149
A.4	Ethics Approval . . . . .	152
A.5	Co-Author Permissions . . . . .	153
	Bibliography . . . . .	154

## List of Tables

1.1	Our GQM approach . . . . .	9
4.1	Process and context differences between process stages. . . . .	59
5.1	Case Study 2 - Project Schedule . . . . .	79
5.2	Case Study 2 - Team composition . . . . .	83
5.3	Staging Environments Descriptions . . . . .	86
5.4	Case Study 2 - Data Table for Bug Fixing Responsiveness Days to Closure (Postponed bugs excluded) . . . . .	90
7.1	Our GQM approach . . . . .	136

## List of Figures

1.1	Overview of our Research Approach . . . . .	7
2.1	Example of an IT Program - Organizational Chart . . . . .	15
2.2	Company's IT Vision . . . . .	16
2.3	Timeline . . . . .	17
3.1	The second figure of Winston Royce article "Managing the Development of Large Software Systems" [77]. . . . .	22
3.2	Craig Larman representation of an iteration [53] . . . . .	24
3.3	Jacobson <i>et al.</i> representation of the two dimensions of the RUP Process. . . . .	26
3.4	Abrahamsson <i>et al.</i> Software Development lifecycle support . . . . .	31
3.5	Dingsoyr <i>et al.</i> [26] empirical agile studies assessment and suggest goal for 2015. . . . .	32
3.6	Ken Beck representation of the inter dependencies between XP practices [14] . . . . .	33
3.7	Quality Support Practices for RUP, XP and MSF by Zuser <i>et al.</i> . . . . .	40
4.1	Case Study 1 - Projects' Data Flow . . . . .	46
4.2	Bug-density Distribution . . . . .	55
5.1	Measuring Project Success by Zachary Wong . . . . .	78
5.2	Case Study 2 - Code Activity per Iteration . . . . .	81
5.3	Bugs breakdown per Staging Environment - case study 1 and 2 comparision . . . . .	87
5.4	Bug fixing responsiveness - days to closure . . . . .	89
5.5	Bug fixing responsiveness - days to closure relative to Iteration-end . . . . .	91
6.1	Challenges encountered during the RUP adoption by Case Study 1 and Case Study 2 . . . . .	99
6.2	Example of the XML test inputs and oracle . . . . .	110
6.3	Pseudo-code for a folder permission test execution. . . . .	112
6.4	Pseudo-code for a back-end functional test execution. . . . .	114
6.5	Example of the XML functional inputs . . . . .	115
6.6	Example of a data transfer object (DTO) . . . . .	116
6.7	A web page showing the results of environmental configuration testing. . . . .	118
6.8	Environment Configuration Test - refactored Architecture . . . . .	123
6.9	Environment Configuration Test (BVT) - reporting example . . . . .	124
6.10	Environment Configuration Test (BVT) - notification example . . . . .	125
6.11	BVT Change Request . . . . .	127
6.12	Release sign-off document . . . . .	128
A.1	Andrew Pettigrew's Contextual Analysis . . . . .	145
A.2	Case study 1 - Normality Plot (Q-Q) . . . . .	149
A.3	Case study 1 - Pre-RUP histogram . . . . .	149

A.4	Case study 1 - Transition histogram . . . . .	150
A.5	Case study 1 - Partial RUP histogram . . . . .	150
A.6	Case study 2 - Normality Plot (Q-Q) for High Attention category . . . . .	151
A.7	Case study 2 - High Attention category histogram . . . . .	151

# Chapter 1

## Introduction

With the ascendance of Agile Methods it has become hard to read articles related to process improvements without hitting the words “Iterative and Incremental Development” (IID) or simply “Iterative Development.” Fundamentally **Incremental** means “add onto” and **Iterative** means “re-do” or “refine” [22]. IID is intended not only for the rework of a code base but also for the evolutionary advancement of a project’s development phases [31, 14, 52].

This thesis reports on three case studies that were conducted to understand and influence the adoption of an IID approach by a group of IT projects in a large bureaucratic government agency. The improvement efforts were motivated by several concerns identified by the business clients of those projects: poor software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays.

The primary focus of this work, the goal of the first two case studies, was to understand how effectively the process changes have been able to deal with the concerns raised by the business clients. The first case study explores how the changes affected several existing projects. The second case study explores the practices in the context of a new project. In these two case studies the effects of these changes are considered both quantitatively and qualitatively.

The third study was an action research project, with the goal of addressing two issues unresolved by the adoption of IID, identified during the data analysis of case studies 1 and 2: staging environment instability and lack of support for test automation. The term “staging environment instability” refers to downtimes experienced by stakeholders during the validation of builds at different stages of the development lifecycle. This

downtime was related to configuration management issues in the physical servers. The government agency had multiple concurrent projects. The physical staging environments integrated code from all the projects in the organization. These environments served as decision gates to stakeholders as they needed to validate the progress of projects in so far. These environments were composed of many machines, such as web servers, application servers, and database servers, among others. And these machines needed to be configured properly for all the projects using the environment at a given point in time. Since deployments were mostly manual, many configurations were overwritten or wrongly applied, leading to many hours of downtime that in turn caused delays to stakeholders' validation of builds. To deal with this issue, one project (comprised of five systems) was targeted to introduce a novel type of testing, which we termed Environment Configuration Testing. Also, management did not support any kind of test automation. Regression testing was done manually; taking many hours to validate builds. In many occasions, full regression testing was not achievable due to aggressive timelines. The same suite of applications used by the Environment Configuration Tests was used to introduce an automated regression testing strategy.

In reporting on these studies, this thesis makes three main contributions. The first is an addition to the body of knowledge about the effectiveness of IID in environments with higher degrees of formality. The second is a description of the experience the organization had in making those changes, which is hoped to be of practical benefit for others undergoing similar process improvement efforts. The third is a description of a test strategy that successfully introduced test automation solutions to legacy applications and addressed configuration management issues. It is expected that this test strategy can help other large multi-project organizations that face challenges in regards to legacy applications and configuration management consolidation.

In this first chapter, the motivation and the scope of this work are described. It

contains a presentation of the chosen research methodological approach used to address the key research goals and it highlights the key contributions provided by this work.

## 1.1 Motivation for Studying the adoption of IID practices

An investigation of IID practices in an industrial setting merits examination for various reasons. According to Larman, IID - or its best well-known subset which is Agile Methods (described in Chapter 3) - is often seen as the innovative response to traditional engineering practices [52]. In regards to innovation, Everett Roger's Technology Adoption Curve [76] has five categories of adopters of innovation based on a normal distribution: innovators, early adopters, early majority, later majority, and laggards. The "Chasm" is defined by Geoffrey Moore as the gap between acquired and deployed technologies [63], between early market success with visionaries (innovators and early adopters) and the mainstream acceptance by more pragmatic adopters (early and later majorities) [58]. Gaps represent transitional difficulty in moving from one group to the next.

Many believe that IID, and consequently Agile Methods, have crossed this chasm and have hit the mainstream of application development organizations [84, 8]. Others believe that "we're not there yet" [30]. Perhaps these contradictory views are related to the fact that much of the evidence about Agile adoption comes either from: (1) projects in the "Agile sweet spot" [50], (2) from medium to large companies developing software products or providing software consultancy, or (3) from sources that have been decontextualized. In broader terms:

(1) The Agile sweet spot can be summarized as small co-located teams, developing low to medium safety systems in a friendly management environment [50, 14].

(2) The evidence about Agile adoption in medium to large organizations comes mostly from experience reports from software companies like Microsoft [54], Salesforce [35],



Yahoo! Music [21], Sabre Airline Solutions [65], BMC Software [13], CGI [38]. Literature from non-software companies comes usually from small projects of short duration [26], from companies reporting on specific technical practices like pair programming, test-first programming, and XP in education [46, 38, 26], or from companies that sell both software and non-IT products [79, 83, 18].

Software development in a company that sells software is a very different value proposition than in a company that develops software to support other processes. A value proposition is defined as a clear and succinct statement that outlines to potential clients and stakeholders, a company's (or individual's or group's) unique value-creating features [55]. In software companies, IT projects are the heart of their profitable earnings and an active part of their value proposition [55]. As such, software companies need to position themselves as innovators and early adopters of technology. On the other hand, in non-IT companies software projects are operational costs, a "necessary evil" [32] in support of other processes that result in profitable earnings and value propositions. In this context IT managers are faced with more pragmatic choices of "*tried and true*" software methodologies [40].

(3) Decontextualization happens when relationships with the natural/original environment, where the process changes were introduced, have been dismissed from conclusions [51]. "*Most often, what is left out is the exact context where the idea, concept, or process applies, or the context in which it was created and found useful*" [51]. A strong example of decontextualization is how Royce's Waterfall model has been abused and taken out of context. In his keynote speech at the 31<sup>th</sup> International Conference of Software Engineering, Steve McConnell referred to Royce's decontextualized Waterfall model representation (Figure 3.1) as: "*I am not bad, I am just drawn this way.*"

The contradictory views about IID practices in the mainstream of development are well summarized by Pettigrew's views on change: "*there is the problem of perspective.*"

*Where we sit not only influences where we stand, but also what we see. Give history and social processes the chance to reveal their untidiness” [67].*

Another reason to investigate IID practices is summarized in the work Dingsoyr *et al* [26]. These authors provide a recent roadmap of empirical research in IID practices calling for more industrial collaboration through Action Research and more knowledge on how Agile principles, such as IID, work in different contexts. This understanding is needed in order to find how to apply Agile principles most effectively in different real-life situations. But, according to Krutchen [50], in real-life situations, projects’ contextual attributes such as: size of system, criticality, age of system, rate of requirements change, business model, stable architecture, team distribution, and governance should influence which practices are best suited for a particular context.

In this thesis, the IT department of the government agency under study was composed of over 200 professionals. This department supported and developed complex systems that interfaced with legacy applications in a governance model faced with constant and strict conformance to rules and regulations. This environment resulted in formal decision-making processes, as also experienced by Berger in another study of a government agency [16]. This culture and context led managers to adopt IID through a methodology with a higher degree of “ceremony” and formality [53], the Rational Unified Process (RUP). The RUP framework has the ability to scale up and down with optional support for higher degrees of formality and documentation [53].

It is in this context that an empirical study of projects that adopted IID is presented. Such an understanding is particularly important to pragmatic managers that prefer to adopt processes that have been successfully implemented by others to reduce the risk of failure [40] - later adopters.

## 1.2 Overview of Research Goals and Methodological Approach

This thesis relates the findings of case studies conducted with a large government agency. In conformance with its bureaucratic and private nature, a Researcher-Client Agreement (RCA - see definition in Appendix A.2) was created, approved, and signed in August of 2007. From a research perspective, the original research goal was to only analyze how the concept of iterations affected these areas of concern. However, the introduction of process improvements in real life is a complex problem that involves many simultaneous factors that cannot be responsibly detached from each other.

The client was interested to find out how, in more concrete terms, they benefited from the roll out of the IID practices. But in order to keep our research goals focused, we narrowed the study to investigate how the adoption of IID affected the main areas of concern from business clients that drove this agency to abandon Waterfall practices.

Since our focus became to analyze how the IID adoption changed certain existing organizational problems, we present our study following Pettigrew's contextual framework for studying organizational change [68]. In Pettigrew's views "*causation is neither linear nor singular*", and in order to analyze change, we need to consider how three important variables interact with each other: **context** (the "why" of change), **process** (the "how" of change), and **content** (the "what" of change). Further details about Pettigrew's framework are provided in Appendix A.1).

Figure 1.1 provides an overview of the methodologies used in our research.

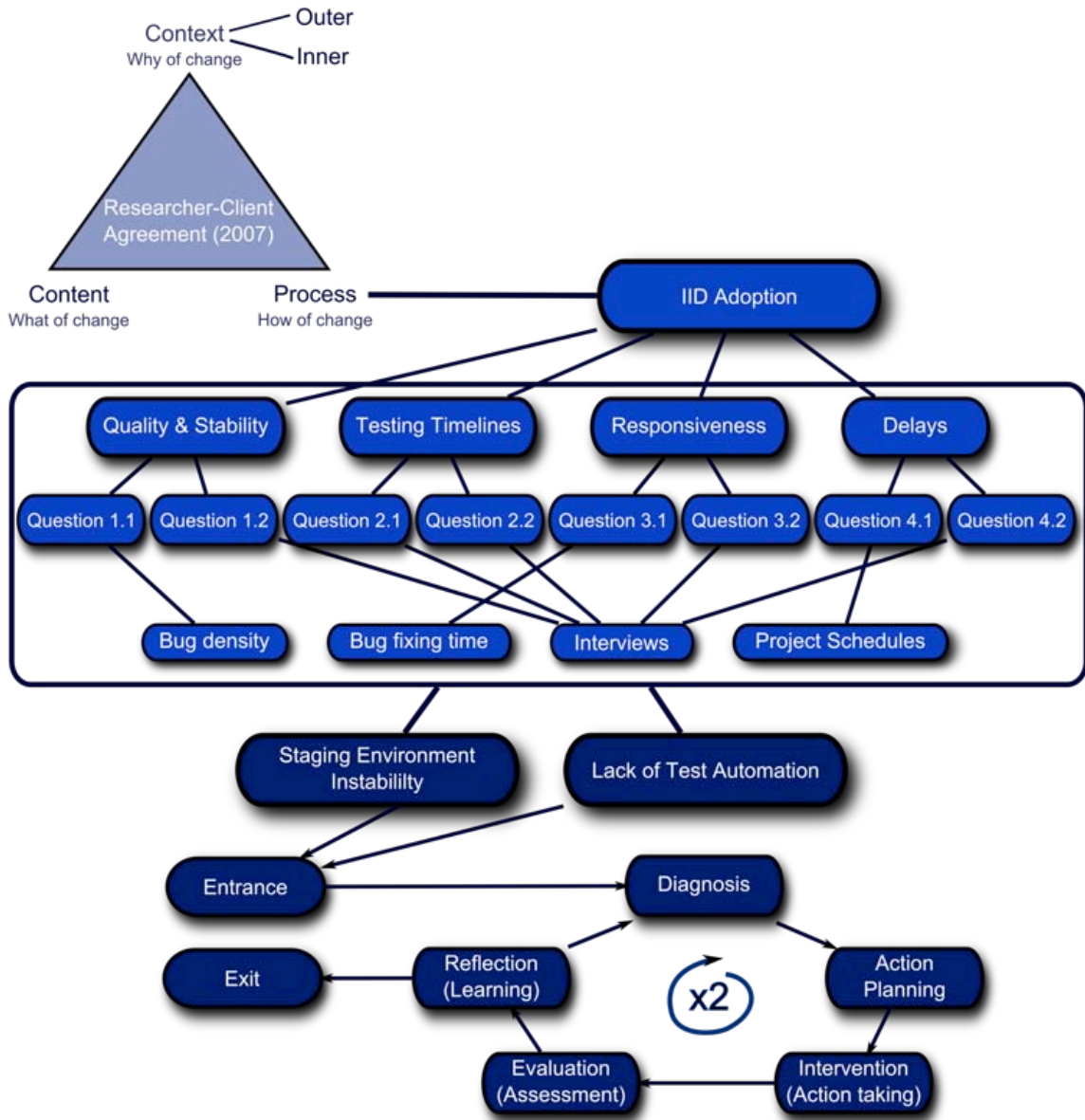


Figure 1.1: Overview of our Research Approach

For the purposes of this thesis, context, process, and content are defined as follows:

**Context** - the “why of change” was defined through open-ended exploratory observations of this government agency IT department supported by field notes and informal interviews with key employees and consultants. This was facilitated by an on-site office being granted to the author of this thesis. Further details will be presented in

Chapter 2, Organizational Context. Context specific to the projects involved in the case studies will be presented in their respective case study chapters.

**Process** was defined as the introduction of IID practices.

**Content** was defined as the concern areas that led this organization to adopt IID by conducting formal interviews with key business clients and consultants (more details in Chapter 2, Organizational Context).

In particular, the primary goal of this thesis is to investigate how IID adoption affected software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays. Pettigrew’s contextual framework provides a way to identify contextual information about the IID adoption, in order to minimize the possibility that unknown external factors influenced the presented results.

The Goal Question Metric approach (GQM) proposed by Basili *et. al* [86] was adopted in order to formalize the above research goals and to find appropriate measurements to answer them. According to Basili *et al.* the result of the application of the GQM approach is *“the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data”*. This model has three levels: conceptual level (Goal), operational level (Question) and the quantitative level (Metric). The rectangular portion of Figure 1.1 provides an illustration of the relationship between the three levels. Table 1.1 illustrates how the main set of research questions was translated into appropriate measurements and rules.

**Relationship between case studies.** The set of questions and metrics presented in Table 1.1 was applied to a suite of existing projects that transitioned from waterfall to IID, as presented in case study 1, Chapter 4. The interviews’ results of this case study compelled us to also apply the same set of questions to a project that followed IID since inception, presented in case study 2, Chapter 5. Whenever possible, we provided comparative analysis between projects from case studies 1 and 2. These comparisons

Goal 1	
Purpose	Improve
Issue	Poor and unstable
Object (process)	quality of releases
Viewpoint	IT Management and Business clients
Question 1.1	How was the quality and consistency of quality of the releases before IID was introduced?
Metrics	Bug-density Standard deviation Subjective views of business clients
Question 1.2	How was the quality and consistency of quality of the releases after IID was introduced?
Metrics	Bug-density Standard deviation Subjective views of business clients
Goal 2	
Purpose	Improve
Issue	Rushed or insufficient
Object (process)	testing timelines
Viewpoint	Business clients
Question 2.1	How were releases tested before IID was introduced?
Metrics	Existing project schedules subjective views of business clients
Question 2.2	How were releases tested after IID was introduced?
Metrics	Existing project schedules subjective views of business clients
Goal 3	
Purpose	Improve
Issue	Long wait and red-tape
Object (process)	bug-fixing requests
Viewpoint	Business clients
Question 3.1	How long did it take for bugs to get fixed in existing projects that transitioned to IID?
Metrics	bug-fixing responsiveness in days subjective views of business clients
Question 3.2	How long did it take for bugs to get fixed in new projects that started development using IID?
Metrics	bug-fixing responsiveness in days subjective views of business clients
Goal 4	
Purpose	Decrease
Issue	Delays
Object (process)	Release delivery
Viewpoint	IT Management and Business clients
Question 4.1	Were releases delivered on-time prior to IID?
Metrics	Project schedules subjective views of business clients
Question 4.2	Were releases delivered on-time after the IID adoption?
Metrics	Project schedules subjective views of business clients

Table 1.1: Our <sup>9</sup>GQM approach

revealed that a common set of issues were still unaddressed by the adoption IID practices due to the existing contextual constraints: staging environment instability and lack of support for test automation. *“The past is alive in the present and may shape the emerging future”* [67].

A test strategy to address the two issues stated above, found during the empirical analysis conducted in case studies 1 and 2, constitute the the secondary focus of this thesis. Specifically, a canonical action research (CAR) project was conducted with one project (comprised of five systems). The CAR project enabled us to provide a further contribution to our industry partner and to the research community, by investigating the following research questions:

- What artifacts or processes can alleviate issues with configuration management of the staging environment?

In companies with concurrent projects and legacy systems, not all teams are allowed to utilize test automation and automated deployment techniques in all staging environments. But they all still share these environments and need to validate server configurations to run their applications. This was the case of the projects under study in this thesis. Due to differences in delivery schedules, teams deployed builds to staging environments at different times (manually or through scripts), leading to staging environment configurations being overwritten or lost, and code from other teams to fail after being tested and deployed.

- What artifacts or processes can support projects, specially legacy ones, in gaining support to invest time and money in test automation?

It is common for organizations to skip automated testing due to budget or time constraints [17], as they are not initially as important to the stakeholders as the implementation of extra features. In the projects from case studies 1 and 2, aggressive

timelines drove upper management to see test automation as a peripheral activity when compared to the tangible delivery of scheduled tasks and bug fixes. While current literature provides many experiences and techniques to introduce test automation into software during construction, many of us at one point in our career as software engineers will work with legacy applications in adaptive maintenance modes that lack automated tests.

The bottom portion of Figure 1.2 illustrates the Canonical Action Research (CAR) principles, as defined by Davidson *et al.* [24], used in our research (more details presented in Chapter 6). These principles attempt to marry rigor and relevance of action research conducted in system-related research, “*although they can be hard to achieve in a single CAR project*” [24]. The CAR process is iterative and collaborative. It focuses on both organizational development and the generation of knowledge. A description of the five CAR principles is provided in Appendix A.2.

The results from case studies 1 and 2 provided preliminary diagnostic information. The issues were then categorized into two sets: technical or contextual. Two CPM cycles were accordingly conducted. The same qualitative techniques used in case studies 1 and 2 were then used to gain further knowledge about the set of common problems. The use of well-known testing criteria for effective testing strategies [37, 33] supported the definition of the intervention and evaluation steps.

### 1.3 Key contributions

The work described in this thesis is covered in three publications [71, 69, 82] and one workshop presentation [70].

The data collected and analyzed from the three case studies allows this thesis to make three key contributions. The first is an addition to the body of knowledge about



the effectiveness of IID. It confirms past findings [14, 81, 42, 44] with new data from an environment with higher degrees of formality. These findings include improvements in software quality and stability, and limited improvements to bug-fixing responsiveness in a suite of existing projects. It also highlights the differences in benefits from adopting IID practices in existing projects versus following IID practices since project inception. A study of a project that followed IID since inception showed that this project avoided the quality and stability issues seen in former waterfall projects, provide better bug-fixing responsiveness than existing projects that migrated to IID, and supported key managerial decisions that lead to on-time and on-budget delivery.

The second contribution is a description of this government agency's challenges and lessons learned from the process of adopting IID practices. These lessons learned can be of practical benefit for others undergoing similar process improvement efforts. In particular, the IID adoption included challenges such as decreased software quality and stability during transition (as also experienced by others [35, 44, 83]) and hard paradigm shifts for business clients in regards to iterative delivery and testing. It provides some practical advice such as minimizing the amount of other changes introduced during the transition stages (as also suggested by Jacobson *et al.* [43]).

The third contribution is the description of a test strategy that successfully introduced test automation solutions to legacy applications and alleviated configuration management issues. We accomplished this by delivering:

- A suite of back-end NUnit functional tests that used XML-driven data inputs leveraged the maturity of maintenance applications. This strategy was able to introduce a change in the perspective of the cost-value relationship of test automation for legacy applications.
- A suite of automated environment configuration tests that verify staging environ-

ments on demand. This test suite has helped this government agency identify many deployment dependencies and assist teams to resolve configuration issues in staging environments in a timely manner. It also abstracted environment configuration management into a live and evolving test suite that shows failure and it is easily maintainable.

It is expected that this test strategy can help other large multi-project organizations that face challenges in regards to legacy applications and configuration management consolidation.

## 1.4 Thesis Overview

The following chapters describe in more detail the implementation of our research approach.

In Chapter 2 we describe the inner and outer contexts of our industrial partner.

In Chapter 3 we provide the background and context around IID practices. We provide an overview of IID methodologies, in particular the Rational Unified Process and Agile Methods. We provide a discussion of existing IID adoption strategies. We also share related literature relevant to companies transitioning to iterative methodologies in large organizations.

In Chapter 4 we present our first case study with a set of existing projects that transitioned to IID. In Chapter 5 we present our second case study with a project that followed IID since inception. In Chapter 6 we define the set of common issues between case studies 1 and 2 and present our Canonical Action Research project.

And finally we conclude our work in Chapter 7 going back to our research questions and demonstrating how we addressed and complied to our research approach. We discuss some generic lessons-learned and suggest some future work.

## Chapter 2

### Overall Organizational Context

In this section we provide some contextual information about our industrial partner. The experience related in this thesis comes from case studies about projects in an oil & gas government agency. Jutta Eckstein states that *“large is not a well-defined magnitude, and neither is the largeness of a [project] team”* [28]. She uses five dimensions to categorize the largeness of a project: scope, people, time, money, and risk. These five dimensions are interrelated. People and scope are considered first-order dimensions as they exist as a first-order consequence from a company’ requirements and constraints. The remaining variables are usually derived from these first-order dimensions.

**People.** This government agency has been in business for more than four decades and has a workforce of over 900 employees. The IT department has approximately 10 different IT programs (portfolios of many projects bundled under a common business area domain - Figure 2.1) with a total of 191 IT professionals, not including supportive administrative staff. This thesis studies four projects (seven software systems) from the largest IT Program in this agency with approximately 50 IT professionals.

**Scope.** In order to preserve the anonymity of this agency we can only report that their IT business focus is to develop, enhance and maintain contemporary systems to enable timely responses to requests from the oil & gas industry. These systems are considered contemporary as they enable realtime validation of complex government rules and regulations during on-line submission of data and pertinent attachments. They also automatically generate e-mail notifications and status reporting to request submitters. In broad terms, these systems support the automation and re-engineering of internal business process flows. These flows were previously supported by legacy applications,

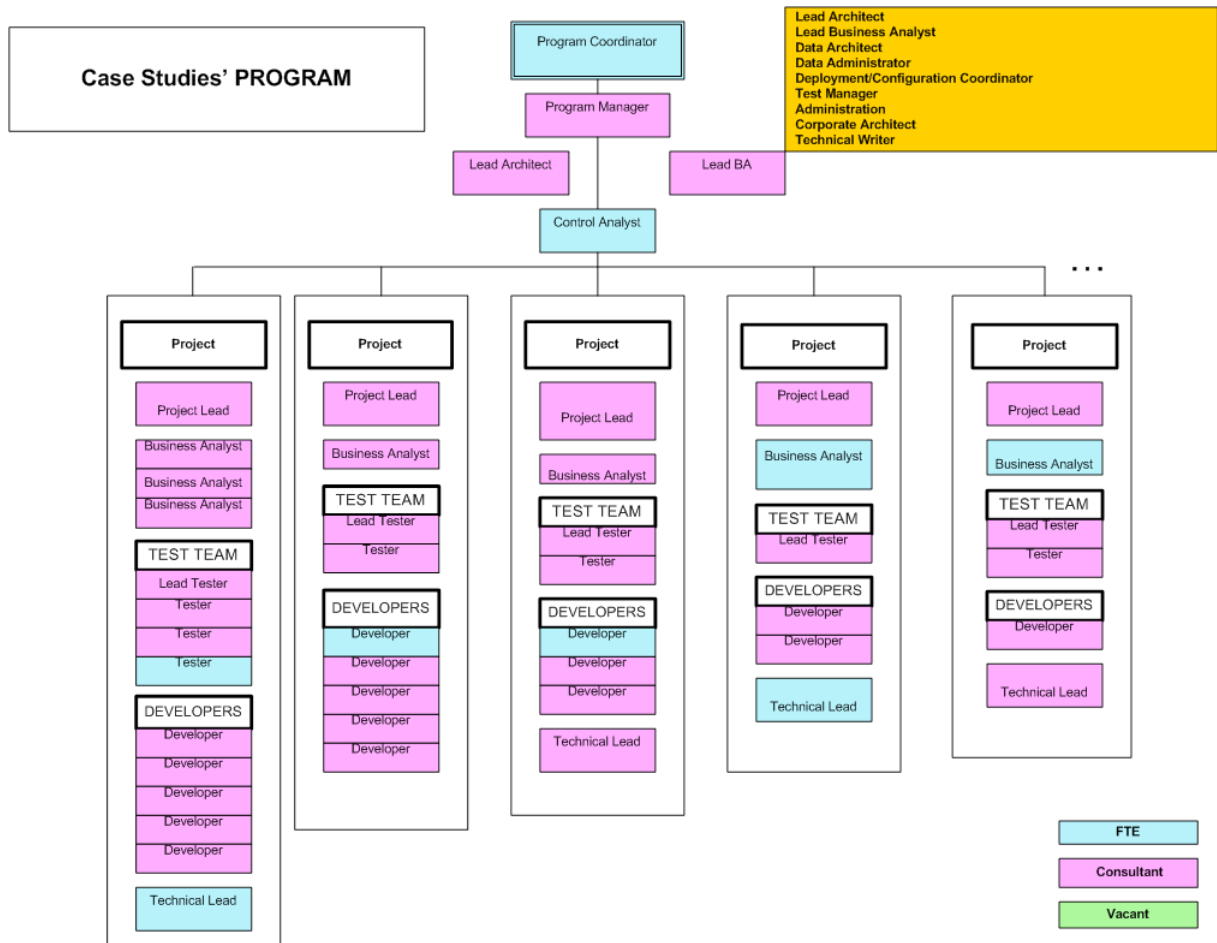


Figure 2.1: Example of an IT Program - Organizational Chart

like Mainframe, Paradox, and Excel spreadsheets amongst others.

The IT Program under study supports the internal business processes responsible for receiving, sorting, validating, processing (accepting/declining), and publishing tens of thousands regulatory requests and regulatory objections per year from the oil & gas industry. The main business driver behind this IT program is to cut down the time it takes to process an industry request from end-to-end.

Figure 2.2 illustrate this IT program’s long-term vision. The project names presented in a blue indicate new projects that will be added during that Generation. The case studies presented in this thesis come from projects: X, Y, Z and A. Figure 2.2 was created sometime after Project Z was delivered. The client representatives for these projects were

the internal business area leaders (business clients). These leaders represented the interest of their internal and external users.

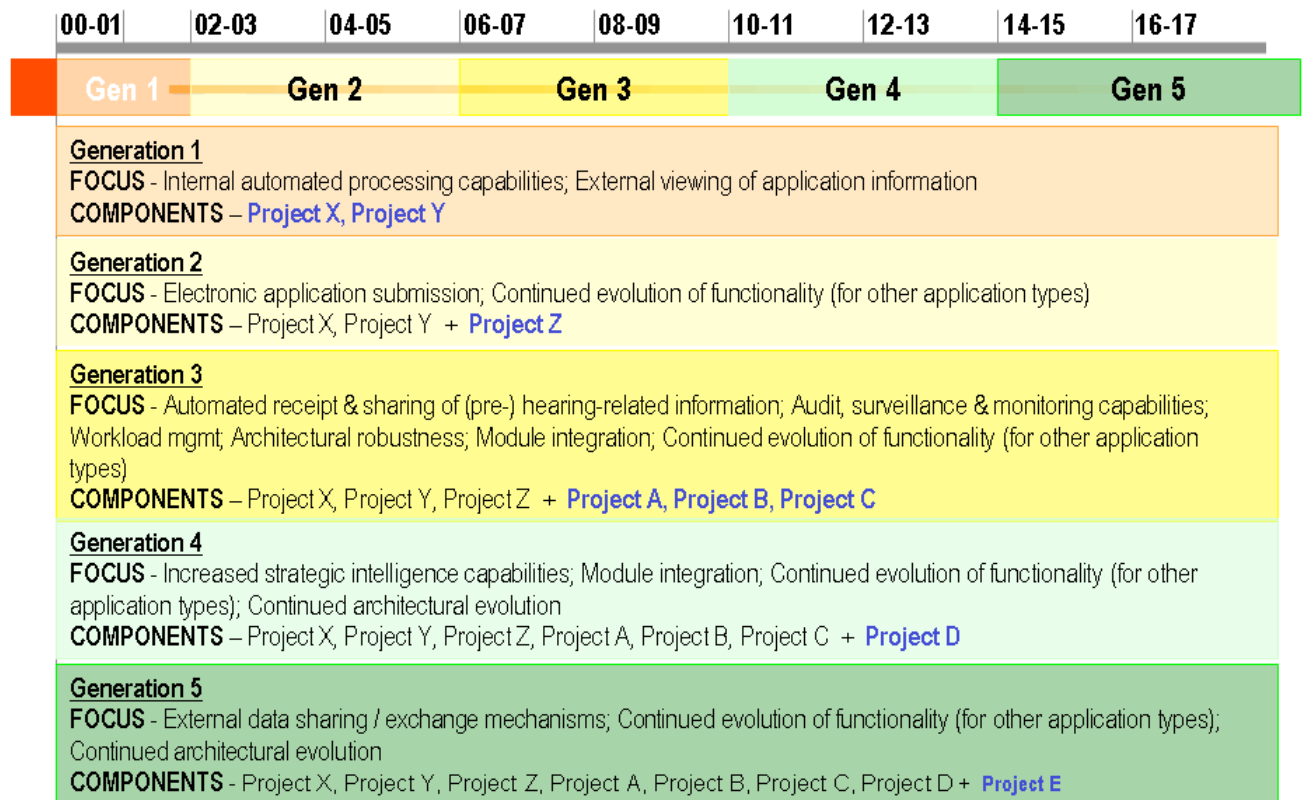


Figure 2.2: Company’s IT Vision

**Money.** The 2007-2008 fiscal IT Plan included 21 initiatives: eight IT programs, seven individual projects, one mainframe retirement initiative, and five infrastructure projects. Projects were categorized in the following budgeting brackets: budgets of over \$1.5 million dollars, budgets between \$500,000 to \$1.0 million dollars, and budgets bellow \$500,000 dollars.

Although we were not allowed to view exact dollar figures, the projects under study had budgets in the millions of dollars and a user base of over 11,000 users.

**Time.** Projects ranged in duration from one year to over two years for a first production release.

**Risk.** The suite of projects under study is considered innovative and unconventional in the context of government systems. Specifically, project Z is considered a “state of art” government application with many other government agencies interested in implementing a similar concept.

## 2.1 Inner Context

As Figure 2.1 demonstrates, this agency has a hierarchical structure supported by chains of command-and-control. Its Human Resources department offers competitive wages, benefit packages, and career development plans for employees. This agency was recently voted one of the top employers in Alberta. Of importance is the fact that 54% of professionals in the IT Department (103 contractors) are external consultants hired to work in-house for periods of one-to-three years. Figure 2.3 illustrates some of the key contextual milestones of interest in our study.

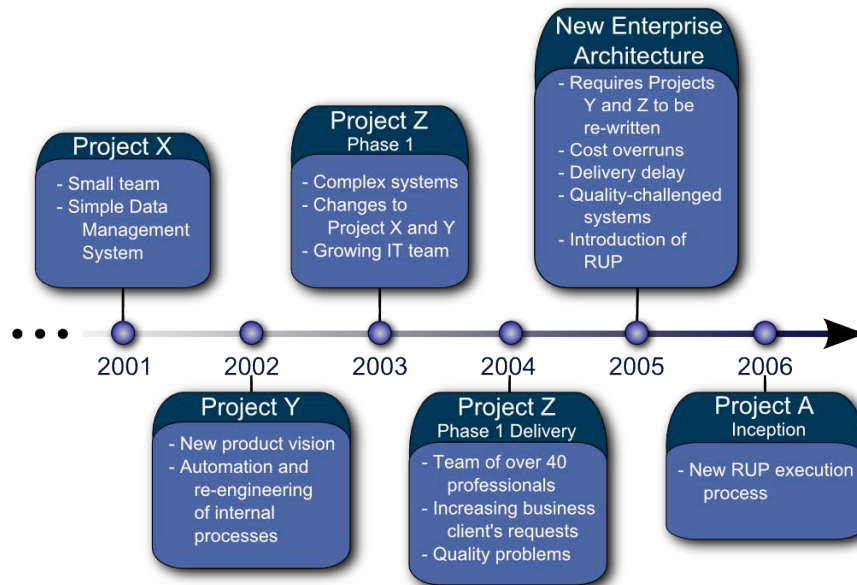


Figure 2.3: Timeline

Development for the first release of Project X (client-server application) and Project Y

(external publishing site) started in early 2001 with a small group of four to six developers, two business analysts and one project manager. Project X took approximately one year to deliver its first release to production. The original goal was to develop a simple system to provide a central data management point for the business clients. The development team followed a waterfall development approach: gather all the system requirements, develop the entire application, and finally hand off to the business clients for testing and approval. The first production release of Project X was well received by the business clients. This was supported by the good teamwork and presentations of screen mock-ups during requirement gathering sessions.

The business clients changed their simplistic goals to a more ambitious vision after the first release of Projects X and Y were put into production (user acceptance testing was only performed at the end of the development cycle, close to production). The new plan was to create a one-of-a-kind workflow system to provide quicker turn-around processing times for digital submission of data.

Late in 2002 most of the original team had left the company and a new team started the development of Project Z (second on-line application) with many enhancements to Projects X and Y to support the submission and validation of the industrial electronic data. By early 2004 the small team had evolved from four developers to a larger team of approximately 15 developers. There were a total of 40+ team members (including business analysts, testers, project managers, business clients, architects, and technical writers) operating in a single-pass waterfall approach with fixed scope and fixed time projects, the latter being of supreme importance due to external economical pressures (see next section). Requirements were documented in many formats, such as Microsoft Word and Excel spreadsheets, and did not follow pre-defined templates. At the same time an IT enterprise group was founded, comprised of senior architects, to support and standardize the technical aspects of systems' development.

Development was fast-paced. Many of the project-specific releases were rushed or delayed due to fixed delivery dates that forced unreliable acceptance testing schedules on the business clients. This was also the first time that business clients were involved in the development of a large application. IT management cut the lines of communication between developers and business clients (more details presented in Chapter 4). The result was poor software quality, low team morale, and many “fixer-up” releases to address unhappy business clients needs. IT pointed to business clients for not properly testing the systems or not specifying the requirements correctly. Business clients pointed to IT for unrealistic testing schedules, wrong time estimates and poor accountability. This resulted in trust issues.

Business clients communicated to upper management the need for better accountability (on-time delivery), better quality, better stability and more reliable testing schedules. At the same time the burned-out development team voiced the need to implement a process with more automated practices, better business involvement and faster response to changes.

IT Management, motivated by the IBM Rational Unified Process (RUP), made a small number of process changes over a period of approximately 13 months, to a suite of existing projects (Project X, Y and Z - details presented in Chapter 4) and, afterwards specified a process to be followed by new projects (Project A - details presented in Chapter 5). A part-time RUP process engineer from IBM was contracted to engage expert help as early as possible, as suggested by Megan [85], Jacobson *et a.* [43], and Fry and Greene [35]. The process engineer’s role was to support the customization of the RUP framework. The specific process changes made, the results, and challenges of those changes are discussed in Chapter 4 (case study 1), Chapter 5 (case study 2), and Chapter 6 (case study 3).



## 2.2 Outer Context

The oil & gas industry is the economic heart of Alberta. Technological advancements have allowed companies to drill into reservoirs that were unreachable or protected by the government. These technological advancements forced the government to review and implement different regulatory rules for the oil & gas industry.

In the past years Alberta has experienced an “*Oil boom*” [41] with the ascendance of the development of the Fort MacMurray oil sands. Time has been critical for many companies and this has forced government agencies to streamline internal processes to support this economical growth.

# Chapter 3

## Literature Review

### 3.1 Background and Context

Although Agile Methods have the concept of incrementally growing a system through iterations at the core of their engineering practices [15], and are often referred to as synonyms of IID, they are only a subset of IID methodologies [53]. IID roots can be traced back quite a few decades. These practices were inspired by the “plan-do-study-act” (PDSA) cycles proposed by a quality expert at Bell Labs in the 30’s, Walter Shewhart [52]. IID was also considered a major contributor to the success of the X-15 hypersonic jet project in the 50’s, seeding NASA’s early 60’s IID Project Mercury [53]. As a matter of fact, in his famous 1970 article titled “Managing the Development of Large Software Systems” [77] Winston Royce refers to the figure that coined the Waterfall process (reproduced in Figure 3.1) as:

*“I believe in this concept [single-pass], but the implementation described above is risky and invites failure... The Testing phase which occurs at the end of the development cycle is the first event for which timing, storage, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable.”*

Royce’s recommendation was to do the single-pass process twice. He suggested that if a computer program is being developed for the first time, create a pilot model, learn from it and *“the version finally delivered to the customer for operational deployment is actually the second version insofar.”*

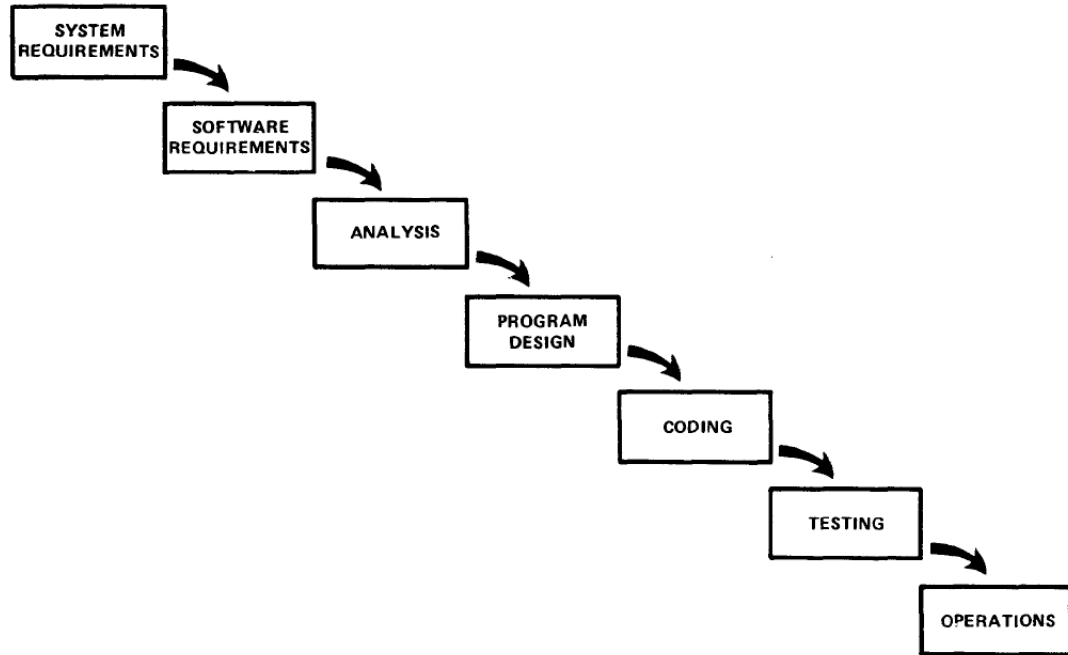


Figure 3.1: The second figure of Winston Royce article “Managing the Development of Large Software Systems” [77].

The promotion of Royce’s model inspired waterfall-promoting standards like the United States Department of Defense (DoD) 2167A. In turn, these standards influenced commercial software engineering projects for many years with varying degrees of success, including the company under study in this thesis.

Studies on software projects success rates have been conducted by the Standish Group, an IT Value Research organization [39] for a few years. The results of their first report in 1994 showed alarming failure rates of over 70%, resulting in the report be named the “CHAOS Report.” Many scholars rightfully doubt the validity of these findings [36, 45] as data collection methods and research design information have not been released to the general public. Regardless, these reports have been a source of citation for many studies on the topic of software crisis and the need for new software engineering practices.

Some years later, in 2001, the term “Agile Methods” was coined by a group of 17 leading developers and proponents of iterative and incremental light-weight engineering

practices [80] with the creation of the “Agile Manifesto” [15]. Agile Methods “*constitute a space somewhat hard to define [...] they are a family of approaches and practices for developing software systems*” [51]. Kruchten [51] defines Agile Methods in two ways: (1) by enumeration or (2) by some predicate like compliance to the Agile Manifesto. Here are some enumerative examples of Agile Methods: Extreme Programming [14], Scrum [54], Lean [73], and Agile Rational Unified Process (RUP) or OpenUp [9] “*(though some would claim RUP to be anti-agile)*” [51]. Of interest is also Robert C. Martin “dX” method, which streamlines the RUP process to a minimal set of workproducts [57] mimicking the principles of XP as tailored RUP activities. “*Apart from meaning a small change, dX is XP upside down*” [7]. The highlights of the Agile Manifesto are to value:

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

*“That is, while there is value in the items in the right, we value the items on the left more”* [15].

## 3.2 Iterative and Incremental Development Methodologies

As mentioned in the previous section, iterative and incremental development is the heart of its most well-known subset, Agile methods. The Rational Unified Process (RUP) is not seen as a “*particularly Agile software development method, but it can be one*” [7].

IID builds software with a development lifecycle composed of iterations. An iteration is illustrated in Figure 3.2. Iterations act like mini-projects delivering activities in all four basic software engineering disciplines: requirements, design, implementation, and test.

Modern IID methods suggest that the length of iterations should range from one to six weeks long [14, 53]. Iterations should be timeboxed, which means that the end-date of an iteration is fixed and cannot be changed. If planned activities cannot be delivered within an iteration, the iteration end-date is kept unchanged and under-estimated activities will be deferred to the next iteration. Each iteration delivers integrated and tested code, but most iteration releases are non-production releases. Each iteration delivers an increment of the system, which grows in different areas depending on project drivers such as: business value, risk, and architectural components.

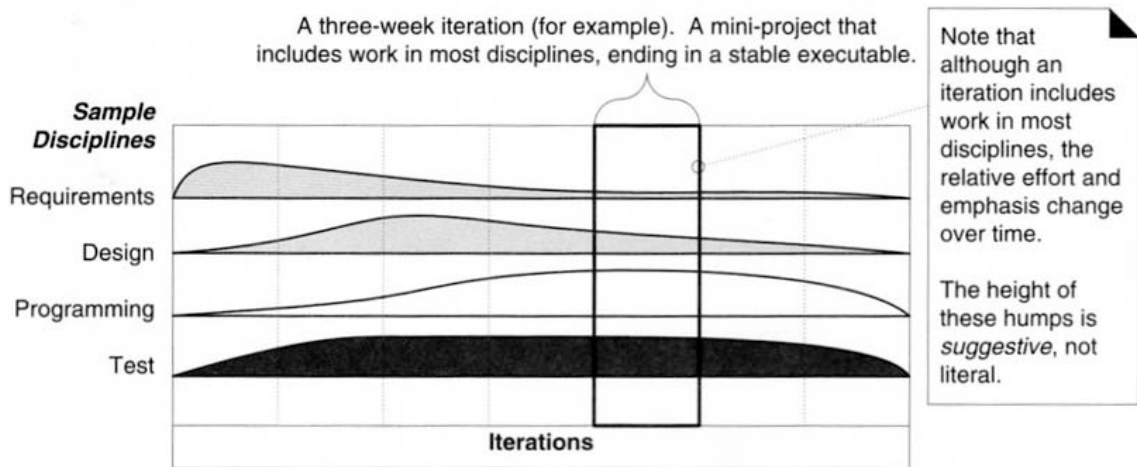


Figure 3.2: Craig Larman representation of an iteration [53]

Agile methods are empirical. They are based on frequent evaluation and response to change [53]. They are defined by their maneuverability and the motto to embrace change [14]. The RUP is viewed as a semi-defined process [53], defined by the list of customizable process-related artifacts, activities, workers, and workflows.

In the next sections we provide an overview of the RUP and Agile methods' principles.

### 3.3 The Rational Unified Process

Since the government agency under study adopted IID through the Rational Unified Process (RUP), we found it useful to the reader to provide some background information on the RUP framework.

The Rational Unified Process (RUP) is a refinement of the iterative and incremental Unified Process (UP). RUP was coined in 1998 by Rational Software as an upgrade of the Rational Objectory Process 4.1 [66]. RUP is a configurable process, and due to its configurability it is viewed as a semi-defined process [53].

RUP is a process framework that should be customized to fit specific project needs [66, 53]. RUP *“is not a concrete process description for one particular project”* [53]. Jacobson *et al.* describe RUP in a nutshell as: use-case driven, architecture-centric, iterative, and incremental [43]. *“A use case is a piece of functionality in the system that gives a user a result of value. Architecture is a view of the whole design with the important characteristics made more visible by leaving the details aside. Iterations are mini-projects, controlled and carried out in a planned way”* [43].

Conceptually, RUP aims at building teamwork, dealing with project risks early in the process to deliver business value using short, timeboxed iterations. The process can be visualized in two dimensions as shown in Figure 3.3: organization along time and along content. The horizontal axis represents the cycles, phases, iterations, and milestones. The vertical axis represents which artifacts (*what*), activities (*how*), workers (*who*), and workflows (*when*) are part of the structure of the process [66] - the four Ps: people, project, product, and process [43].

The RUP software lifecycle is broken down into four phases:

- Inception

This phase should be ideally short. It includes the definition of the project vision:

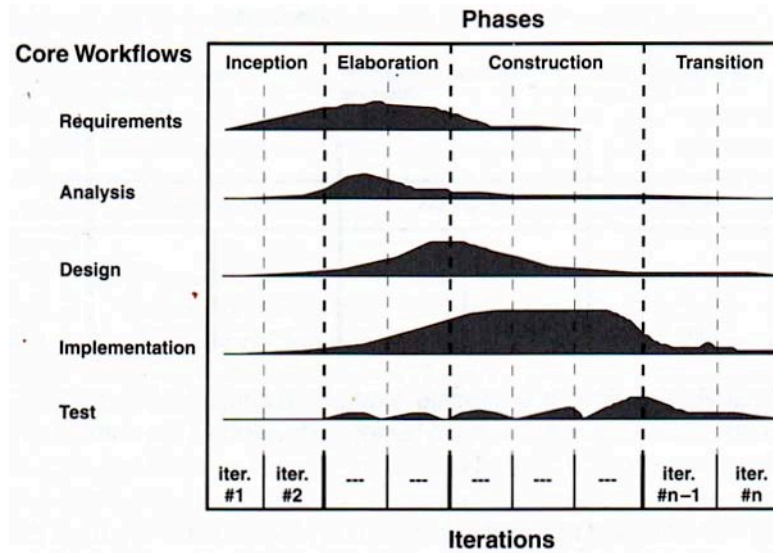


Figure 3.3: Jacobson *et al.* representation of the two dimensions of the RUP Process.

key requirements, features, and constraints. Key project risks should be analyzed by an initial risk assessment with the creation of a *Risk List* document. Some projects may decide to create a business model to define the existing business workflows and a business case to define the context, success criteria, and budgeting. Rational specifies that 10% to 20% of use-case models should be completed [66] while others mention that a requirements workshop and the detailed specification of 10% of the most architecturally influential requirements should be enough [53]. The milestone for this phase is called the Lifecycle Objectives Milestone.

- Elaboration

This phase focuses on eliminating the highest risks of the project and developing the core architecture of the system. The elaboration phase includes programming work and is considered the most critical of the four phases. At this point a *Development Case* should be created that tailors the RUP process for the specific project's needs. Teams are encouraged to select the minimal set of work products necessary [53, 9] and to use UML models instead of large documents [66, 10]. This phase is marked

by creativity, discovery, collaboration and **risk management**, which proved to be really effective for the projects presented in Chapter 5. Some suggestions include the creation of: 80% of use-case models, a software architecture document, an executable prototype, a risk list document, and a preliminary user manual. The milestone for this phase is called the Lifecycle Architecture.

- Construction

The rest of the software system is built during this phase, based on the architectural decisions made during elaboration. Software reuse is fostered by a component-based architecture, which, in turn, should be flexible and adaptive. Iterations should be tested in integrated system environments. The milestone for this phase is called the Initial Operational Capability.

- Transition

RUP takes a phased-release approach. A release candidate is chosen and released for “beta testing” to a selected group of users. The focus here is on deployment and achieving self-supportability. It in this phase that the system is fine tuned. User-oriented documentation such as help files should be present to support the initial production releases. At the end of transition the stakeholders need to answer questions like: *“is the user satisfied? Are the actual expenditures versus planned expenditures still acceptable?”* [66]. The milestone for this phase is called Product Release. It is important to note that a project can have many transitions. It is important not to confuse the word phases with the Waterfall phases. In RUP phases are stages where the team is performing mostly activities related to a particular lifecycle goal.

As shown in Figure 3.3, RUP defines a set of disciplines such as: requirements, analysis, design, implementation, project management, and testing. Each discipline has



related workproducts and workers (roles). A common criticism of the RUP framework is the large number of workproducts specified by this framework. On the other hand, most of the workproducts are optional and should be tailored to project-specific needs through a Development Case document. The same applies to the roles of workers. These roles should be seen as “hats” team members put on to consistently deliver certain activities. A team member can act in more than one role and a role should be fulfilled by more than one team member.

There are six overarching best practices implemented and supported by RUP:

1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software

The RUP process can be supported by a suite of tools, originally developed by Rational, currently sold and supported by IBM. The use of these tools is optional. *“The process-tool relationship is another chicken-and-egg problem. Which one comes first? Successful development of process automation cannot be achieved without the parallel development of the process framework in which the tools are to function”* [43]. The RUP process integration with tools includes: source control repository (ClearCase), requirements elicitation (RequisitePro), modelling tools (Rational Rose, XDE), testing tools (Rational Functional Tester, Visual PureCoverage, PerformanceStudio), task and bug tracking (ClearQuest), and document generation (SoDA) [20].

### 3.4 Agile Methods

The goal of this section is not to describe in detail the practices of Agile methods, but to inform the reader of the commonalities and key features. It is to describe their key strengths and weakness.

The term Agile methods was officially coined with the creation of the Agile Manifesto in 2001. The Agile Manifesto [15] contains the twelve overarching principles driving Agile methods:

1. “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity - the art of maximizing the amount of work not done - is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. ”

Abrahamsson *et al.* published a review of Agile methods and practices in 2002 [7]. The main research goal of these authors was to answer the question: “*what makes a development method an Agile one?*” They concluded that this is the case when software development is:

- “Incremental (small software releases, with rapid cycles),
- Cooperative (customer and developers working constantly together with close communication),
- Straightforward (the method itself is easy to learn and to modify, well documented), and
- Adaptive (able to make last moment changes).”

Figure 3.4 presents Abrahamsson *et al.* findings of software lifecycle coverage of ten studied Agile methodologies, including RUP in 2002. Abrahamsson *et al.* concluded that not all Agile methods were suitable for all phases of the software development lifecycle. DSDM and RUP were the only methods that provided full coverage. This has been reconfirmed in a more recent publication by Dingsoyr *et. al* [26] in 2008. These authors provide a recent roadmap of empirical research in Agile practices. Figure 3.5 illustrates their assessment of the current status for agile software development and suggested goal for 2015. In order to achieve the projected goal, they suggest that empirical

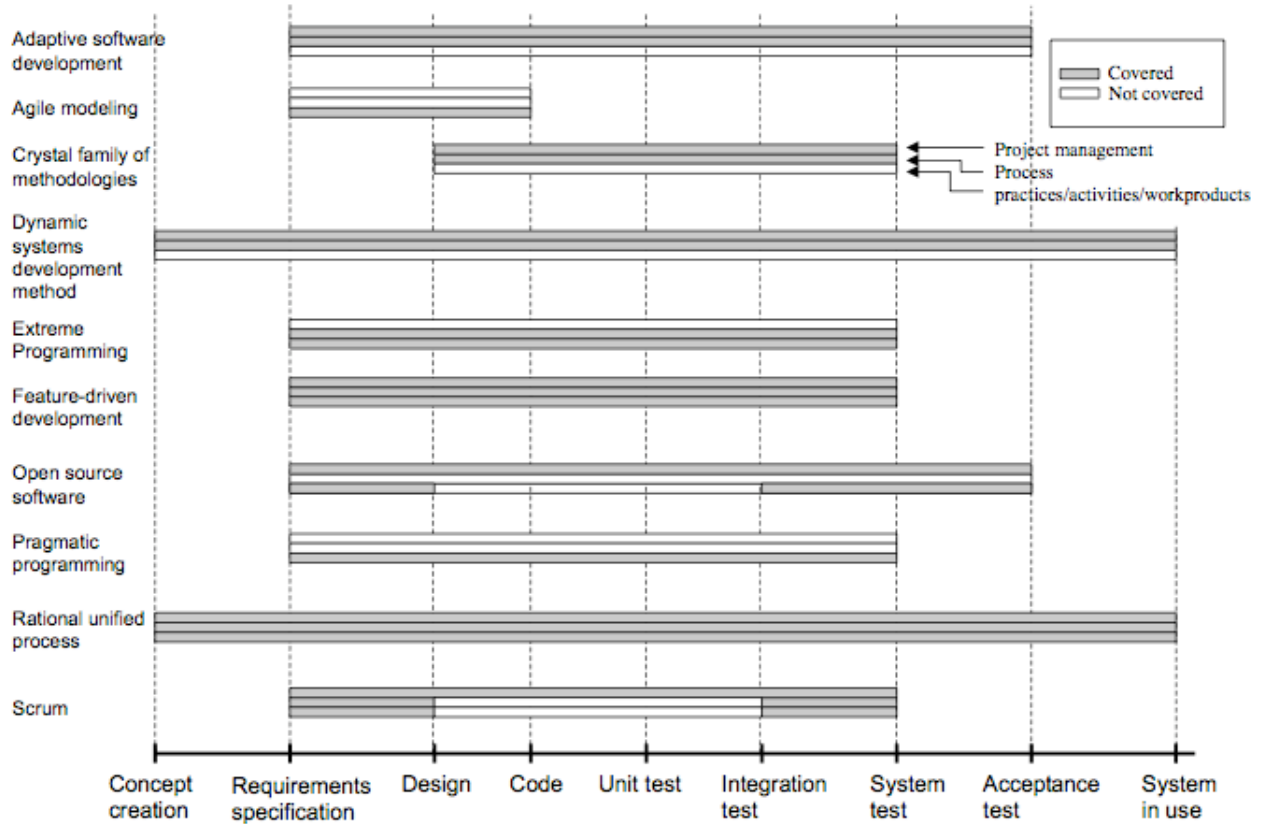


Figure 3.4: Abrahamsson *et al.* Software Development lifecycle support

researchers should focus on *experienced agile teams and organizations, connecting better to existing streams of research in more established fields, giving more attention to management-oriented approaches, and finally give more emphasis to core ideas in agile software development in order to increase our understanding.* More specifically, these authors call for more industrial collaboration through Action Research and more knowledge on how Agile principles, such as IID, work in different contexts.

The size of teams also used to play a role in deciding which Agile methodology to implement. XP and Scrum focused on small teams of less than ten software engineers[26, 14, 53, 7], while Crystal, DSDM and RUP claim to support larger teams. That being said there are current research trends about scaling Agile to larger teams. These will be discussed in the next section.

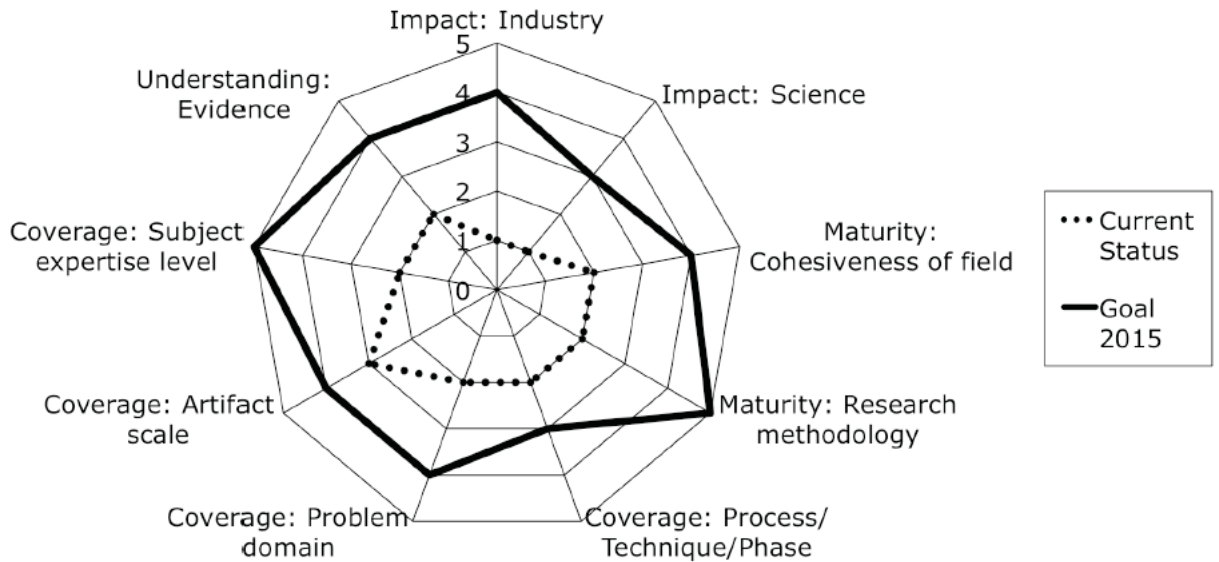


Figure 3.5: Dingsoyr *et al.* [26] empirical agile studies assessment and suggest goal for 2015.

The two mostly cited Agile methods, XP and Scrum, have phases and roles. The XP phases include: exploration, planning, iterations to release, productionizing, maintenance, and death [14]. The most desirable state for an XP project is maintenance as it is the most profitable state for software development. Scrum has three phases: pre-game, development, and post-game. The main differences between these phases and the phases in the RUP lie on their elapsed time, level of formality, and expected milestones.

While some Agile methods detail specific technical practices, “practice-oriented” like eXtreme Programming (XP) and Agile Modeling (AM), others give more abstract definitions of how to manage an Agile team, such as Scrum. Figure 3.6 shows the support relationship of the practices advocated by Beck for XP [14].

### 3.5 Adoption strategies

In 2008 Mike Cohn published an article detailing his experiences with patterns of Agile adoption [23]. There are three set of opposing pairs:

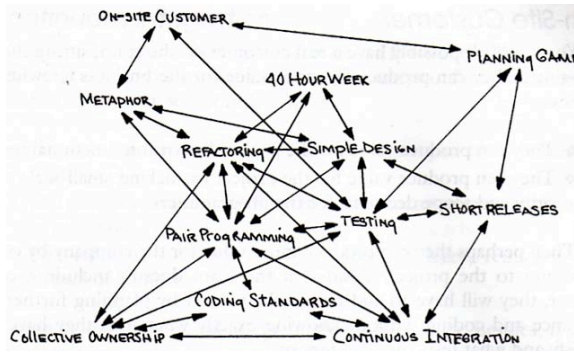


Figure 3.6: Ken Beck representation of the inter dependencies between XP practices [14]

- Start small or go all in

This is also known as staged versus big-bang adoption approaches. In “start small”, a company selects one to three teams to experiment with Agile Methods. The advantages include: reduced costs of mistakes, selecting projects that are in “your favour”, and creating the seeds for future in-house Agile mentors. Disadvantages include: the success of a small project may not appear so compelling, “start small” adoptions take longer to reach the whole organization, and this may mean a lack of commitment from the company to actually move towards a new development approach.

In the “go all in” pattern, the whole organization migrates to the new methodology. The advantages include: the company is committed to the change, the transition is over quickly, there is no problem with differences in approaches from different teams, and resistance can be reduced since it is not just an experiment. Disadvantages include: the cost of mistakes are much higher and are much riskier, it requires the reorganization of management roles, and can create a lot of stress.

- Technical practices first or iterative first

The “technical practices first” pattern resonates with the emergence of XP, a practice-centric Agile method. By adopting Agile practices, Agile will emerge [14].

The main advantage of this strategy is that if all practices are adopted rapid improvements are possible and the transition can be quick. Disadvantages include: if only a portion of practices are introduced, their result may not be as great as expected, there is a learning curve involved, and the team may become too technically focused and forget about the customer collaboration aspects of Agile.

The iterative first pattern indicates that a team is simply working in an iterative way. The main advantage of this pattern is the most people do not argue against working iteratively. The main disadvantage is that many companies may stop there and not adopt other dimensions of Agile practices.

- Stealth mode or Public Display

The “stealth mode” pattern is used when a team adopts Agile practices but only the project team knows. It is a “do it and ask for forgiveness” approach. Advantages include: no one can tell you to stop, and if the project fails no one can blame the difference in methodology (that is if no one notices the difference in approaches). Disadvantages include: lack of organizational support (which is key) and the late news of success due to change are not as compelling. In “public display”, teams adopt Agile and publicly spread this fact to the whole organization. Advantages include: Hawthorne effect (improvements due to the fact that people are watching over the changes), and demonstration of a high level of commitment to the change. The main disadvantage is that some teams go public without being ready to do so bring in skeptics that may hinder their success.

The IID adoption strategy for the government agency under study followed a “go all in, but iterate first with some public display.” Although the mandate was for all projects in the organization to adopt RUP practices, there was close to no adoption strategy for existing projects. Chapter 4 provides further details.

Amr Elssamadisy's work provides a road map for Agile adoption, linking Agile practices with the organizational or technical "smells" it can address [29].

Bottom-up adoptions happen when practices are introduced by the development team, while a top-down adoption is supported and influenced by top-level managers [48]. Krishnamurthy's Agile adoption strategies are based on over fifteen Agile adoption projects in Valtech. This author suggests that top-down approaches work better because they provide: higher visibility, confidence, transparency, and consistency. Bottom-up approaches lack buy-in from other teams and top managers, take time, and require that developers have good salesman's skills. Developers have passion for the practices, subscribe to blogs, and mailing lists but *"I am not saying that the developers do a lousy job in implementation [of Agile] - it is simply that they lack active support from the sponsors"* [48]. This support relates to the monies required to invest in new teams and the powers of influence to change chains of command and control needed to build trust in the team. Krishnamurthy concludes that *"I firmly believe that any organization will succeed in Agile adoption with the right mixture of both top-down and bottom-up strategies."*

Jacobson *et al.* states that *it is clearly a responsibility of the executive at the top of the software engineering business [to transition to RUP]"* [43]. They believe that top level managers needs to create a "case-for-action" statement explaining the need for change, be it time to market or cost reduction amongst others. This statement can include things like: the current business situation, customer expectations for the future, competition, challenges and their side affects, the risks of not changing, and what needs to be done to address these concerns. A possible systematic way for RUP transition could contain:

- The champion. The champion is someone that understands RUP, see its benefits and can sell it to others. This role is most likely filled by a project manager or senior architect. This person must be a good mentor.



- The manager of the first project. This must be an exceptionally capable manager that is in need for the changes that RUP can introduce into the team.
- The mentor. The mentor is someone with experience using RUP and will serve as a supporting stone for the champion and the first manager.
- Where to start. It must be a real project, mission-critical but not overly time-critical.

Furthermore, Jacobson *et al.* reiterate that RUP is a process framework that requires tailoring for a project’s specific needs. They advise companies to *“not doing too many new things at the same time. Process, yes. The tools that go with the process, yes [...] don’t pile too many new things [such as new technologies] at the same time.”*

## 3.6 Related Experiences

In this section we present related experiences and lessons-learned from organizations migrating to iterative and incremental methodologies.

### 3.6.1 Experiencing transitioning to IID

Lewis and Neher [54] give an experience report of the challenges encountered by a pilot development team while migrating to Scrum at Microsoft IT. To diminish the impact on the overall IT management, they provided a map between the existing software development life cycle to the Scrum practices, added iterative periodic reviews, and excluded the complex User Acceptance Testing (UAT) process. The pilot team considered the adoption beneficial, and attributes the success to staying on course with the changes, even when the team resisted them. Lewis and Neher suggest that one of the key success dynamics is to compose the team of “Agile minded” individuals. They forecast that there

will be issues with complex dependencies and with the “co-habitation” of teams in the future.

Megan [85] relates the challenges faced by a QA team used to complex UI testing cycles while moving to Scrum. This author advises to get help from Agile experts early in the process, to be ready for roles and responsibilities changes, and to establish shared ownership of software quality. Beavers [13] reports on his experience as a manager of a team migrating from a waterfall process to Scrum. He advises managers to trust the methodology and intuition, to empower the engineering team, and to become more engaged with the team than only in a process control role. When adopting Scrum, Seffernick [83] suggests establishing an Agile project management framework, engaging product owners from the start, and adding developer best practices. Fry and Greene [35] describe the challenges and opportunities related to a “big-bang” rollout of Agile initiated by the company founder of an R&D company. Their suggestions included to get professional coaching early in the process, to engage as many individuals as possible, and to give the key executives concrete deliverables around the rollout. Jochems and Rodgers [44] describe their experiences with the adoption of an Agile development process mandated by management. They offer six lessons learned: gradually introduce new concepts if the team is resistant to change, get more buy-in from the entire project team, provide the right training for Agile testing methods and tools, allow the project team to evolve and own the process, select a consulting team that meets the needs of the project team, keep the lines of communication open, and quickly address issues.

All the above experience reports had the support, if not the mandate, of the top-level management to adopt an Agile methodology. They were also either in the context of software companies or have been decontextualized. Decontextualization happens when relationships with the natural/original environment where the process changes were introduced have been dismissed from conclusions [51]. Some examples of such contextual

information include: size of system, criticality, age of system, rate of change, business model, stable architecture, team distribution, and governance.

The work presented in this thesis contributes to the IT community by providing complementary empirical evidence of the migration to the RUP iterative methodology in a bureaucratic non-IT environment. It is an addition to the body of knowledge about the effectiveness of IID in environments with higher degrees of formality. It also describes the experience of this non-IT organization in making those changes, including the later adoption of other Agile-related practices, which we hope will be of practical benefit for others undergoing similar process improvement efforts.

Blotner describes a hybrid Agile process introduced at Sabrix [18]. This hybrid process treated each iteration as a mini waterfall project similar to the process adopted by our company during the initial transition stages. Blotner provides the reasons for the adoption of a hybrid process and why each Agile technique such as iteration, unit tests, and business involvement were introduced. Although Blotner describes the influences of the new process adoption on the bug fixing process, he does not provide empirical evidence of software quality improvement. Kobayashi *et al.* conducted two interview case studies with companies adopting XP and found that solo programming introduced more bugs and misunderstandings than pair programming [46]. Rational followed the Booch iterative-development methodology for the third major iteration of the Rose product and found that high code quality was achievable, thus supporting the 80-20 defect categorization and bug criticality (80% of all bugs are found in 20% of the code). Walsh focused largely on bug distribution and established that more abstract subsystems had higher rates of defects, so showing that subsystem hierarchy plays an important part on the bug distribution [87]. Berger discusses her longitudinal study findings based on field observations and interviews dealing with the issues of trust, the lack of collaboration, and the censure between business managers when an Agile process is introduced into a

bureaucratic government agency [16].

The work presented in this thesis provides an example of how companies can use available project metrics to measure how a new practice, in our case iterative and incremental development, benefited the company in the long run and how a team can leverage this success to support further process improvements.

### 3.6.2 RUP and Agile methods comparisons

IID became widely known under the umbrella of Agile methods. But in this thesis we relate the experience of a government agency that chose RUP, a process framework that is viewed as “non-Agile” by many in the Agile community [51]. It is not the focus of this work to confirm or deny the agility of the RUP framework. This section provides some existing literature about some comparisons between Agile and RUP. It provides some grounds for why IID practices may be introduced with processes with higher degrees of formality than the most well known Agile methods.

*“The Japanese samurai Musashi wrote: ‘One can win with the long sword, and one can win with the short sword. Whatever the weapon, there is a time and situation in which it is appropriate. Similarly, we have the long RUP and the short RUP, and all sizes in between. RUP is not a rigid, static recipe, and it evolves with the field and the practitioners, as demonstrated in this new book [Agility and Discipline Made Easy [49]] full of wisdom to illustrate further the liveliness of a process adopted by so many organizations around the world.”* Philippe Kruchten, Professor, University of British Columbia (foreword of [49]).

Zuser *et al.* conducted a comparative study of quality development and assurance between RUP, XP and Microsoft Solution Framework (MSF) [90]. They conducted an analytical bottom-up approach to select principles and techniques for the three chosen methodologies. The criteria to add a principle or a technology was that it needed to

be in practical use in industry and presented in at least one of the three methodologies. Figure 3.7 shows the results of this comparison. It is interesting to point out that Zuser *et al.* did not classify RUP as risk-driven. This is in contrast with the definitions of RUP given by Rational and Jacobson *et al.* The justification behind that classification is that they felt that RUP risk mitigation is optional in comparison to the well-defined MSF risk mitigation processes. One of their goals was to construct a de-facto standard for quality development and assurance. They define a de-facto standard as “*is a standard that is so dominant that everybody seems to follow it like an authorized standard.*” This de-facto standard include the following practices: iterative software development, continuously verification of quality, customer requirements, architecture-driven, and focus on teams.

Criterion	MSF	RUP	XP
Iterative software development	✓	✓	✓
Quality as an objective	✓		
Continuously verification of quality	✓	✓	✓
Customer requirements	✓	✓	✓
Architecture driven	✓	✓	✓
Focus on teams	✓	✓	✓
Pair programming			✓
Tailoring with restrictions	✓	✓	
Configuratrion&Change Mgmt.		✓	
Risk management	✓		

Figure 3.7: Quality Support Practices for RUP, XP and MSF by Zuser *et al.*

To illustrate the commonalities between a light-weight customization of RUP and XP practices, Robert C. Martin describes a fictitious process called “dX” [57]. The “dX” method, which streamlines the RUP process to a minimal set of workproducts [57] mimicking the principles of XP as tailored RUP activities. “*A part from meaning a small change, dX is XP upside down*” [7].

Scott Ambler states that “*RUP done right is Agile*” [9]. Use-cases can be replaced by user stories. Ambler states that RUP has scaling advantages over other Agile methods through: Risk-driven milestones, explicit “go/no-go” decision points, stakeholder concurrence gained during the Inception phase, architecture proven via working software during the Elaboration phase, and managed deployment during the Transition phase.

Michael Hirsch relates experiences applying a light-weight RUP approach in fifteen small projects from Zühlke Engineering [42]. Typical project sizes ranged from one to ten person-years, and three to ten software engineers. Their application of RUP included:

- 10 to 15 core workproducts. Their list of “must have” workproducts included: a software development plan, an iteration plan and iteration assessment for each iteration, a software architecture document, a vision document with a list of required features, a change request list, and a defect list.
- Activities were used as text book definitions, consulted during iteration planning when needed,
- Roles (workers) were used only as skill-checks,
- Weekly meetings,
- Hitting schedule over achieving task completeness (timeboxing).

Hirsch concluded that RUP could be customized to be fairly Agile, that iterative and incremental development is key for success, that no amount of planning can substitute customer collaboration, and that even small projects could benefit from at least 50% of a customer’s time for feedback. Rational and IBM also provide a wide range of white papers about consultants’ experiences implementing RUP, including RUP in “an Agile way” [6].

In the next chapter we present our first case study. This case study presents the adoption of IID practices by a set of existing projects. These improvement efforts were motivated by several concerns identified by the business clients of those projects: poor software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays. This first case study explores how the changes affected three complex existing projects. The effects of these changes are considered quantitatively and qualitatively.

## Chapter 4

### Case Study 1 - Adoption path for process changes in a suite of existing applications

As we mentioned in Chapter 2, the rapid growth of development activities in a suite of existing projects presented some challenges to the existing waterfall methodology. Many releases were rushed or delayed due to fixed delivery dates that forced unreliable acceptance testing schedules on the business clients. These projects quickly started to face problems with software quality and stability leading to dissatisfied business clients that had lost confidence in the software being delivered.

To address these problems IT management, motivated by the IBM Rational Unified Process (RUP), made a small number of process changes over a period of approximately 13 months to a suite of existing projects (Project X, Y and Z). This chapter will discuss these changes to investigate how they impacted two of their biggest hurt areas: software quality and stability (research goal 1) and testing timelines (research goal 2).

Software quality is a term hard to define. Quality means different things for different stakeholders involved in software projects [61]. According to the Software Engineering Book of Knowledge [5], quality has been defined differently by many authors throughout the years, including: conformance to user requirements, achieving excellent levels of fitness for use, market-driven quality, customer-driven quality, and more recently the degree to which a set of inherent characteristics fulfills requirements.

We focus the term software quality in this study to the robustness of the software product releases being delivered to business clients. We focus the term software stability to the consistency of robustness being delivered in each software release. We measure this



robustness and consistency in two ways: qualitatively by interviews with stakeholders, and quantitatively by a study of bug-density. We discuss the use of bug-density data in the following subsections.

In order to address the remaining research goals - how IID affected poor bug-fixing responsiveness (research goal 3) and delivery delays (research goal 4) - a comparative analysis among projects that migrated to IID and a new project that used IID practices since inception is presented on case study 2, Chapter 5.

#### 4.1 The why of change - project specific context

The mission for the three projects under study was to support business critical functions including the digital submission of requests, the internal processing of such information, and the publishing of the results to the public.

The “as-is” business process was to receive requests from external partners by mail in large envelopes containing numerous pages of paper. These envelopes had to be disseminated in a timely fashion to the appropriate departments for processing. Internal “request processors” would then organize all these paper documents into categories and start the manual business rules validation. Paperwork was stored in filing cabinets throughout the organization and business rules were formalized in long documents. The biggest concern with this process was time. Due to the complex nature of these validations, the request processors had to contact the request submitters to clarify and ask for additional information. This additional information followed the same mailing process. Requests used to take anywhere between a couple of weeks to months to be completed. And once the request was completed, the results needed to be manually faxed to all people involved. This information was available to the general public only by mail order or by coming in person to this government agency.

The “to-be” business process would be supported by three distinct applications. The first project (Project X), a native client-server application, was originally created to provide a central data management point for these requests once they had been received in paper format. But after just a few months from its first production release, business clients saw the benefits brought by this centralized data management system and a new vision was born. This vision was to automate and re-engineer the most used business rules validations and processes.

To address the dissemination of the request resolution documentation, a publicly accessible web application, Project Y, was created. This system provided a searchable interface to view the status of requests and related scanned documentation. This enabled requests’ submitters, and other individuals interested in oil & gas related information, to access information in a timely, paperless manner.

The long term vision was to have a completely “paperless” request process. To this end, a new system was built to accept and validate requests through a web application (Project Z). The high number of regulatory business rules paired with their complexity and interdependencies led to a very complex system implementation (over a half million lines of code). This new web application would also receive all related documentation as uploaded attachments to a request. Due to the complexity and innovation involved in this system, Project Z was broken into three phases, each delivering an automated solution to a set of stakeholders. The system delivered by Project X was revisited and changed to automate many other aspects of the business flow involved in completing such requests. This included automating the creation of request resolution documents. Project X’s system evolved into a business workflow application serving internal business users, approximately 800 people. The systems from Project Y and Z had over 11,000 external users. Figure 4.1 provides a high-level illustration of the data flows among these three systems.

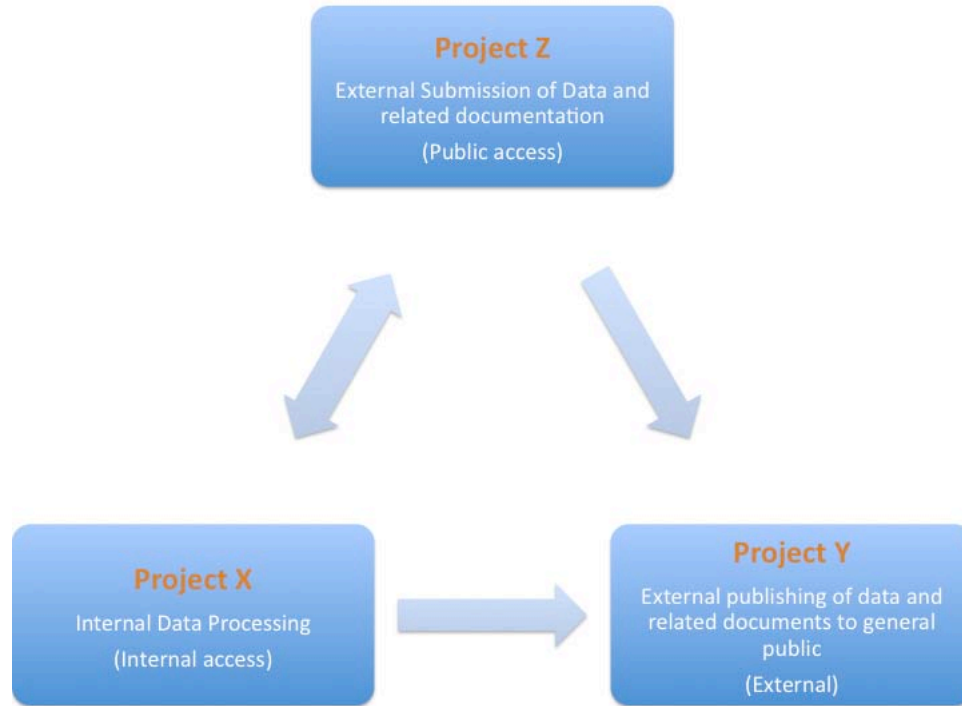


Figure 4.1: Case Study 1 - Projects' Data Flow

In the next sections we will discuss the data gathering approach, the process changes and findings from the data analysis.

## 4.2 Data gathering - Available Project Metrics

It is important to note that this research was conducted after IID practices were introduced by the adoption of the RUP framework. The research goal was to look backwards at the IID adoption to assess how effectively it has been able to deal with the concerns raised by the business clients. More specifically, this case study aims at identifying whether the IID adoption improved the poor and unstable quality of the releases being delivered from the viewpoint of IT Management and Business clients (see Chapter 1, Table 1.1). In order to answer the associated research questions quantitative and qualitative information was gathered about the quality and stability of releases of the systems from Projects X, Y, and Z.

#### 4.2.1 Quantitative Measurements

##### **Bug Reports**

Although there are a number of metrics that can be considered as measures of software quality [61], the most convenient source of quantitative data for the projects under study comes from bug report data. Bug report data was the only metric available from before and after the process changes were made that could be quantitatively analyzed. In this thesis, the terms “bug” and “defect” are used interchangeably. Both refer collectively to faults and failures [61] as the teams made no distinction when bugs were being logged. Enhancement requests were logged in a separate part of the logging system and are not included in our calculations. Business clients defined bugs as “something that should not be happening in the system but it is, or that should be happening but it is not.” More formally, the IEEE Standard Glossary of Software Engineering Terminology defines a fault as a potential “flaw in a software system that causes a failure, and errors as a passive fault execution leading to incorrect system behavior or state” [11].

Specifically, the quantitative data used to answer the research questions below is bug-density data for each release of the systems from Projects X, Y and Z:

- How was the quality and consistency of quality of the releases before IID was introduced?
- How was the quality and consistency of quality of the releases after IID was introduced?

A release happens at the end of an iteration cycle after two weeks of code-freeze in an integrated production-like environment. Production releases happen after hours on the last weekend of the month to minimize disruptions to the business clients (approximately 800 internal and 11,000 external).

Bug reports were extracted from IBM Rational ClearQuest that also contained reports migrated from older logging systems. Code activity data was extracted from two different source control systems: Microsoft Visual Source Safe and IBM Rational ClearCase. Data scripts were created to extract activity data from the code repositories. Both the bug and repository data were parsed and imported into a relational database to facilitate analysis.

IID practices were adopted by the implementation of the RUP framework in this government agency. To reflect this, we grouped the data into three time periods based on release dates: pre-RUP adoption, transition, and partial RUP adoption. This was done to capture the timing of the process changes being introduced. The transition period was included to capture the process changes in motion, as suggested by Pettigrew [67]. The goal was to measure the differences in the releases' bug-density per time period.

The bug-density for a particular release was calculated based on the number of bugs divided by the thousand lines of code changed, added or deleted in the source control repository for the development period of the release (bug/KLOC activity). The release bugs included bugs found during development, testing (in various staging environments) and in production (after the release was deployed). Thus, in this thesis, bug-density equals to the number of bugs reported (during development, testing and after the code was deployed to production) divided by the code activity for the development period. The development period was defined as the first day after a production release up to the day of code freeze for the release. This is in accordance with the definition of bug-density provided by Mohagheghi *et al.* [62]. The normalization by coding activity, instead of total lines of code, provides a better representation of varied levels of effort per release.

We would like to clarify the chosen approach for calculating bug-density data. Perhaps a better method for calculating bug-density would have been to determine the density of bugs based on the release in which they were introduced into the system (which could

potentially be much earlier than the release in which it was found). Unfortunately this was not possible in our case. Before the adoption and customization of the RUP tools, there were no links between coding activities (check-ins) and bug reports. Some check-ins had comments that specified the bug report number, but this was not done consistently by the developers. Thus, there was no reliable way to link the coding activity to a specific bug fix in order to analyze when the bug was introduced.

But in the case of the projects under study, a release delivered code to production and into the hands of real users. The systems under study have over 11,000 users, and although possible, it is unlikely that a bug would remain unfound from release to release. This user base also decreases the likelihood that by increasing the testing effort in a given release, the bug-density for that release is also increased without changing the actual quality of the code produced. By increasing testing efforts, the bug environment categorization changes as we have less production bugs and more bugs found during testing. This shift in categorization does not change the overall bug-density, since all bugs reported (including ones found during testing) were included in the calculations.

Another method of calculating bug-density would have been to only include bugs that slipped through to production. But since our goal was to measure a change in the perception of quality and stability from the business clients perspective, the following contextual facts lead us to include all bugs in our calculations:

- Before the adoption of IID the business clients performed all the system testing without the help of a testing team or automated testing. Thus bugs found before production also influenced how the business clients felt about the quality of the release.
- After the adoption of IID the business clients needed to prioritize new system development against bugs found regardless of which staging environments they

were found in. Thus, the overall number of bugs still influenced how the business clients felt about the quality of the release.

#### 4.2.2 Quantitative Measurements - Data Limitations

The data used to calculate bug-density has several important limitations. Reliable and complete bug data was unavailable prior to July 2004. During this period, only critical bugs that could not be resolved immediately were logged in excel spreadsheets. These spreadsheets were no longer available. Only a few reports were migrated into the new bug logging system and the data sample used for the pre-RUP period starts with release six (R6 see Figure 4.2) of the systems dated from July 2004. It is also important to note that, although data was available from July 2004 on, no formal logging process was in place and developers indicated that it is very likely that bugs were still not being consistently logged. Another limitation includes the loss of code activity for five months of the first Transitional RUP release. The development team started using the IBM Rational ClearCase tool and that presented numerous challenges and overhead to the team. A decision was made to migrate the code back to Visual Source Safe. The first five months of development were comprised mostly of proof of concept code. To minimize the effect of this limitation, the numbers of lines of code in the new files were used as the activity data for that period. The most likely effect of these limitations is that our numbers potentially underestimate the bug-density of the releases made before the process changes we are describing in this thesis.

#### 4.2.3 Qualitative Measurements

Qualitative data was gathered using interviews and field notes. Interviews were designed in part to compensate for limitations in our quantitative data [62]. The interviews also aimed to gather information about various stakeholders' views on the process changes.

Specifically, they were the main source of data to address research goals 2 and 4 (see Chapter 1, Table 1.1):

- Analyze if the IID adoption improved the rushed or insufficient testing timelines from the viewpoint of Business clients (research goal 2);
- Analyze if the IID adoption decreased the delays in release delivery from the viewpoint of IT Management and Business clients (research goal 4).

The interview questions were designed based on qualitative interviewing techniques [78]. The interview questions were structured to ask probing questions and concrete examples [2, 4, 3]. For example, when asking questions about sentiments or specific processes (how do you feel about, how was process X conducted ...), a follow-up question was conducted asking interviewees to describe a time when they felt that way or to provide a specific example of when they were part of the execution of a process.

We interviewed a diverse sample: three business clients (the business area leaders), an architect, a project manager, a business analyst and a technical lead. To allow time for improving interview questions, the interviews were scheduled a few days apart from each other. This was inspired by the reflective coding techniques [75]. This schedule also allowed time for reflection on how to improve our interviewing techniques. Seven audio interviews were recorded and ranged from 35 to 50 minutes each. We also collected field notes from informal question&answer sessions with two developers and the lead tester, as they felt more comfortable with that approach instead of a formal audio recorded interview.

Later, the interviews were transcribed and the data analyzed. Again, we followed qualitative interviewing [78], open coding, memoing, and selective coding techniques. Using Microsoft Word, we highlighted code words and phrases from each transcript. We cross-checked the transcripts to validate they were reoccurring. Ideas and points of in-



terest were memoed. Since the research goal was to investigate some specific research questions and not to derive a theory from the data, we grouped them into three overarching categories (based on commonalities or extreme differences) that related to the research questions we aimed at analyzing: contextual information (the why of change), views on process improvements (the what of change), and challenges encountered during these process improvements (challenges implementing the process of change). Using selective coding, we found subcategories within the context of change (either organizational or project specific) and within the content of change category (discussed in the section 4.4.3).

We also compiled a list of verbatim quotes that consisted of interesting choices of wording and strong sentiments, some of which we use in the following sections.

The following section will illustrate the process of changes introduced based on our data analysis.

### 4.3 The how of change - process changes introduced

Figure 4.2, on page 53, illustrates the bug-density distribution and process changes introduced for all 23 releases, grouped by stage. The prefixes on the x-axis of the graphs indicate R{number} for a standard waterfall release and IR{number} for releases of the code base undergoing the process changes.

#### 4.3.1 Process change 1: Adopting Rational Tools and New Development/Architectural Approach (release IR1)

At this stage, the adoption path for existing projects was limited to the adoption of the IBM Rational Tool set to make project deliverables more visible to all stakeholders. The Rational tools included software for source code repository management, for requirement gathering, and for bug and task logging. Release dates were still booked according

to business needs, as they had been before, and followed a fixed time and fixed scope approach. As well as the organizational adoption of the IBM Rational tools, an Enterprise initiative required that the high profile systems move to .NET, to follow a new Object Oriented Analysis and Design (OOAD) approach. This approach was supported by an in-house development framework primarily responsible for consolidating the serialization and remoting of business objects and database connectivity. This new enterprise approach was also one of the drives to move to iterative development since it would require many systems to be completely re-written. The motivation behind this enterprise initiative was to assist teams in building better and more reliable and maintainable systems, and to consolidate the numerous different architectures that used to cause implementation and configuration issues during deployments.

#### 4.3.2 Process change 2: Iteration Length Based on Tasks (releases: IR2 through IR6)

After release IR1, the team moved into what one of the developers termed a “fire-fighting mode” due to the high number of bugs and software instability. This meant long hours of overtime because of continuous system issues that led to a high volume of support calls. The number of high-priority/high-criticality tasks that needed to be delivered at this stage, and how quickly business clients needed them, defined the length of the iteration; but, the iteration could not exceed 12 weeks until production deployment.

#### 4.3.3 Process change 3: Formal Manual Testing (release IR3)

In the context of this thesis, “formal manual testing” means testing manually executed by a team of trained testers hired to perform this role. Such a team was not available during the Pre-RUP stages when the business clients performed all the acceptance testing. As suggested by the RUP key principles to business driven development [12], management finally agreed to hire a formal manual testing team during IR3.

#### 4.3.4 Process change 4: Code Refactoring (release IR7)

Prior to the implementation of RUP, management considered re-factoring and the development of any kind of automated tests peripheral activities. Some developers felt that management saw these techniques as “developers’ fads”, but the team still believed that management support for refactoring came as a result of the improved communication channels supported by the iterations. The refactoring was started to deal with design flaws in the system especially in the security transactions of the in-house development framework. The developers saw this effort as an outlier to the standard releases as business functionality was not being directly delivered.

#### 4.3.5 Process change 5: Fixed 6 Week Iterations (release IR8)

Management reviewed the adoption strategy for existing projects and moved all project teams into time-boxed 6-week iterations with tasks prioritized to fit the timeframe. Many Agile practitioners may consider 6 weeks too long for a feedback cycle to be deemed iterative, however, for this organization that supports concurrent development projects with dependencies (especially old legacy systems such as mainframes), 6 weeks seemed to be more appropriate to handle the cross team dependencies and risks. This is particularly true when, at the end of each iteration, the code is indeed put into production and is available to internal and external business clients. It is important to note that according to our interviewees, during the Waterfall stages project releases took anywhere between one to three years to be moved into production with many smaller “fixer-up” releases to handle the missed or wrong requirements.

## 4.4 The content of change

As mentioned in the previous sections, we calculated the bug-density for all available releases and grouped the releases into three adoption stages: Pre-RUP (waterfall), Transition, and Partial RUP. These calculations are show in Figure 4.2.

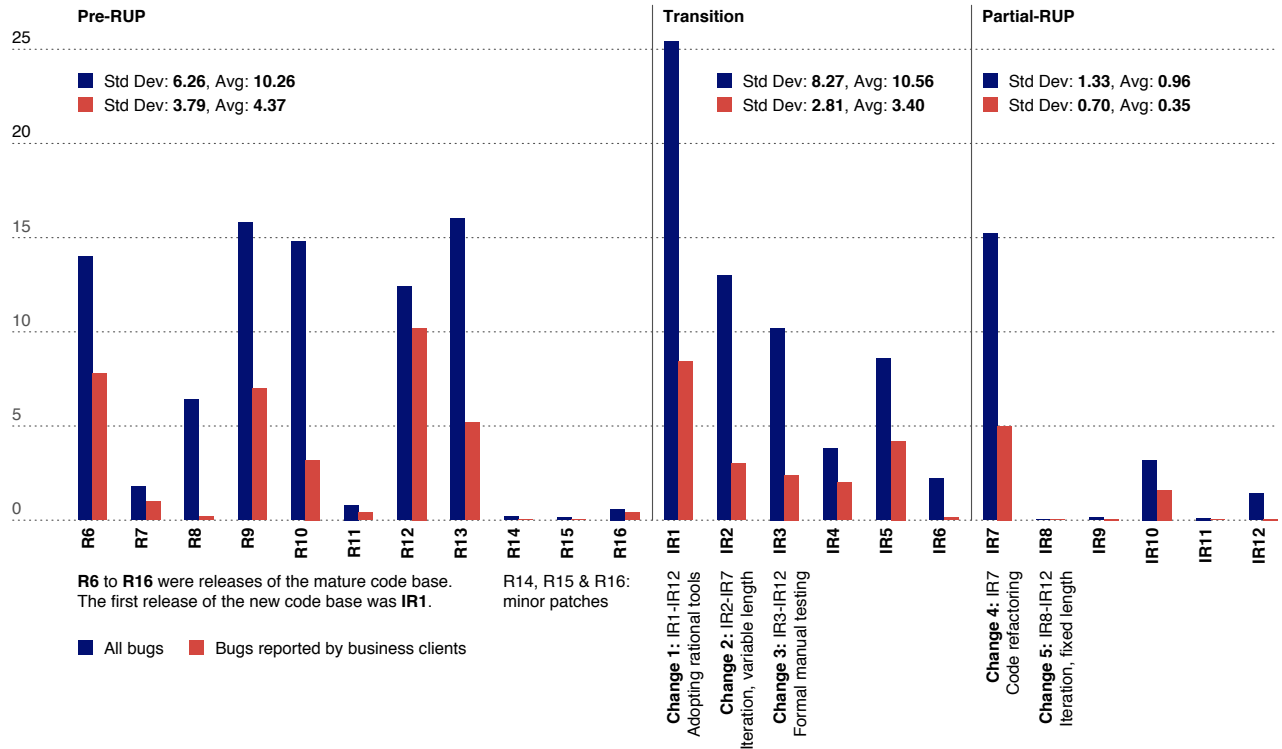


Figure 4.2: Bug-density Distribution

### 4.4.1 Discussion on the use of average and standard deviation

A shapiro-Wilk W test at 95% confidence interval showed that the data set was not normally distributed (test Figures available in Appendix A.3), which is in accordance with Pyzdek experiences with industrial quality-related data (non-normal is the norm) [74]. This was also true when normality was tested for each adoption stage separately. As such, the data sample was not suitable for statistical tests such as ANOVA, so no

tests of statistical difference between the stages is provided. Also, the limited number of data points was not suitable for more sophisticated statistical calculations. Instead, the findings are discussed in terms of average and standard deviation per adoption stage, as well as in terms of the qualitative data gathered during interviews.

The arithmetic mean (average) describes the central location of the releases' bug-density in a given adoption stage. It is the sum of the releases' bug-density divided by the number of releases in a given adoption stage. We use the average bug-density as a quantitative indicator of the overall quality delivered at a given adoption stage.

The standard deviation describes the spread of releases' bug-density from the calculated average. It shows the broadness of the values from the average. We use the standard deviation as a quantitative measurement of the variance in the quality of the releases being delivered at a given adoption stage.

Given these metrics, the optimal results for the projects under study would be to have a low bug-density average paired with a low bug-density standard deviation. A lower average paired with a high standard deviation would still indicate that the quality being delivered by releases is not consistent. On the other hand, a high average paired with a low standard deviation would indicate that teams are consistently delivering low quality releases.

#### 4.4.2 IID, aided by code re-factoring and formal testing, improved software quality and software stability

**Pre-RUP.** Once we grouped the data by release, releases R14, R15, and R16 stood out. Using the interviews, we found that these three last releases of the Pre-RUP period only included minor patches to Project X. As they were not representative of a standard release, they were not used in our averages and standard deviation calculations. The average bug-density between the releases in the Pre-RUP period was 10.26 bugs/KLOC

and the standard deviation was 6.26 bugs/KLOC. This is a good indication of the lack of consistency in the quality of the software being delivered, and was later confirmed by the interviews with developers and business clients. The lack of consistency, referred to as instability by the business clients, generated issues of trust and censure between IT and business clients as detailed in the following section. The highest bug-density encountered in this stage was 15.96 bugs/KLOC; this was also the last “standard” release of the legacy code-base, which had been in production for approximately three years.

**RUP Transition.** The standard deviation of 8.27 bugs/KLOC as well as the highest data spike (25.33 bugs/KLOC), indicate that things actually got worse during the transition period. This is consistent with other industrial reports which indicate that any new approach first makes things worse, as there is a learning curve [13, 35, 79, 83]. Based on our interviews and observations, we found that in addition to challenges involved in moving to a new process, the following contextual factors also contributed to these results:

1. Adoption of new technologies and implementation of new complex functionality. As mentioned in Process Change 1, the projects migrated to an OOAD development approach using .NET and an in-house development framework. The systems we measured here were the “guinea pigs” of the new in-house framework. They suffered from the lack of design direction with missed documentation about how to properly use this framework and from the bugs encountered during development. Table 4.4.2 summarizes the major system differences between the pre and post stages. Another substantial functional module was implemented that added yet another set of stakeholders to the picture. The new module’s requirements were complex and poorly documented, in some cases spread over more than four documents. Since the projects under study were existing systems, they suffered from the initial unknowns

of adopting a new methodology as well as the “direct migration syndrome” where code was simply adapted to the .NET framework instead of being revisited and refactored. On many occasions, the team felt lost with limited support from the RUP Process Engineer who was a part-time, shared resource.

2. Increased development team. The development team more than tripled in size, from four to over fifteen developers during the first RUP transitional release. Adding extra resources at first reduces productivity and may lengthen the delivery timelines (Brooks’ Law) [19]. Because of the lack of the business domain knowledge, new developers would fix an issue but also create new bugs. The subsystem drive architecture abstracted many areas of common business rules related code. These particularly more abstract areas of the subsystems were usually the ones which broke during this phase. This is in accordance with Blotners findings [18] that code abstraction plays an important role in code bug concentration. Accordingly, the first release of the new code base (IR1), that was initially estimated as a minor effort, did not follow an iterative approach and had the scheduled production date delayed four times. The total development time was over 47 weeks (in elapsed time). The team morale was at its lowest, developers were burned-out due to long overtime hours, and there were over 680 bugs logged against the release.

As the transition was still at the very beginning, management and business clients attributed this failure to the limited RUP adoption; “honestly, it felt like nothing had changed, other than we had to do more documentation.”

**Partial RUP.** The standard deviation of 1.33 bugs/KLOC paired with the lowest average of 0.96 bugs/KLOC indicates an increase in the overall software quality and software stability. We would like to discuss some interesting points in the data. First, the de-

velopers attributed the high bug-density of the first Partial adoption release (IR7) to a major system re-factoring (Change 4) done without comprehensive automated unit tests coverage. Since developers identified this release as an outlier to the standard releases, we did not include it in our averages and standard deviation analysis. Second, the team attributed the spike seen in IR10 to another re-write of the system to support a new in-house framework version, again without any significant automated testing coverage. But since it also included new business functionality it was included in our average and standard deviation calculations.

<b>Pre-RUP</b>	<b>Transition &amp; Partial RUP</b>
3 years of maturity of VB/ASP code	New code Base written in .NET 1.1
Simple 2-tier design	Distributed n-tier application
Releases booked based on a need basis	Releases booked on a need basis during transition, than moved to a 6 weeks iteration schedule
Established technology	New technology, new in-house development framework, and new document management tool
Code repository - Visual Source Safe	Code repository - ClearCase (caused many merging issues)
Volatile requirements, using outdated word documents	Requirements using Requisite Pro, and excel spreadsheets due to tool limitations
No automated test	No automated test
No formal testing team	Formal testing team

Table 4.1: Process and context differences between process stages.

4.4.3 IID, aided by code re-factoring and formal testing, increased customer satisfaction

In Figure 4.2, we also isolated the bug-density of bugs logged by business clients alone. The quantitative analysis again points out that the transition stage presented challenges. However, after the first transition release, the bug-density logged by business clients dropped. The Partial RUP adoption stage has the lowest average of bug-density



(0.35 bugs/KLOC) compared to the Transition (3.40 bugs/KLOC) and Pre-RUP (4.37 bugs/KLOC) stages. As well, it has the lowest standard deviation (0.70 bugs/KLOC) that indicates an increase in the quality and stability of the software delivered to user acceptance testing.

During the initial Pre-RUP development stages, the communication channels were open between clients and the small development team. After the team increased in size and the waterfall process broke down, management prohibited business clients from contacting developers: *“at one point it was so bad that, if we were seen talking to a developer, we would get a nasty e-mail from management.”* Business clients missed the small team atmosphere and felt that they had “lost the teamwork.”

When iterative development was adopted, the communication channels were slowly re-opened through prioritization meetings, iteration planning sessions, iteration assessments, and iteration user acceptance testing. Any new system developed at the company is now using a more complete RUP process that provides constant opportunities for business clients feedback. They are welcome to “drop by” and view the progress at any time.

As mentioned in section 4.2.1, the interviews with business clients and field observations were designed to address quantitative data limitations and to answer specific research goals. The data analysis provided some insights into designed research questions (insufficient testing timelines, poor bug-fixing responsiveness), and also provided other sentiments about the experience the organization had in making those process changes (reestablishment of business involvement, and improved communication and management of expectations).

1. Better distribution of Acceptance Testing effort

*How were releases tested before RUP was introduced?* The three section lead managers and a few senior end-users conducted acceptance testing. The interviewed

business clients felt that the original development process did not provide reasonable time for testing the system: *“you would get it [the application] for two days, and you need to approve it and its gotta go.”* They felt rushed and uncomfortable by having to sign-off on a system that took over 10 months to develop, and only a few days to test. At the end of the development cycle, business had compounded testing to do, which caused an overwhelming workload: *“[testing] is not my full time job. I need to deal with core business. Testing work is supposed to be on the side, but [at that point] becomes fulltime work. I am basically doing two fulltime jobs, which makes things difficult.”*

*How were releases tested after RUP was introduced?* Iterative development has time-boxed the testing effort required by business to two weeks per iteration. Testing was not compounded, but it could still feel rushed based on the number of changes implemented during the iteration. The business clients saw the organized and scheduled acceptance testing effort as a big improvement: *“it is better to plan that way, even from a personal life perspective. It is just way more organized than it used to be.”* Some interviewees actually stated that this organized schedule was the major improvement provided by the process changes made to the existing projects.

## 2. Introduction of testing team

*How were releases tested before RUP was introduced?* A Quality Assurance (QA) team was not available in the pre-RUP stage as management perceived formal testing as peripheral in comparison with other more pressing deliverables. The code would go from the developers who did not implement any automated tests to the business clients for testing: *“we used to joke around saying what is the point? I open it [the application] and get the yellow screen of death<sup>1</sup>, so you are just wasting*

---

<sup>1</sup>This refers to the fatal application errors in .Net, which display the error message in a yellow screen.

*my time!”*

*How were releases tested after RUP was introduced?* As suggested by the six key RUP principles for business-driven development, management hired a full-time testing team at the end of the third transitional iteration. After the introduction of the testing team, all code went through a round of formal testing before being given to the business users. As a result, the business clients found fewer fatal errors during acceptance testing. They could also focus on the areas that have been changed or included, as the testing team was responsible for the regression testing, which was considered a big time saver: *“it is night and day.”*

However, the testing required by the QA is complex and time consuming. The QA team is shared among all projects, and may not have enough resources to provide the appropriate levels of manual regression testing. That, in addition to the lack of unit tests, has been a sore spot for pre-existing systems. Problems reappeared in production after being fixed (more details provided in Chapter 6). New systems are now implementing unit tests that allow developers to regression test the application even before it is handed off to the formal testing team.

### 3. Less pushback on necessary changes

*How long did it take for bugs and enhancements to get implemented in existing projects that transitioned to RUP?* In the former Waterfall process, IT management would push back to implement changes: *“so you get stuck with it.”* It was very difficult for business clients to define the project’s scope to the degree of granularity needed at the initial requirements gathering stages: *“it is virtually impossible to foresee all the details and functionality of an application to define a hard scope document. To expect that when creating a scope document is unreasonable and shortsighted.”* Business clients would have to make go-no-go decisions close to

the production date, and many releases were delayed as much as a year due to poor testing results, and essential requirements being missed in the original scope document. IID provided business with set release dates scheduled 6 weeks apart from each other that allowed critical items to be negotiated, prioritized and included in the next release.

*How long did it take for bugs to get fixed in new projects that started development using RUP?* For new development projects, the iteration assessments and demos allowed business clients to provide the feedback necessary to avoid major changes later on in the process. A visible result of that was the number of bugs in production for the first system developed using RUP, which is less than a dozen compared to the hundreds found in the former ad-hoc projects.

We provide more details and quantitative data about this in Chapter 5.

#### 4. Reestablishment of business involvement

During the first system release, the small team atmosphere allowed the business clients to have an active role in the requirements gathering stages of the system development. The project manager would set up business meetings with the involved stakeholders to gather requirements. Some requirements were documented in Word or Excel, others were only verbally communicated to the development team. Later the development team would create screen mock-ups of the application, and present them to the business clients for feedback during meetings. Although acceptance testing did not occur until development was completed, business clients found the screen images extremely useful: *“even though we didn’t get to test until the end, when we got the application, it was not about testing the screens and see how they looked like, it was testing to see if they worked, if they met the requirements.”* They were very pleased with the first release of the system, which took approximately

one year to be production ready.

As the number of requirements increased, so did the IT team size. More rigorous management procedures were put into place. Developers needed to follow the project plan more closely, in some cases resulting in frustration, as the plan was quickly outdated. Business clients were used to contacting developers with requests. The developers would in turn implement the requirements, causing a delay to the defined project plan. This was also found by Blotner [18]. As a result, managers prohibited business clients from contacting developers directly: *“we got cut off by management: thats it, no more talking to the developers!”* It got to the point where management would complain about e-mails sent to developers by business: *“don”t be seen talking to a developer, and really, that environment was not good. For us that doesn’t work!”* Business clients felt that they lost the element of teamwork and this caused friction and “blaming games” between them and IT management. This censure was *“very disruptive to everyone involved.”*

The introduction of the six week iterations helped business clients become more involved in the iteration planning by prioritizing which items need to be worked on first, and which ones require more analysis. They felt more ownership and accountability over the decisions made which helped rebuild the teamwork [40]. Business clients are now allowed to contact developers: *“a developer came and sat with me [to discuss a task] and mocked it up in paper, and asked if it was ok with me, which was fantastic.”* Still, involvement with developers has been limited as most of the communication goes through the project leaders and business analysts. Perhaps this can be attributed to the responsibilities defined in the RUP roles. Business clients feel that this “middleman” approach to communication has advantages, when dealing with developers that lack interpersonal skills, but also has the drawback of information being “lost in translation.” To mitigate this issue,

key developers are now invited to business meetings.

Business clients feel that the most visible gains are in the new systems that started development using the iterative RUP process as they were involved in the process since inception. They were not given functional parts of the system to test until the construction stages, but they had iteration assessment meetings where demos were provided, allowing feedback on system functionality. As a result, the first fully iteratively implemented system was the first project in more than six years to be delivered on-time and on-budget: *“which is significant for the organization, the first in years, [laughs] that says a lot. Our executive was very happy, from our perspective [it] is great.”* We provide more details and quantitative data on this project on Chapter 5.

#### 5. Improved communication and management of expectations

In the former process, issues were logged in Excel spreadsheets, discussed in business meetings, prioritized and put away in a place only accessible to managers. The RUP implementation involved the adoption of Rational Tools including bug and enhancement logging software. Business clients have access to these tools and are able to view what is outstanding. This is very important to assist them in negotiations of shared resources and to have more realistic expectations of what and when changes will be delivered. They also feel that, overall, the projects are much more organized, and due to the iterations, they are in constant communication with the team, which helps reduce “surprises” at the end of a release cycle.

## 4.5 Challenges

The results above indicate that the transition to RUP was not smooth. Lack of direction, compounded by new technologies and growing development teams, resulted in a

temporary decrease of software quality. Because of the complex system changes and methodology transition, the team members indicated that there was simply *“too much going on all at once”*. Supported by the new RUP process, two factors assisted the team to get back on track: the introduction of a testing team, and the major re-factoring of the code base. We asked team members what they think would have happened if a direct transition to an agile methodology had been attempted during the transition stage and several of them felt that the project would have been *“back to waterfall in no time.”* However, management supported RUP and the process survived the transition stage. Due to its perceived success, management is now more open to developers requests.

Problems fixed in prior releases would suddenly show up again in production. As the systems had no automated tests, it was very time consuming for the testing team to perform a full regression test. As mentioned in Change 4, management did not support test automation as it was considered to be peripheral and not visible enough to provide immediate business value. We will provide more insights into this challenge in Chapter 6.

The new system design included extra layers of dependencies with subsystems from other teams and the in-house framework. This has caused many deployment coordination issues, thus testing was scheduled and not on-ongoing. A reader might think that automated builds and continuous integration could resolve integration and configuration issues, but during the waterfall days and even the initial RUP transition, there was simply no buy-in from management to invest in these perceived peripheral techniques. Software builds and deployments were manually made to each of the staging environments. Each staging environment has its own database server, file system, and a document management system. Most in-house systems are dependent on mainframe applications and batch processes and changes to server configurations made by one team have the potential to impact other teams. This situation certainly happened very frequently in the projects

under study. On many occasions, successfully deployed applications started to fail and teams were not made aware of what actually changed in the environment. There was no formal ownership of non-hardware environment configurations. As a result, teams had to constantly and manually verify server configurations to ensure that all different environments were set-up correctly for their applications. The entire situation was almost ad-hoc, caused last-minute surprises, and chaos in many occasions. “It worked for me yesterday, but I don't know why it doesn't work today!” This manual check of configurations was also very time consuming.

The projects under study tried and failed to hold daily stand-up meetings. Instead, a weekly round-table team meeting was held and seemed to be efficient. Not all business stakeholders were available though, as this meeting was seen as less critical than other business engagements. This can perhaps be attributed to the adoption path for new projects as they followed iterative development from inception, thus getting business involvement at the onset.

Project leads review the work products required by RUP and only pass Class and other system-related models to the development team. The leads assume the responsibility of producing iteration assessment documents, phase assessment documents, status updates, and other required work products using the data gathered during daily/weekly stand-up meetings. Their goal is to minimize the list of required work products and this has been surprisingly well received by upper management.

## 4.6 Lessons-learned

As mentioned above, during the RUP Transition, the projects studied also introduced new architectural directions, new technologies, and a new set of stakeholders. Jacobson *et al.* [43] suggest to introduce as few parallel factors to the RUP adoption as possible.



But this was not the case observed here. The project teams felt that there was too much going on all at once, which made it hard to distinguish process-related problems from new technological challenges. A lesson-learned to the teams was that they should have minimized the amount of other changes introduced during the first stages of the IID adoption.

The initial RUP adoption was very much tool-centric instead of process-centric. The teams feel that they would have seen quicker improvements if a process-centric approach was followed first. The Rational code repository tool was also not well received by the teams, due to many merge and delivery problems that caused delays to testing schedules.

The business clients felt that could have benefited from formal training in regards to the changes being introduced by the RUP framework. In many occasions they expressed feeling like “puppy dogs being led here and there” without initially knowing the purpose behind iteration meetings, use case meetings, among others.

The projects observed developed a quality issue before IID was introduced. It took over 13 months to see a real stabilization of the quality being delivered to business clients. Another lesson-learned is that management should have realistic expectations in regards to the process improvements introduced. It took time to see real improvements, but IID was a very worthwhile investment to these teams.

## 4.7 Limitations

In this case study we report on one organization’s experience making process changes in several existing projects. Care should be taken in drawing general conclusions based on our results. However, we believe that our findings can be valuable for organizations facing similar challenges, especially when aggregated with other findings [64]. Exploring the consequences of the process changes made using bug-density data is valuable as re-

ducing bugs is a goal for many software development organizations. However, accurately capturing all of the consequences of the changes made in this organization would require a wider range of data (e.g., total development cost) than we are able to report on at this time. This limitation is mitigated by our qualitative data, which fills in some of the gaps.

## 4.8 Research goals addressed

This chapter explored how the IID adoption affected several existing projects. More specifically, this case study discussed how the IID adoption impacted software quality and stability (research goal 1), and testing timelines (research goal 2).

Through the analysis of quantitative bug-density data and qualitative data gathered during interviews, we discovered that IID paired with investments into formal test teams and refactoring allowed existing projects to:

- Improve the software quality and stability of quality-challenged complex systems.
- Provide five areas of improvements to business clients: reestablishment of business involvement, better distribution of acceptance testing effort, introduction of a formal testing team, less pushback on necessary changes, and improved communication and management of expectations.

This chapter also provided some experiences and practical lessons learned from the projects adopting IID practices, such as:

- During the transitional stages software quality and stability actually worsen as experienced by others [35, 44, 83].
- Lack of appropriate regression testing resulted in previously fixed bugs to reappear in production.

- Staging environment configuration management created issues leading to deployment delays and staging environment instability.
- Existing projects held weekly meetings instead of daily stand up meetings.
- Project leads took the responsibility for creating most of the required RUP work products.
- The government agency under study had complex system changes happening at the same time as the methodology transition. The team members indicated that there was simply “too much going on all at once.” This made it difficult to differentiate problems due to the IID introduction or due to other technical or contextual issues.

## Chapter 5

### Case Study 2. Adoption path for process changes in a greenfield project

In the context of this thesis, a greenfield project is one that is being developed from ground-zero, a new project.

In case study 1, it was demonstrated that the adoption of an iterative and incremental development approach aided by the introduction of formal testing teams and re-factoring increased the quality and stability of an existing suite of software systems. It also provided five areas of improvements for business clients, including the improvement of testing timelines. An important point raised by business clients was that, although there was an improvement in the existing systems, the most visible gains from the RUP adoption came from new projects that followed the iterative RUP process since inception. In particular, they expressed strong sentiments of satisfaction and success towards one project from the same IT program from case study 1 (Project A - Figure 2.2).

This chapter will discuss how effectively the process changes have been able to deal with the concerns raised by the business clients in the context of a new project. It will address the remaining primary research goals - how IID affected poor bug-fixing responsiveness (**research goal 3**) and delivery delays (**research goal 4**). In order to properly address these research goals, this case study presents a comparative analysis with the existing projects that migrated to IID, presented on Chapter 4, case study 1.

Due to some data limitations, quantitative metrics related to software quality and stability were not used in this case study. Only qualitative measurements were used instead to answer research goal 1 (how IID affected software quality and stability). We

provided further details in the following sections. On the other hand, the data from this case study allowed us to present other available project metrics to validate the perceived success of this project.

## 5.1 The why of change - project specific context

The mission for Project A was to provide electronic registration and management of parties involved in legal hearings. These hearings resolved conflicts of interest with regulatory requests being processed by this government agency. The “as-is” business process was to receive documentation from involved parties manually by CDs or by e-mails. Many full-time employees were responsible for manually generating an inventory for the documentation. The inventory and all related attachments then needed to be disseminated in a timely manner to all people involved in these legal disputes. Employees would send e-mails or burn CDs (due to the large volume and size of attachments - gigabytes of data), and have them delivered to involved parties on a semi-weekly basis. This information was available to the general public only by mail order or by in-person request.

The “to-be” business process would be supported by four distinct systems:

- One external facing web application that required authentication (for people involved in the legal hearing process only),
- One external facing web application for general public access of the data,
- One internal web service application to support the downloading and uploading of documents programatically through network boundaries,
- One internal application to support the management of information.

The first vision and feasibility study for this project was delivered in 2005. At that point the project was staffed by only one senior business analyst that delivered numerous use cases detailing the desired requirements, still influenced by the waterfall design up-front concept. This project was put on hold until 2006 because of budgeting constraints that resulted from the challenging transition stage of case study 1. Finally in June 2006, a team was put into place for an actual project kick-off.

During the Elaboration phase, the existing use cases were revisited. The large majority had to be re-written and broken down into smaller use cases. Many requirements had changed as the business clients streamlined their business processes using the projects from case study 1 (Projects X, Y, and Z). The use case normalization was needed to support the concept of quick delivery of smaller portions of the system.

The next sections discuss the available project metrics analyzed for this case study.

## 5.2 Data gathering - available project metrics

### 5.2.1 Quantitative measurements of success

Our first quantitatively focused research goal was to analyze if the IID adoption improved the poor and unstable quality of the releases being delivered from the viewpoint of IT Management and business clients (goal 1 - see Table 1.1, Chapter 1). In the projects of case study 1, bug-density for 23 releases of the systems was used as a quantitative metric. Unfortunately, measuring bug-density for Project A presented some challenges. There were no bugs logged in the bug logging system before Phase 1 Construction Iteration C5. This was clarified by one of the developers and the business analyst. During the first iterations, the project was very dynamic. Developers would interact with the business analyst and business clients to clarify issues. If a problem was found, it was dealt with and validated by the business analyst without the need to log it. But once business

clients started to become more involved in the actual hands-on acceptance testing of the system, the team started to use the bug logging tool.

In case study 1, each data point (a release - see figure 4.2) represented code delivered to production into the hands of real users. This played a big factor in the accuracy of where the bug was actually introduced. The systems in case study 1 had over eleven thousand users, and although possible, it is unlikely that a bug would remain unfound from release to release. In the context of Project A, iterations did not lead to a production release. The only iterations that delivered code to production were three of the transitional iterations. As it will be detailed below, the business clients were not involved in the “hands-on testing” until Phase 1 Construction Iteration C5 when bugs started being logged into the tracking tool. Due to the elapsed time leading to production releases, we were not confident that we could link the bugs to the iteration they were logged against. As such, bug-density was not considered a reliable quantitative measure for the data we could retrieve for this project. Instead, we used qualitative measurements, such as interviews, field notes, and informal question&answer sessions with team members to answer these research questions.

Our second quantitatively focused research goal was to analyze if the IID adoption improved the long wait and red-tape involved in addressing bug fixes from the viewpoint of business clients (goal 3 - see Table 1.1, Chapter 1). We summarized this goal as “responsiveness.” In order to measure responsiveness, bug reports were extracted including their change history from IBM Rational ClearQuest. Bug history included the following distinct actions: Assign, Close, Duplicate, Import, Modify, Open, Postpone, PreValidate, Re\_open, Reject, Resolve, Submit, ToAcceptance, ToDevelopment, ToTest, Unduplicate, and Validate.

Bugs were grouped based on priority. Mixing all the bugs together would lead to a less realistic representation of bug fixing responsiveness as higher priority items would

most likely be worked on first. Bug priority was used instead of bug criticality because according to interviewees, a clear definition of a critical bug was not available until the later stages of the RUP adoption. As a result, criticality was often used as a political way to prioritize tasks instead of categorizing its impact to the business process, as also reported by Mohagheghi *et. al* [61]. We define bug-fixing responsiveness as:

Bug-fixing responsiveness (in days) = Date the bug was last moved to a close state (action: Close) - the date it was submitted (action: Submit)

There were a total of 358 bugs logged against the systems in Project A and 1329 bugs logged against the systems in the projects from case study 1 (projects X, Y and Z - from the RUP Transition forward).

Bug reports were also grouped based on a tag that specified in which staging environment they were found. In his keynote at the 31<sup>th</sup> Conference on Software Engineering (ICSE) Steve McConnell [59] stated that one of his top ten most powerful ideas in software engineering still includes the defect cost increase (DCI). The cost associated with fixing bugs increases over time. Although the original statement about the cost magnitude associated with late bug-fixing were elaborated based on waterfall approaches (in fact these same statements support upfront centric processes), McConnell believes that even in iterative and incremental processes the costs associated with fixing bugs increase based on the activity in which they are detected (requirements, architecture, construction, system test, and post-release in ascending order of cost).

### 5.2.2 Quantitative measurements of success - data limitations

The data presented some challenges as the goal was to address research goal 3 (how IID affected poor bug-fixing responsiveness) for existing projects and projects developed after



IID was introduced. There were many differences in the way projects X, Y, Z (case study 1) and project A used their bug tracking system. In case study 1, the actions related to the bug-logging process were redefined at least three times, meaning different things at different points in time. To deal with these differences, we computed the number of days it took to fix an issue by the first and last commonly used actions: Submit and Close. To business clients, bugs became an issue when they were found regardless of when they were introduced in the code. From a management perspective, the Close action is the one that retires a bug from a prioritization list, thus from the business representatives' radar.

The goal was to measure bug fixing responsiveness after the IID process was introduced. This would also allow a more fair comparison between projects in case study 1 that transitioned to RUP and projects from case study 2 that used RUP from inception. Thus only bugs logged in case study 1 from the RUP Transition forward were measured. Bug reports that did not include any action related to a developer analyzing and/or fixing the bug were excluded. From the 1329 bugs logged for case study 1 from the RUP Transition forward, we could only include 958 in our analysis. For the 358 bugs logged for Case Study 2, we could only include 318 bugs in this analysis.

### 5.2.3 Qualitative measurements of success

Data was gathered using field notes based on question&answer sessions with the project manager, technical lead, business analyst, and one of the developers from this project. Three to four sessions were conducted with each of the participants, usually over coffee, for approximately 20 minutes each time. Questions were structured to ask probing questions and concrete examples [2, 4, 3], using the same techniques specified in case study 1 4. Specifically they were the main source of data to address research goals 1, 2 and 4 for this new project under study:

- Analyze if the IID adoption “*improved software quality and stability from the viewpoint of business clients*” (research goal 1)
- Analyze if the IID adoption “*improved the rushed or insufficient testing timelines from the viewpoint of business clients*” (research goal 2);
- Analyze if the IID adoption “*decreased the delays in release delivery from the viewpoint of IT Management and business clients*” (research goal 4).

Based on our experience from the analysis from case study 1, after each question&answer session, any relevant data was grouped into three abstract categories: project contextual information, views on process improvements, and challenges encountered during these process improvements. We compiled a list of verbatim quotes that consisted of interesting choices of wording and strong sentiments, some of which we use in the following sections.

A side goal was to validate the business client’s perception of success towards project A. Being on-time and on-budget are good traditional indicators of success but they are simply not enough to actually deem a project successful. Some emerging definitions of success in project management state that success is “*getting results and feeling good about it*” [89]. Being on-time and on-budget meet *project plan* expectations, but delivering the right product that provides business value meets *people expectations* [89]. In order to truly measure success, we need to address these two dimensions as illustrated in Figure 5.1. The interviewed business clients, who were also business clients of the prior case study did see this project as a success.

Quantitative and qualitative data is provided to address project expectations (delivery times and budgeting) and to address stakeholders’ expectations (project responsiveness).

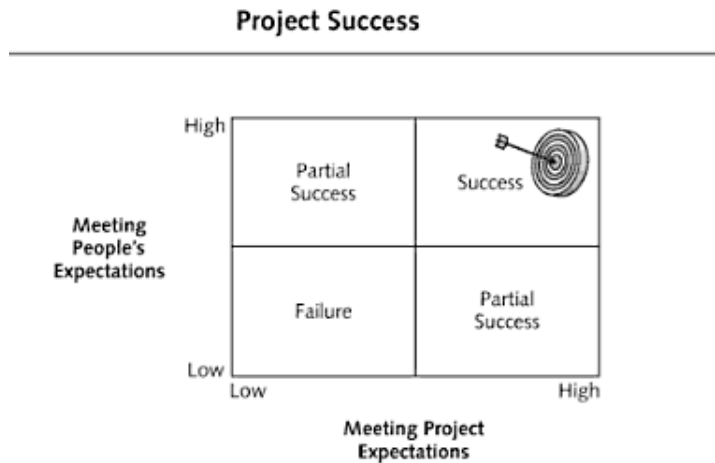


Figure 5.1: Measuring Project Success by Zachary Wong

### 5.3 The how of change - the new RUP execution state

The RUP execution state refers to state of the RUP framework customization. It states the processes, tools, and artifacts being used by the government agency at a given point in time. The newly reviewed RUP execution state of the company included:

- The iterative RUP lifecycle (inception, elaboration, construction, and transition),
- Rational Tools,
- Role sets,
- Selected work products.

#### 5.3.1 RUP Phases

The project followed the four phases of the Iterative RUP process (Table 5.1). During the Inception Phase, a project vision, mission statements, and project kick-off meetings were held. The in-house RUP expert provided some guidance to the team in regards to process-related expectations.

Project Schedule		
Phase 1		
RUP Phase	Iteration	Length (in weeks)
Inception	Iteration I1	7
Elaboration	Iteration E1	3
	Iteration E2	6
	Iteration E3	4
	Iteration E4	3
Construction	Iteration C1	3
	Iteration C2	3
	Iteration C3	3
Transition (Internal Beta 1 Release)	Iteration T1	7
Construction	Iteration C4	3
	Iteration C5	3
	Iteration C6	3
Transition (Internal Beta 2 Release)	Iteration T2	7
Transition (Production Release)	Iteration T3	7
Phase 2		
RUP Phase	Iteration	Length (in weeks)
Inception	Iteration I1	6
Elaboration	Elaboration E1	3
Construction	Iteration C1	3
	Iteration C2	3
Transition (Internal Beta 1 Release)	Iteration T1	7
Construction	Iteration C3	3
	Iteration C4	3
	Iteration C5	3
	Iteration C6	3
Transition (Production Release)	Iteration T2	7

Table 5.1: Case Study 2 - Project Schedule

As shown in Table 5.1, the team experimented with iteration length during the Inception and Elaboration phases. At the end of the first Elaboration phase, the team decided to follow three-week iterations. Transition phase iterations were an exception; they were seven weeks long.

The team also started to hold daily stand-up meetings during the Elaboration phases. These meetings only included the development team.

*“How were releases tested after RUP was introduced?”* During elaboration the developers implemented unit tests to prototype and evaluate the risk of project tasks. Later, in the construction stages, the team stopped developing and maintaining these unit tests. Manual testing occurred during the last week of an iteration. The iterations in the Construction phases delivered code to a staging environment at the last week of the iteration. This staging environment, the “Sandbox”, was used by a dedicated tester and the business analyst to test and validate the builds. The team prepared demos for the business clients at the end of each iteration in the Sandbox environment.

Many heavy-weight implementations of the RUP process led the software engineering community to a perception that having Inception and Elaboration phases is the same as following a “big design up-front” approach to software development. To investigate this further and to support the claim of symbiotic relationships between Agile methods and formality [26], the code activity, by thousands of lines of code added, deleted or changed in source control per iteration, was measured. Figure 5.2 shows the results of that data.

The peak of code activity occurred during the first Elaboration iteration. Based on field notes, the Technical Lead and the developer attributed the code activity peak to de-risking activities. In the context of this thesis, the interviewees referred to the term “de-risking” as the act of eliminating or resolving identified risks. De-risking activities included in elaboration included the highest priority architectural and technical risks defined on the risk list document. This de-risking was accomplished by producing

prototypes and unit tests and will be further detailed in the RUP work products section.

It is interesting to note that during Elaboration iteration 2, 3 and 4, two out of the three developers involved in the project at that time were requisitioned on a part-time basis to assist the projects from case study 1 (Iteration IR6 and IR7 - Figure 4.2) while the business analyst and project manager de-risked business-related issues. Developers were invited to meetings to assist in business-related discussions when deemed necessary. This explains the minimum code activity between Elaboration iterations 2 and 4. The length of iterations decreased as more became known about the project.

This cross-team interaction also proved beneficial as developers leveraged some of the technical lessons learned from the teams in case study 1 about the new enterprise architectural direction.

This code activity shows that for this particular implementation of RUP, developers were actually producing code during the Elaboration phase, thus showing that these two earlier phases were not dedicated to a big design up-front.

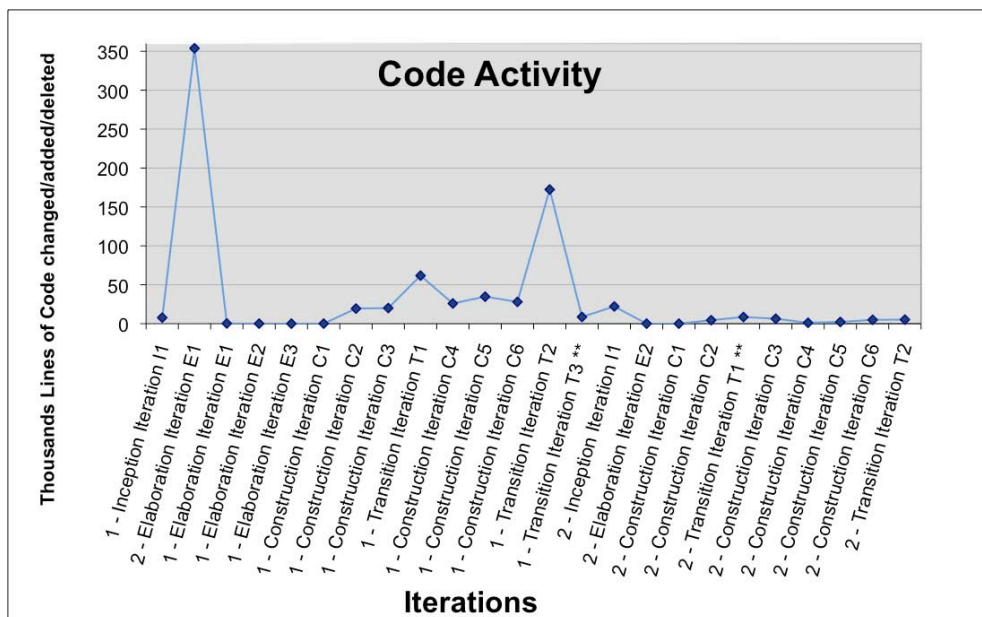


Figure 5.2: Case Study 2 - Code Activity per Iteration

### 5.3.2 RUP work products (artifacts)

The selected work products found for this project included: project vision document, design models, class models, use-case diagrams, software architecture document, iteration plan, iteration assessment, phase assessment, risk list, issues list, test plans and test cases, disaster recovery plan, and release sign-off. From these work products, a project vision document was created during the Inception phase, and design models, class models, use-case diagrams, software architecture document, and risk list were initiated during the Elaboration phase.

The most mentioned work product during interviews was the “risk list” document developed during the Elaboration phase. This list contained all risks identified grouped by business risk due to business processes or technical risk related to development tasks. This list was considered a *“living and evolving document,”* as risks were constantly revisited, completed and reevaluated by all stakeholders. The project team prioritized tasks by risk level; the most risky items were addressed first. A compromise was made to treat technical and business risks equally.

The iteration and phase assessments were mostly used as status reporting documents to upper management rather than as actual reflections of the current state of the project.

The technical lead indicated that the thinking process involved in creating the design models, class models, and the software architecture document proved very useful to the team. He felt that it was beneficial to have the time and chance to think about the overall architectural skeleton before starting development. On the other hand, the RUP engineer enforced a level of detail that was viewed as unnecessary, time consuming, and wasteful. These documents were created and approved but were not kept up-to-date.

A Development Case was not found in source control for this project.

Team sheet		
Role	Availability	Week (hours)
Project Lead	75%	30
Lead Architect	10%	4
Technical Lead	75%	30
Business Analyst	73%	29.2
Developers	75%	30
	75%	30
	75%	30
	73%	29.2
Business Representatives	50%	20
	25%	10
Project Sponsor	10%	1
Test Manager	50%	20
Technical Writer	50%	20

Table 5.2: Case Study 2 - Team composition

### 5.3.3 Role set

Table 5.2 details the composition of this project’s team.

The availability column shows the actual amount of time the resource would spend purely in project-related tasks (total time inhouse minus expected overhead). The hierarchical structure of this government agency divided related support roles to other separate departments as these support roles were shared amongst other projects. These support roles included: the RUP process engineer, enterprise architects, the project management office (PMO), the project management steering committee, enterprise testing team, enterprise database team, and the technical support infrastructure team (TSI).

## 5.4 The content of change

### 5.4.1 On-time Delivery

After the first iteration in the Elaboration phase, the number of unaddressed items on the risk list was a clear indication that the project was more complex than anticipated



by the business clients. This complexity was related to former rigid scope documents that lead business clients to ask “*for everything and their dog*” ahead of time. This rigid scope disallowed business clients to change their minds when needed as there was “*too much red tape*” involved.

But due to the constant interaction between business clients and the project team and the constant demonstration of progress during demos and requirement gathering meeting, trust was re-built. Demos provided windows for business clients to provide on-going feedback and revisit their needs.

As shown in former projects (see case study 1, Chapter 4), business clients would go years from the time they specified a requirement to actually seeing working software. The risk list also provided a more tangible way to better manage and negotiate expectations. After experiencing the delays with projects X, Y, and Z (case study 1), business clients bought into the idea of having a subset of the project delivered first instead of prolonging the delivery timelines. The RUP engineer reinforced the idea that a quicker delivery would also give the business clients the opportunity to revisit their more complex needs once their business process was actually being supported by the newly developed systems.

The project team worked with the business clients to prioritize and define the Minimum Marketable Features (MMF) [73], based on the items on this risk List. At the end of Elaboration Iteration 2, the business clients had defined this MMF and agreed to break the project into two phases, as showed in Table 5.1.

*Were releases delivered on-time after the IID adoption?* Although the project took longer than it first envisioned back in 2005, both project phases were delivered on-time and met the deliverables milestones. This was the first IT Project to be on-time in six years in this corporation.

### 5.4.2 On-budget

The budget for the project was set during inception. This government agency has a set budget given to the IT Department at the beginning of each fiscal year. Business clients present their overall business needs, and based on base-case analysis each project is dedicated a set budget for the year.

We were not allowed to access actual project costs (dollars). However the interviewed business clients and the project manager stated that this project was on-budget. Based on the informal question&answer meetings with one of the developers, no overtime was required or imposed to the project team. This was confirmed by the overall perception of this project in the organization that this was the first IT project to be on-time and on-budget in six years.

Although project costs were not accessible, since bug reports were available, we decided to analyze where in the development process most bugs were being logged for the projects that migrated to RUP (case study 1), and the project that followed RUP since inception (case study 2). This analysis relates to cost as costs associated with fixing bugs later in the process increases in orders of magnitude compared to the cost of fixing them during development [60].

The bugs reports were grouped based on the staging environment where they were discovered. The ascending hierarchical order of staging environment is: Dev (Development), Test (Testing) and or Sbx (Sandbox), UTE (External User Testing Environment), Act (User Acceptance), Prod (Production). A description of each environment is provided on Table 5.3.

A core goal for software quality is to deploy bug-free software. Bugs found before deployment are a sign that the overall process is working. Figure 5.3 shows that during the waterfall days (Pre-RUP), 40% of all bugs were found after a production release. A total of 47% of all bugs were found in the two latest staging environments (Act and

Data Table	
Staging Environment	Description
Dev (Development)	Lowest environment. Used to validate developers builds. first integrated testing environment.
Test (Testing)	Second integrated testing environment. Used by internal testers to test and validate deployments
Sbx (Sandbox)	It replaces the Development environment and is used concurrently with Testing, when available. Some teams used a sandbox machine to deploy and validate builds on a continuous basis.
UTE (External User Acceptance)	External. Used to expose the applications to a set of external focus group users for validation, feedback and testing.
ACT (User Acceptance)	Pre-release. This staging environment mimics the production environment. It contains the current state of the Production server machines, and it is used for User Acceptance and client sign-off for acceptance of a release.
Prod (Production)	Final Environment. This is where the actual application is deployed and used by all external and internal users.

Table 5.3: Staging Environments Descriptions

Prod). Projects X, Y, and Z (case study 1) did not use a Sandbox environment nor did they have a dedicated user testing environment for external testing and validation. Focus group involvement was minimal and was based on scheduled demos with no hands-on testing. The interview with business clients revealed that, in many cases, user acceptance was conducted in a Testing staging environment. This was due to aggressive timelines and deployment issues migrating to production-like environments. During the RUP transition, the percentage of bugs found after a release dropped to 34%, and after the Partial

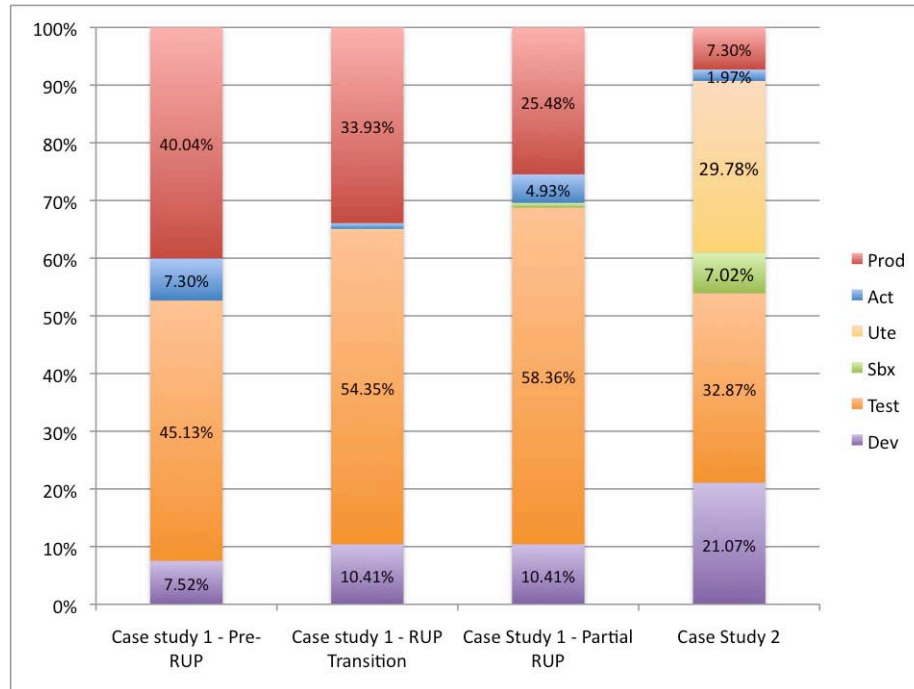


Figure 5.3: Bugs breakdown per Staging Environment - case study 1 and 2 comparison

RUP adoption, the overall numbers dropped to 26%. We do see an improvement in the amount of bugs found after a release (from 47% to 34%).

For Project A, Figure 5.3 shows that only 7% of all bugs were found after a production release. Close to 9% of all bugs were found in the latest staging environments. An interesting difference is that in case study 2 an external user testing environment (UTE) was set up for external focus group participants to test the application. 30% of the bugs reported were found during such testing. As mentioned before, the business clients of these applications were the business area leaders of this government agency, representing requirements for external users.

While projects in case study 1 only mitigated this risk using sporadic focus group demos, project A opened the application to the external users for testing using the UTE environment. Although having personas for a set of stakeholders is better than having no client involvement at all [14], these results are an indication that such compromise

does bring a loss to the quality of the software being delivered.

If costs associated with fixing bugs later in the process indeed increase in orders of magnitude compared to the costs of fixing them during development [60], it follows that the adoption of RUP had a positive effect on project costs aiding the on-budget delivery of project A.

#### 5.4.3 Responsiveness

One of the re-occurring themes from the interviews with business clients in case study 1, was that changes and bug fixes had to go through “*too much red tape*” to be approved and resolved. Based on this perception we decided to measure the bug fixing responsiveness of project A as well as of projects X, Y, and Z (case study 1).

As mentioned above, bug-fixing responsiveness was measured by:

Bug-fixing responsiveness (in days) = Date the bug was last moved to a Close state (action: Close) *minus* the date it was submitted (action: Submit).

Bugs were grouped based on priority. The arithmetic mean (average) in days, median, standard deviation, minimum (min), and maximum number of days (max) that it took to fix a bug were calculated for each group. During the first round of analysis, the max number of days for the groups in case study 1 seemed too high. To investigate this, the ten highest data reports from all of the four priority groups from case study 1 were manually checked. Many bugs had moved from a postponed state to a closed state, indicating that either the bug was too old to still be a real issue or the issue was indirectly addressed by another resolution. To filter these items out, another set of analyse was performed excluding any bug reports that went directly from a postponed state to a close state.

78 items were removed from case study 1 and none from case study 2. Even after

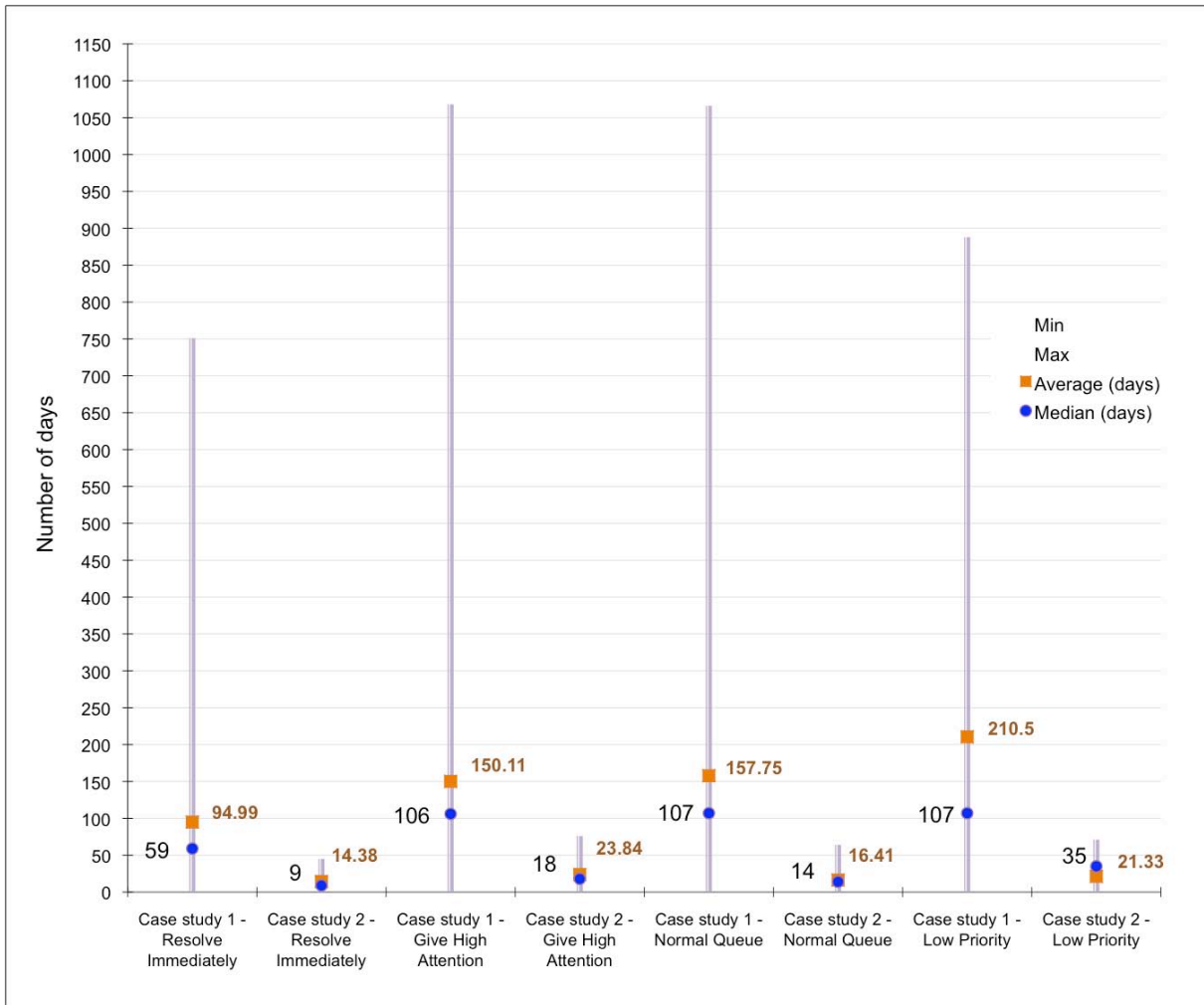


Figure 5.4: Bug fixing responsiveness - days to closure

this filtering which represented close to 8.2% of the bugs in case study 1, the bug fixing responsiveness was still quite different between the two projects absolute numbers. The results of our analysis are shown on Table 5.4 and Figure 5.4.

The data was not normally distributed (normality plot q-q tests and histogram available in Appendix A.3). Table 5.4 and Figure 5.4 show that the standard deviation was always higher than the averages. This high variation in time to fix bugs within the same priority was not a surprise. The level of effort required to fix a bug is not always associated with its business impact and priority. Some bugs required little effort while others were quite complex in nature. This variability should be taken into consideration when

Data Table: Excluding postponed bugs		
	Case Study 1 (days)	Case Study 2 (days)
Resolve Immediately	Avg: 94.99 Std: 128.61 max: 751	Avg: 14.38 Std: 22.97 max: 45
Give High Attention	Avg: 150.11 Std: 165.86 max: 1068	Avg: 23.84 Std: 27.38 max: 76
Normal Queue	Avg: 157.75 Std: 181.06 max: 1066	Avg: 16.41 Std: 19.41 max: 64
Low Priority	Avg: 210.8 Std: 221.82 max: 888	Avg: 21.33 Std: 31.24 max: 71

Table 5.4: Case Study 2 - Data Table for Bug Fixing Responsiveness Days to Closure (Postponed bugs excluded)

trying to balance delivering quantity and quality of bug fixes.

*How long did it take for bugs to get fixed in a new project that started development using IID practices?* After bugs began to be logged by business clients, they were discussed and prioritized during iteration planning meetings. A quick turnaround to address an issue showed respect and concern towards the business clients. The business partners felt they could trust the team to fix issues in a timely manner. If an important bug was not addressed, it could be set as number one priority three weeks later in another iteration. To validate this sentiment, bug-fixing responsiveness was also measured relative to the iteration end dates on Figure 5.5. This was calculated by:

Bug-fixing responsiveness (relative to iteration-end in days) = Date the bug was last moved to a Close state (action: close) *minus* the end date for the iteration in which it was reported. Negative values indicate bugs that were fixed before the iteration end date in which they were reported.

As shown in Figure 5.5, in case study 2, bugs labeled as “Resolve Immediately” were, on average, addressed within the iteration they were found. “Give High Attention” and “Average” bugs were addressed within the next available iteration. The high standard deviation for all of these categories shows once again that the level of effort required to fix a bug is not always associated with its business impact and priority.

Unfortunately that was not true of case study 1 where due to the high volume of bugs introduced before six-week iterations were adopted, existing bug competed against new ones. On average, bugs were fixed well after the end date of releases in which they were found.

Case study 2 managed bug fixes more readily and consistently than in case study 1.

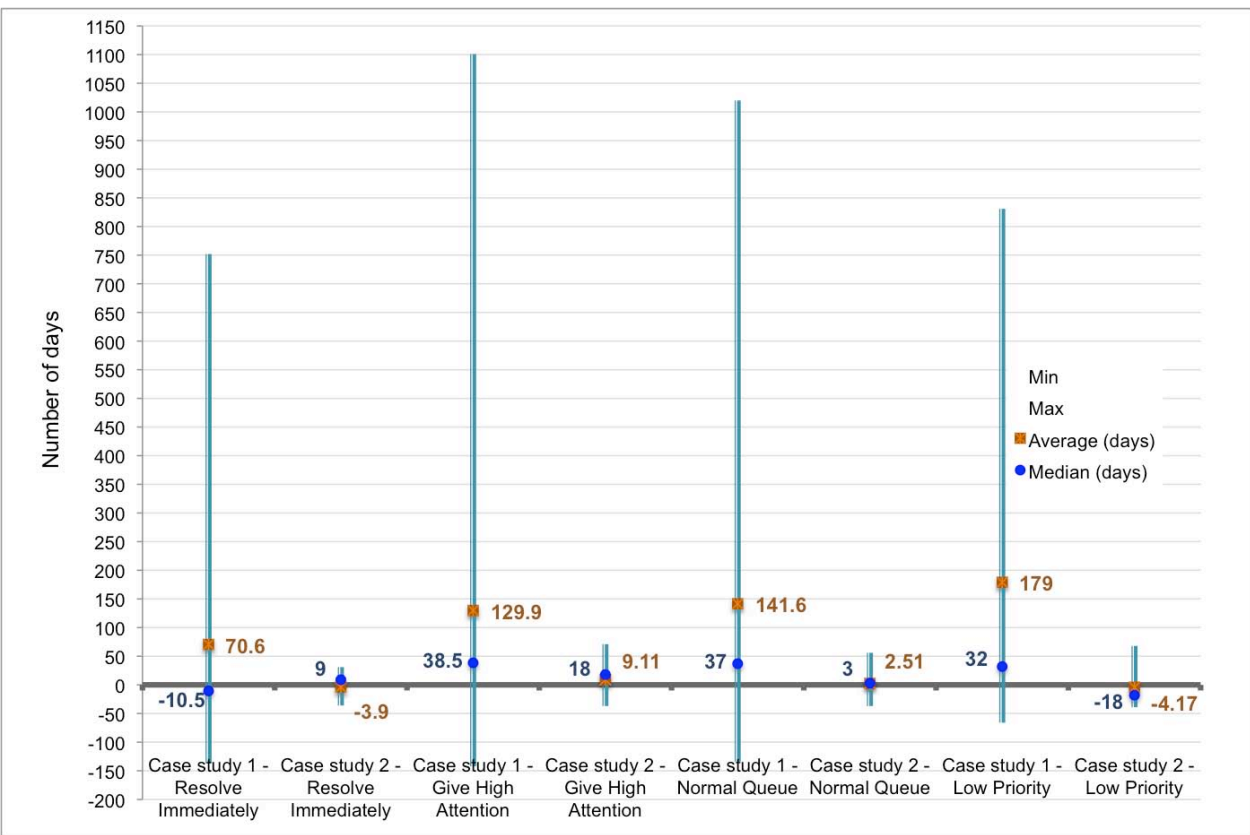


Figure 5.5: Bug fixing responsiveness - days to closure relative to Iteration-end



## 5.5 Challenges

Adopting iterative development in this greenfield project was not uneventful. Iterative development requires a paradigm shift based on trust [16]. Trust was lacking between IT and business clients. This section discusses some of the challenges this team met following the newly adopted RUP process.

The four systems under Project A were developed to support a business process flow. In previous projects, business clients waited months or years from the time the requirements were specified to actually seeing any working software. This also meant that they were always testing a fully developed end-to-end application. Demos were seen as key players in making this team successful. However, during early meetings, the business clients tended to focus more on missing or “to be implemented” features than on the actual features presented. It took time and project management skills to explain to business clients the benefits of seeing atomic pieces of the application early and often.

Due to the iterative nature of this project, the business clients decided to delay hands-on testing until a subset of the system that supported end-to-end business scenarios was ready. The applications worked like workflow engines. The business clients did not want to waste time mimicking data to test an atomic part of the application, especially since features were addressed based on risk not precedence. They felt that once more parts of the system were ready they would need to test the atomic part of the application again. The testing responsibility stayed solely in the hands of business analysts and formal manual testers until Phase 1 Construction Iteration C5. This delayed the discovery of some important issues.

In the early stages of this project (inception and elaboration) the team de-risked requirements by developing unit tests following Test Driven Development (TDD) practices. Test driven development is a design and testing technique. A developer following TDD

would: write a failing test that specifies how the “to-be” function should be created; write the code for the actual function; re-factor the test; code until the test passes and the code is the simplest design possible to solve the problem at hand [14]. However, once the team was satisfied with the level of risk left in the project, developers stopped following the TDD practices. Due to the data-centric nature of the systems, the hard-coding of data for testing made it very time consuming for developers to maintain the test suite. Unit tests were abandoned and stopped being maintained. Today, the lack of unit test coverage presents a problem to the maintainability of these systems.

The team had its own “sandbox” environment to speed up delivery for formal testers. Although this solved the problem for testers, it still did not address issues with system integration. These projects were deployed to staging environments manually. Each iteration took many hours to deal with configuration issues and migration problems, as did the projects presented in case study 1 (see the “Challenges” section of Chapter 4).

Daily stand-up meetings only involved the development team. Also, since this was a new concept to many team members, for the first few weeks the meeting took approximately one hour. Beck [14] followed the Scrum idea of a stand-up meeting that should be quick and should only focus on three questions: what have I accomplished since yesterday, what will I accomplish today, what roadblocks do I have. One of the developers brought this to the attention of the technical lead, and daily stand-up meetings were streamlined to last approximately 15 minutes. Business clients were included in these daily stand-up meetings.

## 5.6 Lessons-learned

We would like to highlight the main process-content factors that supported this project in achieving a successful status in the eyes of the business clients:

- Short iteration cycles

The ability to revisit and address issues in the risk list every iteration cycle provided visibility into project unknowns, complexity, and missed requirements. Short iteration cycles enabled the team to “fail early” [14, 25], learn from demo feedbacks, and streamline their testing and delivery processes.

- Scheduled iteration testing

The code being delivered every iteration went through a round of formal testing during the last week of the iteration. Also, the business clients knew when they would be able to test the application, and how long they would have to do so.

- Risk Management, early prototyping, and unit tests

Addressing a combination of technical and business risks early in the process by early prototyping and unit tests and also enabling stakeholders to make key decisions. For example the decision to divide the project into two phases leading to on-time deliverable milestones.

- Iteration end demos

Iteration end demos provided constant demonstrations of progress. Demos enabled business clients to provide feedback and influence the project outcome early and often. This builds common ownership and trust between team members and business clients.

- Iteration planning

Iteration planning provided another window for business clients’ feedback on tasks priorities and risk management.

- External user acceptance testing through focus groups

Mitigating the risk of having business clients acting as personas for external users. This was achieved by proving to a focus group of external business clients the ability to test the application. 30% of bugs were found by such external users avoiding problems in the production environments.

- Quick bug-fixing responsiveness

Addressing issues in a timely manner demonstrated to business clients that the team was effective and cared about their concerns, and avoided quality related issues later in the development cycle. The projects that followed the IID process since inception managed bug fixes more readily and consistently than the projects where IID was introduced in later stages. This shows that while Iterative and Incremental approaches help, they do not perform miracles. There were a large number of bugs introduced in case study 1 before moving to an iterative and incremental approach, and any new bugs introduced competed with existing ones in the completion queue. Existing problems, such as poor quality in the case of case study 1, will take time to resolve and will have other long lasting effects even after process improvements are introduced. It is important to remain realistic with expectations for project outcomes.

Daily stand-up meetings, iteration planning activities, and demos opened the communication channels between members of this team. Constant communication fostered trust. Trust, in turn fostered a healthy, fun environment for the team members [40]. Those interviewed said that they had a lot of fun working on this project. The team also attributed this to the project manager skills of *“making a team out of a group of unrelated people”*.

## 5.7 Limitations

In this case study we report on one organization’s experience adopting IID practices in a new project. Care should be taken in drawing general conclusions based on our results. However, we believe that our findings can be valuable for organizations facing similar changes, especially when aggregated with other findings [64]. Exploring the consequences of the new development processes using bug-fixing data is valuable because a core goal for software quality is to deploy bug-free software. Also, bugs found before deployment and quick turn around time for fixing bugs are signs that the overall process is working. However, accurately capturing all of the consequences of the changes made in this organization would require a wider range of data (e.g., total development cost) than we are able to report on at this time. This limitation is mitigated by our qualitative data that fills in some of the gaps.

## 5.8 Research goals addressed

This chapter explored the IID adoption in the context of a new project. More specifically, it addressed the remaining primary research goals - how IID affected poor bug-fixing responsiveness (research goal 3) and delivery delays (research goal 4) - using comparative analysis with several existing projects, presented in case study 1.

Through the analysis of both quantitative bug-fixing data and qualitative data gathered during interviews, we discovered that IID practices allowed a new project to:

- Provide support for managerial decisions that lead this project to be the first project in six years to delivery on-time and on-budget.
- Avoid quality and stability issues.
- Provide better bug-fixing responsiveness than projects that migrated to IID.

Specific practices included: short iterations, iteration planning activities, scheduled iteration testing, iteration end demos, risk management, early prototyping, and external user acceptance testing through focus groups.

The quantitative study of bug-fixing responsiveness also showed that the IID adoption for the existing projects studied under case study 1 only provided limited improvements to bug-fixing responsiveness. Existing bugs competed against new ones in the priority queue. On the other hand, IID practices in both case studies allowed the projects to detect bugs earlier in the development lifecycle.

This chapter also provided some experiences and practical lessons learned from the project following IID practices since inception. They include:

- The shift to IID presented some paradigm shifts for the business clients. During early iteration end-demos the business clients tended to focus more on missing or “to be implemented” features than on the actual features presented.
- The business clients of the first fully IID-developed system delayed hands-on acceptance testing until bigger portions of the systems were completed. These clients were used to receiving an end-to-end system to test in the former Waterfall process. The business clients did not want to waste time mimicking data to test an atomic part of the application. They felt that once the mock part was ready it would need to be tested again. This delayed the discovery of some importance issues.
- It took time for the teams to understand that the daily stand-up meeting is supposed to be short, focused, and simple.

## Chapter 6

### A test automation strategy - an active research approach

This chapter describes an action research project with the goal of addressing two issues unresolved by the adoption of IID: staging environment instability and lack of support for test automation. These issues were identified during the interviews and questions&answers sessions conducted in case study 1 and case study 2. Specifically, we worked with one project (five systems) to introduce automated regression testing along with a novel type of testing which we termed Environment Configuration Testing.

Action Research has been increasingly used in the fields of Information Systems [34]. This chapter follows the principles and guidelines for researchers using Canonical Action Research (CAR) provided by Davidson *et al.* [24]. Action research provides methods to combine theory and practice in an industrial research setting. An overview of CAR principles is provided in Appendix B.

Following the Cyclical Process Model (CPM) was natural to our research goals. The empirical studies presented in Chapter 4 and Chapter 5 provided two cycles of pre-entrance diagnosis. The last section of Chapter 4 illustrates the challenges encountered by a group of existing projects that migrated to IID practices. The last section of Chapter 5 illustrates the challenges encountered by one new project that followed IID practices from inception. The term “challenge” in this context includes technical and process-related issues from the viewpoint of developers, such as abandoning TDD practices.

The challenges identified in case study 1 and case study 2 revealed a set of common problems. The common problems were related to staging environment instability and the lack of support for test and build automation. Figure 6.1 illustrates these commonalities.

The lead tester confirmed that this set of problems was indeed their biggest “hurt

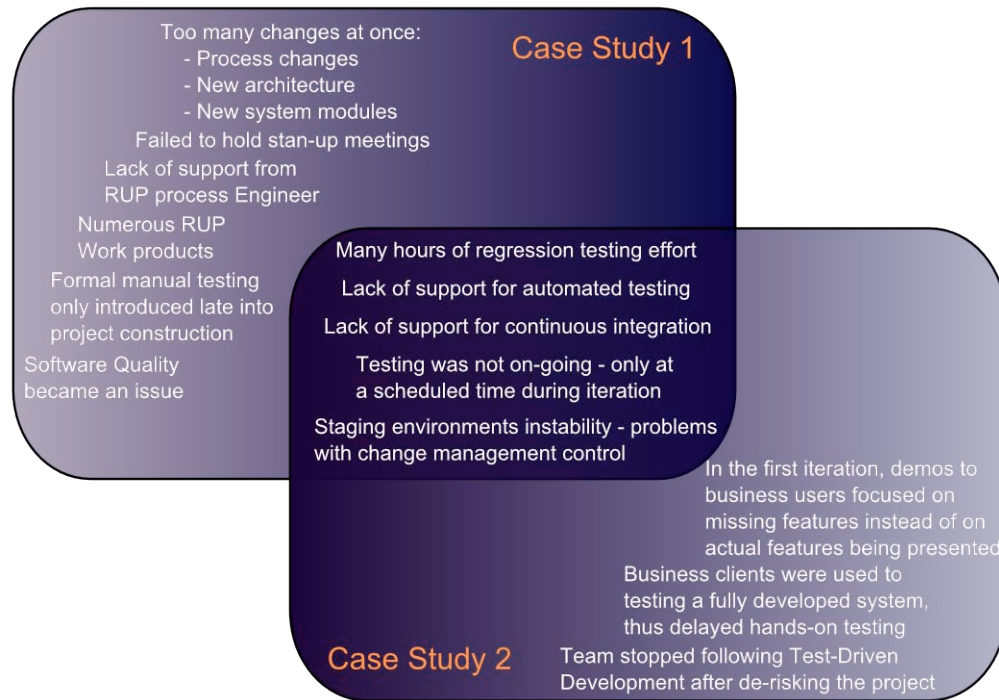


Figure 6.1: Challenges encountered during the RUP adoption by Case Study 1 and Case Study 2

area” in configuration management and testing. A testing automation strategy would give us the opportunity to add extra value to our industrial partner and to provide another contribution to the community of practice. The testing strategy needed to address:

1. Two technical issues: staging environment stability and lack of appropriate regression testing;
2. One organization issue: how to gain organizational visibility and support to implement automation techniques.

A preliminary research strategy was proposed for the projects involved in case study 1 (Project X, Y and Z - Figure 2.2). As mentioned in Chapter 1, a Researcher-Client Agreement (RCA) was created, approved and signed in August of 2007. The client, the government agency under study, provided explicit authorization to use bug report



data, available project metrics, and authorization to interview involved team members. The project manager and technical lead gave us their agreement and support to proceed with the implementation of the proposed test strategy. We estimated that two rounds of Action taking and reflection (CPM) would be needed, one to address the technical issues, and the second to address the organization issue. As such, we embarked on another round of diagnosis to gather more specifics about these hurt areas.

## 6.1 Diagnosis - Understanding the hurt areas

*“We used to joke around saying what is the point? I open it [the application] and get the yellow screen of death, so you are just wasting my time!”* (Business client presenting sentiments about testing applications in different staging environments. It also express frustrations over the lack of appropriate testing done by the project team before user acceptance testing.)

*“[Testing] is not my full time job. I need to deal with core business. Testing work is supposed to be on the side, but [at that point] becomes full time work. I am basically doing two fulltime jobs, which makes things difficult.”* (Business client expressing frustration over accumulated testing effort at the end of a waterfall release cycle. Many of the delays in testing were also due to delays in deployment due to technical issues in staging environments.)

*“You would get it [the application] for two days, and you need to approve it and its got to go.”* (Business client expressing frustration over rushed testing timelines.)

These are some of the verbatim quotes that really stood out from the data collected from formal interviews with business clients presented in Chapters 4 and 5. After the approval from the project manager and test lead, two rounds of question&answer sessions were conducted with available developers, the team leader, and the testing team. These

sessions were designed to identify more details about the hurt areas identified from case study 1 and case study 2.

### 6.1.1 Hurt area 1 - Environment configuration instability

In the context of this thesis configurations are:

- Settings that must be applied to servers,
- Application-related settings stored in files (xml) or databases,
- Security-related permissions that need to be applied to servers, databases, and other supporting machines and/or applications.

Staging environments are in constant change. In larger organizations that support multiple concurrent projects, it is common to find different projects under different schedules but all using the same staging environments to deploy and test the applications. Changes to server configurations made by one team have the potential to impact other teams. This is certainly true for this government agency. All projects share the same staging servers that integrate multiple software systems. There are many staging environments as described in case study 2, Chapter 5. All environments have between two and five servers each, and all these servers need to be configured and tested. On many occasions, successfully deployed applications start failing, and teams were not made aware of what actually changed in the environment.

As a result, teams had to constantly and manually verify configuration to ensure that all different environments were set-up correctly for their application. This manual check of configurations was very time consuming. In the presence of automated deployment scripts, one may suggest that re-deploying the application would be an easier way to solve these issues. However automatic deployments still require core shared services (i.e. COM+, Internet Information Services), and application pools to be shut down and thus

introduce disruptions to other teams that also have tight schedules and can not afford constant downtimes. Automated deployments also do not report which configuration was misconfigured or changed by another team.

When we started this diagnosis cycle, the projects under study had created semi-automated build scripts, consisting of Microsoft Installer files (msi) or simple Visual Build scripts. The word semi-automated means that these scripts do not pull the source code from any source control repository. Instead the scripts copied existing compiled assemblies and files to desired locations. They also did not set server configurations, which were left to be done manually.

It is fair to say that deployment to the staging environments was mostly manual and done by a person following a set of written step-by-step instructions. On many occasions, especially after deployments, parts of the application would simply not function due to missed or changed configurations in the servers.

The business clients also expressed frustration over testing in these different staging environments. They would successfully test the application in one staging environment, but when testing the application in next staging environment, the application would start failing or “acting weird”. This diminished the business clients’ confidence in the overall quality of the release.

The teams from Project X, Y and Z (case study 1) spent numerous hours per release cycle fighting these configuration issues as they were not simple to diagnose. This was especially true in higher staging environments such as user acceptance or production, as the project team did not have access to the servers. Servers were locked down for security reasons.

### 6.1.2 Hurt area 2 - Incomplete manual regression testing due to resource and time limitations

The existing manual testing team was a shared resource between many project teams, and the level of effort involved in manually regression testing the application was very high. As a result, complete regression testing was often not accomplished. Testers needed to focus on repetitive long manual tasks, instead of investing time in deeper more complex layers of testing [17, 33].

As mentioned in case study 2, Chapter 5, due to the data-centric nature of the systems, the hard-coding of data for unit testing made it very time consuming for developers to maintain the test suite. Management did not support the time required to evolve the unit testing strategy. Unit tests were abandoned and were no longer maintained. The lack of unit test coverage presented a problem to the maintainability of these systems. Developers made code changes, and due to the lack of unit testing coverage, they did not know if they had broken existing functionality. As a result, new features and bug fixes often resulted in new faults in existing parts of the system. Previously fixed faults frequently re-appeared in production.

In summary [67]:

**Context - the why of the action research project.** This agency supported multiple ongoing software projects, having a mix of automated and manual software deployments onto staging environments. The fine line between enterprise-wide configuration management of such environments and the configuration needs of individual project teams was a grey one. Changes to the environment by different teams often led to instabilities and rework effort. Also, there was no support and budget for test automation coming from upper management. The development of new features was seen as adding more direct business value.

**Process - how the action research project will address the context.**

The design and implementation of a testing strategy: environment configuration tests and back-end functional tests.

**Content - what the action research aims at changing.** Staging environment instability and lack of support for test automation to aid regression testing efforts.

## 6.2 Action planning - designing test strategies to address staging environment stability and lack of appropriate regression testing

*“A sound testing process is helpful for successful test automation, but an appropriate test automation strategy is vital, that defines which test types are to be performed on which test level, and which tests are automated and/or supported by tools” [17].*

The main goal of this round of planning was to develop a testing strategy to provide repeatable, automated, and systematic test suites to address the top two technical “hurt areas” for the project teams: staging environment instability and lack of appropriate regression testing.

To address these issues, we investigated a strategy to implement a suite of non-functional environment configuration tests, and a suite of functional regression tests. To build a sound testing strategy, the investigation needed to include a satisfiable definition of: test adequacy criterion, test oracles, testware tools and error seeding techniques.

**Test adequacy criterion** is the definition of the scope of a testing strategy; what should you test and when should you stop testing. Goodenough & Gerhart [37] define test adequacy criterion as “what properties of a program must be exercised to constitute a thorough test”, for example, which properties of a configuration can be tested so that a successful execution would imply no errors in the execution of that configuration. A question remains if Goodenough’s last name derives from his theory or

vice-versa.

**A test oracle** defines what should be the outcome of a test execution - how should the system under test behave given the data being tested [33].

**A testware tool** is the software harness chosen to execute a suite of test cases. It enables the automatic execution of a suite of test cases [33].

**Error seeding or mutation** is the process of purposely adding faults to the system under test to validate that your test cases are effective in finding faults.

### 6.2.1 Technical hurt area 1: Environment Configuration

Here we present the techniques used to design a non-functional environment configuration test suite. To the best of our knowledge, there is no related literature that addresses the automated verification of staging environment configuration as a way to consolidate configurations.

Test adequacy criterion: The interviewed developers and the lead tester identified the category of application configurations that caused most of the issues. An error-based (most likely to cause faults) testing criterion for configurations proved to be the best fit. The test suite should cover:

1. Folder permissions,
2. Database (SQL Server) stored procedures permissions,
3. Availability of required services,
4. COM+ dll registration and identification,
5. Universal Data Link (.udl) files,
6. Web Configuration files latest updates,
7. Internet Information Services (IIS) settings,

8. Network groups, machine users, and machine groups,
9. Global Assembly Configuration (GAC) validation,
10. Database (SQL server) users, database roles and roles memberships.

Test Oracle: All teams are required to maintain an up-to-date Disaster Recovery Plan System Manual (DRP). This document was one of the few documents kept relatively up-to-date in source control due to audit requirements. The DRP provides a high-level explanation of how to configure each individual application. The DRP is a lengthy document, but due to the complex nature of the systems, it does not list some required configurations (some examples include: required database permissions, folder permissions, third party shared component dependencies, and some component settings in COM+). Outdated or missing configurations were gathered from deployed server configurations. External dependencies, such as services, stored procedures, and database users were gathered through code inspections. The DRP document, manual server look-ups, and code inspections were used to gather the information required to verify the correct/faulty execution of a test case (implementation details are provided in section 6.3.1).

Chosen testware: NUnit version 2.2.6 [72] was the chosen testware. Other project teams in the company were using the software and this meant that the long bureaucratic approval process from the technical enterprise team could be avoided. Although NUnit is primarily used as an unit test harness, it is simple to install, use, and maintain. It was important to follow the path of least resistance due to existing contextual views of management towards test automation.

Error seeding (mutation): We manually injected errors into the configurations (during mutation testing only) to assess the fault detection effectiveness of the test suite. Some example of the mutations performed included:

- Folder Permissions - verify that certain users have privileges on given folders. Using Windows Explorer, we removed or changed permissions. Existing tests detected all seeded faults for every execution.
- Database (SQL Server) stored procedures permissions - verify that all stored procedures being used by the application have the necessary permissions. Using Microsoft SQL Server Management Studio, we removed permissions. Existing tests detected all seeded faults for every execution.
- Availability of required services - verify that the necessary services are running. Using Microsoft Computer Management Console, stopped, disabled, or paused the services. Existing tests detected all seeded faults for every execution.
- Universal Data Link (.udl) Files - verify that required udl files are able to connect to the Microsoft SQL Server Database. Changed the connection string values inside of the udl file. Existing tests detected all seeded faults for every execution.

### 6.2.2 Technical hurt area 2: lack of regression testing coverage

*“Providing valued customers with lots of new and exciting features is of no benefit if the upgrade process manages to also break features they have been using successfully for the last several years. Regression testing should catch these newly introduced defects. However, despite its recognized importance, regression testing remains the testing phase often given the least attention in the planning of test activities”* [27].

*“Establishing a policy for regular regression testing is key to achieving successful, reliable, and predictable software development projects”* [47].

The functional testing approach was twofold. The system under test (SUT) is the largest and most frequently used web application (Project Z). We developed an XML-driven NUnit functional test suite (gray-box testing) and a suite of automated scenario-



based regression testing (requirements driven black-box testing) using GUI testing with business rules verification points. First, we present the common planning characteristics of these two approaches.

Test adequacy criterion: the testing team performs manual regression testing using paper copies (most recent PDF documents) of scenarios to enter test input data into the application. The test lead identified 52 complex scenarios covering an estimated 80% of total business flows within the application. This rough estimate was based on informal analysis of the test lead knowledgeable experience of the manual business flows he needed to test. Each scenario represented an end-to-end implementation of a business use-case. It was important to identify a significant number of test cases, small enough to save time and money, while keeping the test suite within our time and resource constraints. The experienced test team identified twelve test scenarios out of the 52 mentioned above for testing. The three main generic business flows were created and included actions such as: create a new request, save an incomplete request, submit a complete and validated request.

Test Oracle: The manual test scripts (in paper or PDF format) provided the preconditions and post-conditions needed to verify the functional tests.

Chosen Tools - testware: NUnit version 2.2.6 [72] was chosen for the same reasons previously stated. In addition, Rational Functional Tester 6.1 [1] was used as it was available as part of the Rational Tools Suite purchased with the adoption of the Rational Unified Process.

Error seeding (mutation): The original intention was to run tests against older faulty versions of the system to determine if the test cases could find known existing errors. Unfortunately, this was not possible due to database changes that prevented previous systems versions from running. Setting up sandbox environments to perform such error validation was too time consuming. Thus, no error seeding was implemented to test the

effectiveness of the test suite. Instead, we trusted the judgment of the skilled manual testing team [33] as they provided input as to which twelve critical scenarios to automate.

## 6.3 Action taking and Evaluation (round 1)

After defining the testing strategy, we implemented three suites of automated tests.

### 6.3.1 Non-functional Environment Configuration Testing

In this thesis, we refer to test fixtures as parts of the code needed in order to run the four phases of a test in the NUnit framework (i.e. set up, exercise, verify, and tear down). Test cases were grouped into ten test fixtures based on the type of configuration they were trying to validate: folder permissions, database stored procedures permissions, services availability, COM+, universal data links, web configuration files, IIS settings, network and machine user membership, global assembly cache, and database security. These fixtures were coded with:

- Generic private functions that could “exercise” a predefined behavior. For example: to check if a folder has been granted a determined set of permissions. This generic function takes as input a folder path, and the permission set.
- Test cases executing a call to these generic functions passing in specific test inputs and performing assertions to “verify” that the return values match the expected values. Test inputs and expected return values were declaratively specified in XML files loaded during the test “set up.”
- XML inputs and expected values. Configurations are environment specific. They refer to different server paths, different domain accounts, and different namespaces for components and services. Before running, a test case reads an environment ID. Test inputs and expected values (oracle) are grouped by this environment ID in

each test fixture specific XML file. A “helper” class was created to assist in reading these declarative test inputs (making extensive use of Xpath). This provided an easy way to point the test suite to the desirable environment. It also provided a maintainable way to catalog environment-specific test cases.

```
<TestConfiguration>
  <Name>Verify File Permission</Name>
  <Description><![CDATA[Verifies that the file
permission matches the expected value.]]></Description>
  <Environment name="DEV">
    <Server name="xxx">
      <BuildPackage name="xxx">
        <File name="TAG">
          <Path>\\xxx</Path>
          <expectedValue>
            <Type>FilePermission</Type>
            <Value>Modify</Value>
            <AccountName>userDEV</AccountName>
            <Check>True</Check>
          </expectedValue>
        </File>
      </BuildPackage>
    </Server>
  </Environment>
  <Environment name="PROD" >
    <Server name="xxx">
      <BuildPackage name="xxx">
        <File name="TAG">
          <Path>\\xxx</Path>
          <expectedValue>
            <Type>FilePermission</Type>
            <Value>Modify</Value>
            <AccountName>userPROD</AccountName>
            <Check>True</Check>
          </expectedValue>
        </File>
      </BuildPackage>|
```

Figure 6.2: Example of the XML test inputs and oracle

Figure 6.2 shows an example of the XML input (sensitive data has been replaced by “xxx”). The expectedValue node contains the expected result for the test case execution and is used for assertions in NUnit. In this example, a folder used for downloading

attachments in a web application is given a tag (File name=TAG). This folder has a path in the development staging environment (DEV) and it needs modify permissions granted to a network user called userDEV. The folder path acts as input while the permission and user pair are the expected values (oracle) for the test case execution. A folder that serves the same purpose in the production environment (PROD) is given the same tag (File name=TAG). In PROD this folder has a different path and it needs modify permissions granted to a different network user called userPROD.

Without the declarative XML inputs and expected values, we need to code one test case per staging environment (in our context at least five) to test that the download folder (TAG) has the correct permissions. Functionally speaking, the code executed to check the folder permission is the same for different test cases (a download folder, a temporary folder). We simply need a different set of test input/oracle for the different folders and different staging environments. The pseudo-code for a folder permission execution is illustrated in Figure 6.3.

**Preliminary Evaluation.** When run against the development staging environment, the most instable of the staging environments, the suite successfully found two root causes of staging environment instability: missing stored procedure and file system folder permissions. The suite also detected:

- Three changes to subsystems executables (dll files) in the course of seven days. This was an important indicator to highlight the amount of changes actually happening in staging environments.
- Two virtual directories in IIS pointing to the incorrect .NET framework version. Some investigation later showed that some team's deployment scripts were overwriting IIS settings - another root cause for staging environment instability of the systems under study.

```

<SetUp()> Public Sub Init()
    'Read target environment Id from a parent XML
End Sub
-----
Private Sub GetInput(ByVal tagname, ByVal server)
    'Uses xpath to read user account name and
    'expected permission from FilePermission XML
End Sub
-----
Private Function FolderPermission(ByVal folderpath, _
ByVal user, ByVal expectedPermission) As Boolean
    'Returns if the folder has the permission
    'granted to the given user
End Function
-----
<Test()> Public Sub CheckTAGFolderSecurity()
    'GetInput("TAG", server)
    'Assert.IsTrue(FolderPermission (XXX,user,permission,
    '\User does not have privileges(")')
End Sub
-----
<TestFixtureTearDown()> Public Sub FinalCleanup()
    'Delete in-memory resources.
End Sub

```

Figure 6.3: Pseudo-code for a folder permission test execution.

The test suite proved to be effective in detecting both seeded and real environment configuration issues.

### 6.3.2 Functional System Regression Testing

This section is divided into two parts: one for the NUnit function tests and another for the GUI scenario-based regression testing, both designed to address the lack of automated system functional regression testing.

#### **Back-end NUnit functional tests.**

Project Z is a distributed web application. Distributed applications are divided in many layers located in different network locations (process boundaries). A Data Transfer Object (DTO) is a data container that can be encrypted and moved across these different process boundaries. At any given point, the interaction of a user with this web application results in a call to back-end processes (subsystems) that expose methods with DTO(s)

as inputs.

This data-centric web application acts as the data provider to other applications. It requires many data fields to be completed and validated. The hard coding of all these data fields makes test case specification onerous and hard to maintain. Through our interviews and data gathering sessions, we discovered that, at one point during the construction stages, the development team of the project from case study 2 (Chapter 5) did try to implement unit tests for their systems but only a few modules were created before the hard coding of data input and repeating of code caused the suite to be abandoned in favor of more pressing system features [17, 88]. These test suites no longer run against the application.

To avoid similar hard coding issues, our goal was to mimic the interaction of a user with this web application by generating the data transfer objects and exercising the back-end methods directly from NUnit.

1. Again, we refer to test fixtures as parts of the code needed in order to run the four phases of a test (set up, exercise, verify, tear down) in NUnit. A set of test fixtures was created to exercise back-end processes. Test cases mimic the flow of the three main states of a user interaction: create a new request, save an incomplete request, and submit a complete request. The pseudo-code for a test execution is illustrated in Figure 6.4:
2. The approach was to generate XML files that contained the data a user would input into the application during a specific action (i.e. create a request). We designed an engine that automatically translated these XML files into data transfer objects (DTO, see Figure 6.6) during the “set up” of a test case execution. Figure 6.5 shows an example of a XML file. We replaced sensitive data with “xxx” values for privacy reasons. Each tag corresponds to a property name on a data transfer

```

<SetUp()> Public Sub Init()
    'Load XML file
End Sub
Private Sub CreateDataTransferObject(ByVal XML)
    'Creates data transfer object(DTO) from XML
    'Call subsystem to create and return an empty DTO
    'Verify call success
    'Populate object with XML
End Sub
<Test()> Public Sub TestSubsystemFunction(ByVal DataTransferObject)
    'Call subsystem with populated DTO
    'Exercise specific subsystem functions (ie. save, validate)
    'Verify call success
    'Assert.IsTrue(Subsystem call returns,"Call to subsystem failed")
    'Verify state of returning DTO
    'Assert.AreEqual(DTO.property,expectedProperty,"DTO property is incorrect")
End Sub
<TestFixtureTearDown()> Public Sub FinalCleanup()
    'Delete in-memory resources.
End Sub

```

Figure 6.4: Pseudo-code for a back-end functional test execution.

object. For example, the “create request” data transfer object has a parent object called “application\_type” and a child object called “application\_type\_location.” The XML in Figure 6.5 represents the data a particular test needs to populate into the data transfer object to test some specific business scenario: “create request with application type xxx.”

The populated data transfer objects were used as input to call back-end processes (subsystems) from the System Under Test (SUT). For example, to exercise the “create a new request” business flow, the test case XML would only contain data about the user identity and the type of request being created. The XML for the “submit a complete request” flow would include all the mandatory data fields to satisfy all the business rule validations exercised by the applications business logic in the subsystem. These XML files acted as snapshots of the data contained in a data transfer object during an user-application interaction.

```

<Application_Type>
  <row index="0" isAvailable="True">
    <ApplicationNumber></ApplicationNumber>
    <ApplicationTypeSeq>1</ApplicationTypeSeq>
    <ApplicationCategory>xxx xxx</ApplicationCategory>
    <ApplicationType>xxx xxx</ApplicationType>
    <ApplicationTypeAmendedInd>False</ApplicationTypeAmendedInd>
    <ElectricFacilityNumber></ElectricFacilityNumber>
    <NumberOfCopiesReceived>1</NumberOfCopiesReceived>
    <BasedOnAct></BasedOnAct>
    <BasedOnSectionOfAct></BasedOnSectionOfAct>
    <ScheduleName>xxx xxx xxx xxxx</ScheduleName>
    <ScheduleValidIND>True</ScheduleValidIND>
    <InternalApplicationType></InternalApplicationType>
    <Application_Type_Location>
      <row index="0" isAvailable="True">
        <ApplicationNumber></ApplicationNumber>
        <ApplicationTypeSeq></ApplicationTypeSeq>
        <LocationSeq></LocationSeq>
        <LocationType>xxx_Location</LocationType>
        <LocationTownship>11</LocationTownship>
        <LocationMeridian>5</LocationMeridian>
        <LocationRange>5</LocationRange>
        <LocationSection>20</LocationSection>
        <LocationLSD>16</LocationLSD>
        <LocationQuarter></LocationQuarter>
        <LocationException></LocationException>
        <LocationLatitude></LocationLatitude>
        <LocationLongitude></LocationLongitude>
        <LocationEventSequence></LocationEventSequence>
        <LocationNorthSouthCode></LocationNorthSouthCode>
      </row>
    </Application_Type_Location>
  </row>
</Application_Type>

```

Figure 6.5: Example of the XML functional inputs

3. One of the team's developers created a tool to generate such XML files based on existing requests in the production database. This made it easy to create as many test scenarios as necessary.
4. All these XML files were saved into specific folders. These folders were passed to the NUnit test suite as an input parameter.

*Initial Evaluation:* On its first execution in the integrated development environment, the functional test suite found a problem with the document uploading module due to



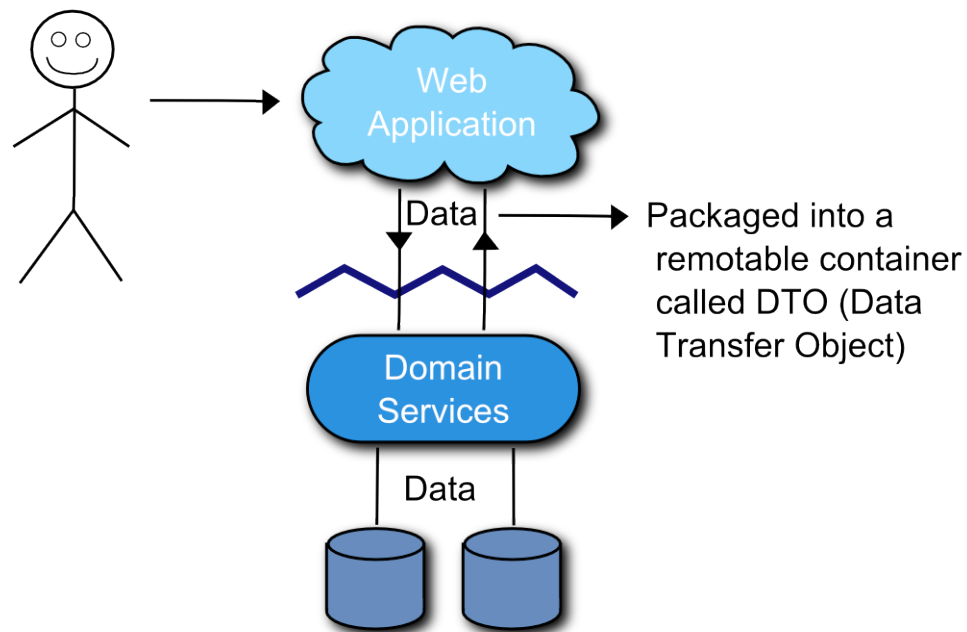


Figure 6.6: Example of a data transfer object (DTO)

missing referential data in the database. This data was overlooked during manual testing, but caught by one of the XML-driven test cases.

**GUI Playback Testing.** The goal was to conduct black-box system testing of the user interface automatically. Twelve manual test case scenarios were prioritized for automation. This prioritization was based on the testers' perspectives of which scenarios cover most relevant business flows. This follows the suggestions of Fewster and Graham [33].

Twelve playback scripts were created with business rules verification points. These verification points would check values in the user interface to ensure the state of the request being automatically created was correct.

*Initial Evaluation:* To provide concrete measures of efficiency in order to gain support for automation, we compared the average time taken to manually execute a test against the average time taken to run the Rational Function Tester script. The Rational

Functional Tester scripts ran in 23.55% of the time the manual tests take to run. The Incident Reporting database (IBM Rational ClearQuest) was queried to look for active incident reports for the applications under test. 10 incident reports were selected and the test suite was executed to see if it revealed any of the logged incidents. The test suite did not execute any of the faults logged against the latest version of the software. To understand this result, the testing lead was contacted. He clarified that the twelve scenarios automated were the most business-critical scenarios (most often used by users), and would cause the most disruption to business processes if they failed. They were not the most prone to find existing errors. Due to time constraints, we did not implement more tests to execute the 10 incidents reports mentioned above.

#### 6.4 Action taking and Evaluation (round 2)

Test automation often fails because the automated tests are not run frequently enough by the team and test results are not visible to all the stakeholders [17, 33]. Beck suggests that someone in the team must be responsible for executing tests frequently and for publishing the results to all team members [14]. We decided to automate the execution of the test suites on a scheduled basis and publish the results on the intranet.

NUnit provides console commands to export test run results into XML files. A batch file was created to call the non-functional environment configuration tests (at 6:00 AM) and another file to call the back-end NUnit functional tests (at 6:30 AM). The environment configuration tests ran first as they validate the configurations needed for the back-end functional tests to complete successfully. No automation was provided for the GUI scenario-based tests due to some limitations of the Rational Functional Tester tool.

Due to its configuration verification nature, the test process had to be run using a user account with enough system privileges to read and access the servers. To provide an

easier and more pleasant presentation of the results, an Extensible Stylesheet Language (XSL) style sheet was generated to display the results in an “Environment Forecast” webpage, presented in Figure 6.7.

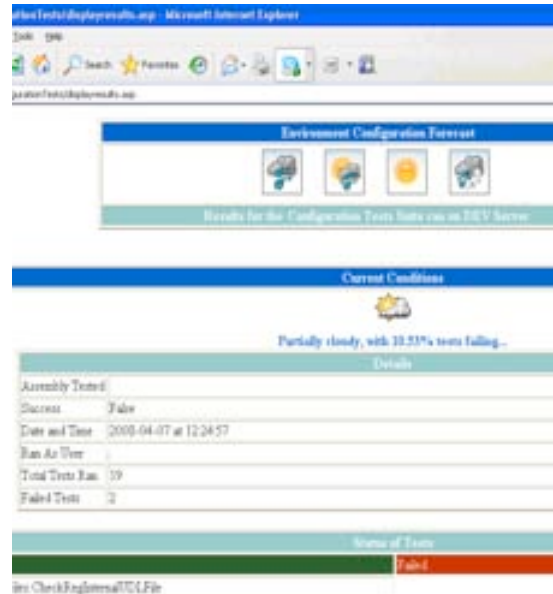


Figure 6.7: A web page showing the results of environmental configuration testing.

A sunny forecast indicates that all tests are passing. A mostly sunny with occasional clouds forecast indicates that less than 25% are failing. A cloudy forecast indicates that 25-50% are failing. And finally a stormy weather icon is displayed if more than 50% of the tests are failing. To provide further details about the environments, information logged during tests (such as the parameters being tested) were exported to a text file. A link to this detailed logging file was displayed at the bottom of the “Environment Forecast” web page (Figure 6.7). Once the test run was completed, e-mails were sent, with a link to the results webpage, to interested individuals including the project lead, testing lead, and developers.

At first, we only had access to run the environmental and functional tests against the development and testing staging environments. We ran these tests for a period of 6 weeks, and based on the test run results, the following issues were identified as root

causes for environment instability:

1. Database security: Many stored procedures had the incorrect level of permission, in most cases more permissions than necessary. This was brought to the attention of the enterprise architecture team. The data architecture team was charged by the enterprise architecture to revisit the security models in order to tighten up the database permissions.
2. Network Folder security: Many folders had more permissions than necessary in the development and test environments. This resulted in certain problems only appearing in higher staging environments, where security was better enforced.
3. Visibility into database administrator changes: Stored procedures permissions seemed to sometimes “go missing” and system roles get regrouped without teams being properly notified.
4. Visibility into Internet Information Services changes: IIS was sporadically reset to different versions of the .NET framework. These changes pointed to iisadmin scripts run during deployments from other teams.

The environment configuration tests also provided easy access to the most important inventory of environment configuration settings. The most important configuration settings of Project X, Y, and Z were no longer sitting in obscure, hard-to-find documents. Change and configuration management procedures required efficient ways to access consolidated environment configuration information in order to quickly address project teams’ needs. Configurations were now easily accessible from the live and growing environment configuration testing suite.

During these six weeks, more flows were added to the functional testing suite of applications based on bugs reported in production, or due to new added functionality.

XML test cases had to be re-factored due to application changes. Refactoring activities took less than eight hours to complete and were simple and time effective.

## 6.5 Reflection - Gaining visibility and support to implement technical automation techniques

We chose a set of Manns and Rising's patterns [56] to assist in propagating this testing strategy to others in the organization. Mary Lynn Manns and Linda Rising provide a set of 48 patterns for introducing new ideas into organizations [56]. A pattern is defined as a solution for a recurring problem. These authors define a "powerless leader" as someone that: belongs to an organization, that has passion for a new idea, and would like to introduce that idea to his/her organization. These 48 patterns can help powerless leaders to drive change into an organization, breaking away myths such as:

- Change requires a specific plan. The authors suggest that a vision is needed instead to "test the waters", to introduce and steer change "step by step", allowing time for "reflection" and time to celebrate "small successes."
- Change just requires a good explanation of the idea; if people understand it they will follow it. However behavior change happens mostly by speaking to a person's feeling. New ideas have to be "in your space" as a reminder of how beneficial it would be if they were actually in place.
- Powerless leaders can lead change alone. Manns and Rising suggest that change requires that a powerless leader become or hire an "Evangelist". This evangelist has passion for the idea and will recruit "innovators" to "whisper in the general's ear" the ideas he or she is trying to implement. "Bridge builders" are individuals that are not directly involved in the change, but that have powers of influence over

others that can help. Once innovators recruit other supporters, a “group identity” will be born of those that believe in and support the idea being implemented. And always remember to “just say thanks” to those that assisted in implementing the new idea.

- Cynics and skeptics are negative and should be avoided. Manns and Rising suggest that powerless leaders should learn from cynics and skeptics. Instead of avoiding them, present the ideas to them and learn about their resistance first, so you can address them before the idea becomes public.

It is important to discuss why these patterns are only presented in this case study, since the primary goal of this thesis was to investigate the change brought in by the adoption of IID practices. These patterns apply to how a powerless leader can introduce change. The IID adoption was supported by upper management, but other than the RUP process engineer, we could not find an individual that took ownership for the process changes. In addition, the RUP process engineer was not available for interviews for our study. Thus, these patterns could not be studied in regards to the IID adoption strategy.

The author of this thesis took the role of the “evangelist” for the environment configuration tests. The team lead and testing lead were the “innovators” that helped to “whisper in the general’s ear” the stories of the test automation. With the formal approval given by the innovators, we followed the “just do it” pattern, implementing the testing strategies on lower environments, taking a “step by step” approach.

After seeing the benefits of the testing suites, we inquired about scheduling them to run against higher staging environments. The enterprise testing team was responsible for the health and well functioning of the staging environments. At first, they were resistant to give any special permissions to run these tests in higher staging environments. This enterprise team played the “cynics and skeptics” role.

Because NUnit was used as the test harness tool, the enterprise testing team assumed the tests were developer unit tests and so unfit to run in more secured environments. To clarify this misconception and to get their support going forward, we booked meetings with the enterprise team, inviting the management group as well (“bridge builders”), to present the test suite and automation strategy. The enterprise testing team lead and his senior tester were also included in the daily test-run e-mail notifications (idea “in your space”). After the presentation, managers and the enterprise testing team became interested to see the source code, and they were granted access. Soon after the demos, this enterprise team assigned a developer to validate the test suite in order to give the permission to run it in higher staging environments. First, they focused on Environment Configuration Testing.

#### 6.5.1 Technical impact of our testing strategy

As we continued to run the tests daily, the enterprise team assigned a dedicated resource to leverage the testing suite to accomplish the following goals:

- Consolidated reporting for multi-environment test execution for individual teams - a test summary for each environment per team should provide an overview of all tests run against an environment with the ability to get more details for each test. Our tests used XML files to specify the environment in which it should be run, as well as the test cases associated with that environment. The enterprise team created another layer of abstraction. A test is given three inputs: the staging environment in which it should run, the name of the test to be executed, and an XML file name. Figure 6.8 illustrates this execution.

The test agents are XML files that contain the list of test packages to run for each environment and include the fully qualified paths of the assembly files to be executed

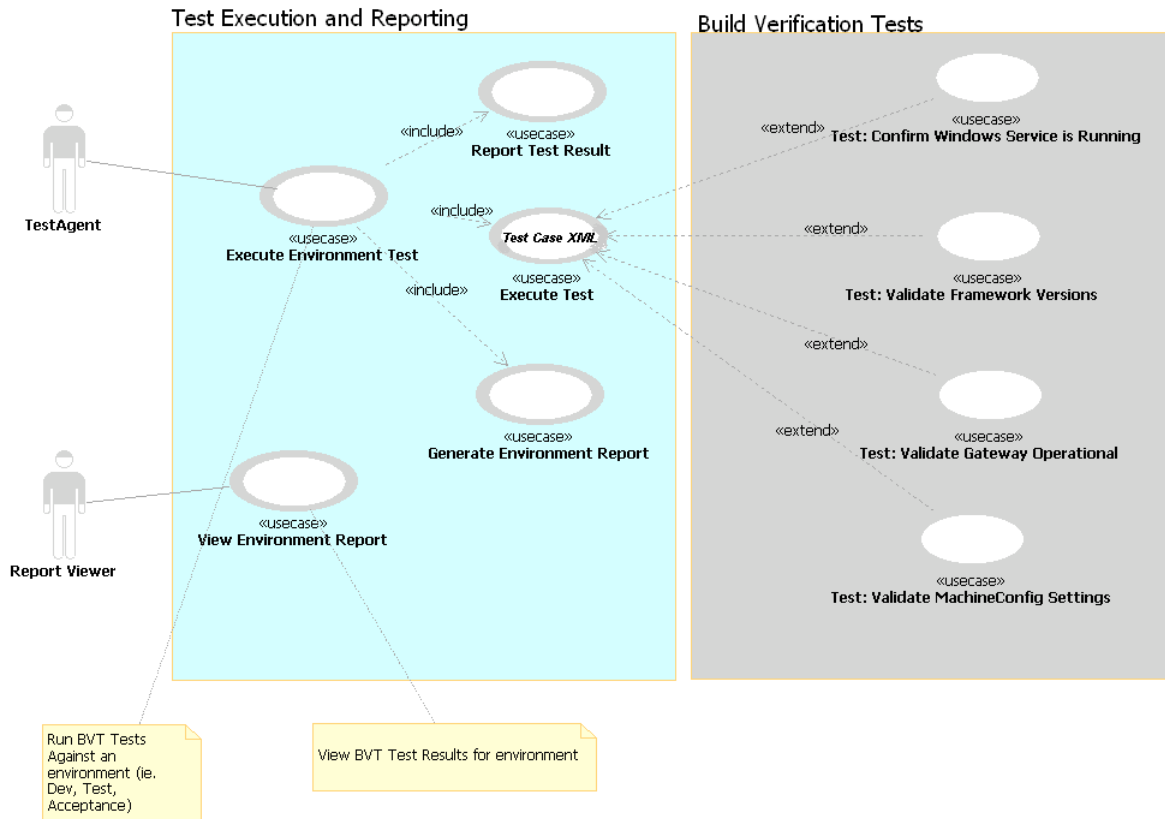


Figure 6.8: Environment Configuration Test - refactored Architecture

and the test specific configuration file, such as our initial XML configuration files. This set-up is what allows multiple projects to use this framework as they can identify test inputs that are specific to their application domain in XML files with no extra coding required.

- Encryption of Security Tokens - A testing infrastructure must allow the execution tool to securely access account information for a test for impersonation or other uses without compromising the account username and password. This was implemented to avoid staging environment security account information breaches.

Our environment configuration testing suite was refactored by the enterprise test team into a testing framework named “Build Verification Testing” (BVT). The BVT



successfully implemented all of our configuration test fixtures. The tool has also included support to run the back-end functional NUnit tests. It has been designed as a “one-stop shop” to view all automated tests running in the organization. These tests have been scheduled to run three times a day in all staging environments, including production, with the exception of functional tests that do not roll-back data. They can also be run on-demand by a group of users that have been granted special permissions. This includes the author of this thesis. We remembered to “just say thanks” for their newly-gained support. Figure 6.9 shows an example of the BVT reporting engine. We have blacked-out sensitive information (such as project and server names).

The screenshot shows a web-based reporting interface for BVT tests. At the top, there are search filters: 'Date to Start Search' (7/30/2009), 'Show:' (One Day), 'Program' (All), 'Agent' (All), 'Environment' (Test), and 'Show Failed Only' (False). A 'View Report' button is on the right. Below the filters is a document management pane on the left and a main table of test results. The table is titled 'Environment/Parent Group: Test' and 'Server/Child Group: [redacted]'. The suite is 'Inf-Test- [redacted]' and the tests are 'Infrastructure tests running against the Test environment on [redacted]'. The table contains 18 rows of test results, all with a 'Pass' status.

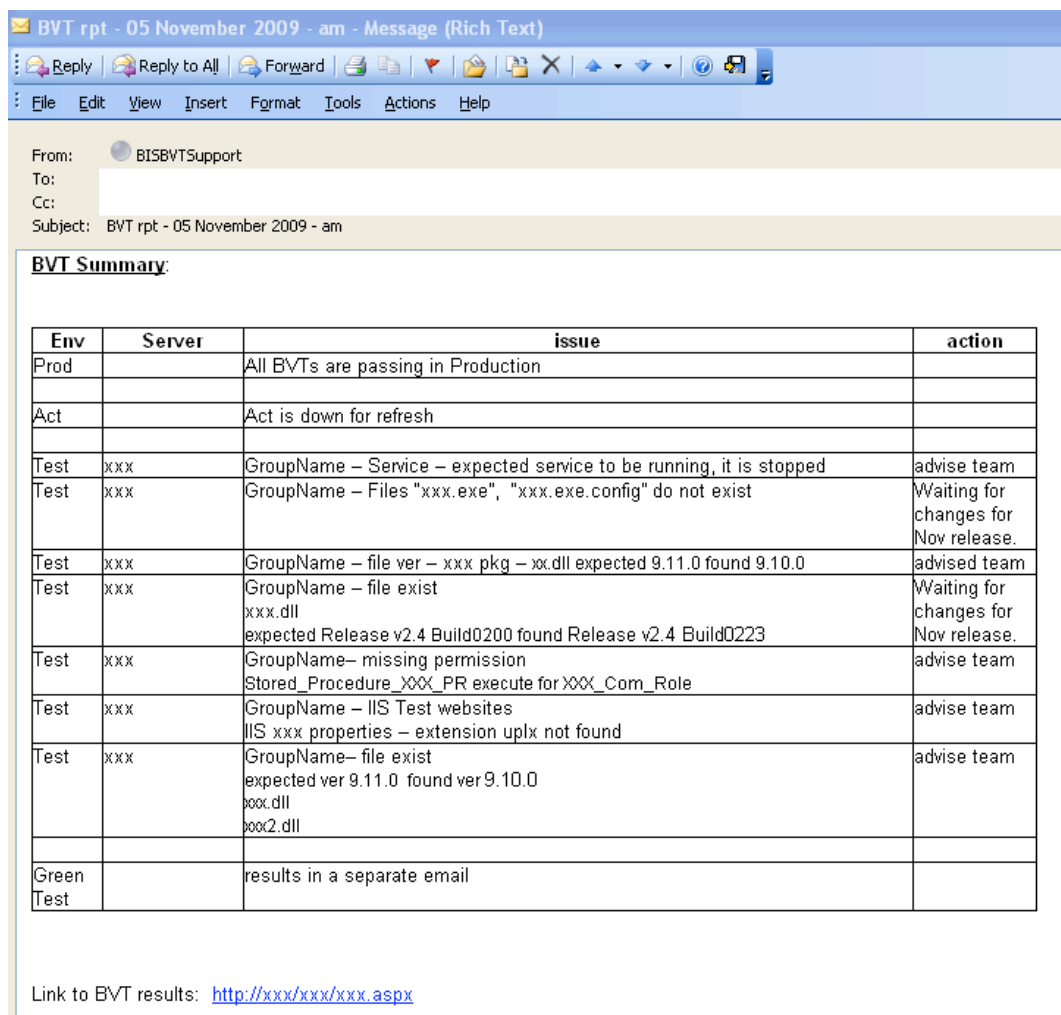
ID	Description	Date	Status
121444	Check Web App - Are Subsystems Online	7/30/2009 12:20:21 PM	Pass
121443	Verify Files Exist - WebStubs	7/30/2009 12:20:21 PM	Pass
121442	Verify IIS AppPool	7/30/2009 12:20:21 PM	Pass
121441	Verify Files Exist - SmartNav	7/30/2009 12:20:21 PM	Pass
121440	Test for XML Node Existence - machine.config	7/30/2009 12:20:21 PM	Pass
121439	Verify XML File is Valid - dlhost.exe.config	7/30/2009 12:20:21 PM	Pass
121438	Verify Registry Value	7/30/2009 12:20:21 PM	Pass
121437	Verify File Version - DM API	7/30/2009 12:20:21 PM	Pass
121436	Verify File Exists - [redacted]	7/30/2009 12:20:21 PM	Pass
121435	Check if DLL in COM - EGIS	7/30/2009 12:20:21 PM	Pass
121434	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121433	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121432	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121431	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121430	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121429	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121428	Check if DLL in COM - [redacted]	7/30/2009 12:20:21 PM	Pass
121427	Check [redacted] Framework Build	7/30/2009 12:20:21 PM	Pass
121426	Check Windows Services	7/30/2009 12:20:21 PM	Pass

Figure 6.9: Environment Configuration Test (BVT) - reporting example

## 6.5.2 Organizational impact of our testing strategy

Overall, we went from an organization that did not support test automation to an organization that was investing time and money into an infrastructure that consolidated environment configuration tests for reuse by multiple teams as well as test run results for functional tests.

All project managers, technical leads and testers receive three daily e-mails with a summarized status of the test's run as well as a link to the reporting engine - Figure 6.10.



The screenshot shows an email window titled "BVT rpt - 05 November 2009 - am - Message (Rich Text)". The email is from "BISBVTsupport" and has the subject "BVT rpt - 05 November 2009 - am". The main content is a "BVT Summary" table.

Env	Server	issue	action
Prod		All BVTs are passing in Production	
Act		Act is down for refresh	
Test	xxx	GroupName – Service – expected service to be running, it is stopped	advise team
Test	xxx	GroupName – Files "xxx.exe", "xxx.exe.config" do not exist	Waiting for changes for Nov release.
Test	xxx	GroupName – file ver – xxx pkg – xx.dll expected 9.11.0 found 9.10.0	advised team
Test	xxx	GroupName – file exist xxx.dll expected Release v2.4 Build0200 found Release v2.4 Build0223	Waiting for changes for Nov release.
Test	xxx	GroupName– missing permission Stored_Procedure_XXX_PR execute for XXX_Com_Role	advise team
Test	xxx	GroupName – IIS Test websites IIS xxx properties – extension uplx not found	advise team
Test	xxx	GroupName– file exist expected ver 9.11.0 found ver 9.10.0 xxx.dll xxx2.dll	advise team
Green Test		results in a separate email	

Link to BVT results: <http://xxx/xxx/xxx.aspx>

Figure 6.10: Environment Configuration Test (BVT) - notification example

The NUnit functional system regression test suite has also proved a very worthwhile

and efficient investment. Here is a quote from an e-mail sent from the manual testing team leader to the management group:

*“With the introduction of the NUnit tests for the [Project Z] system functionality, this has reduced one set of smoke testing time from 3 hours down to 5 minutes. For example, another team requested our program testing with regard to document management changes they made. Typically this requires regression testing of four different applications [that all execute the same functionality]. Normally this would take hours to perform a smoke test across all solutions. By running the NUnit tests this same regression test coverage is completed in approx. 2 hours. As a result, [our team] was able to effectively test urgent changes (in DEV/TST/ACT) with increased test coverage and reduced test time. This combination of NUnit and manual testing significantly increases the confidence in our suite of systems and our understanding of the cross impacts from other integrated systems that we rely on.”*

The testing team leader closed the e-mail by saying that the automated tests have had a very positive effect on the work of the manual testers.

The Build Verification Tests have become mandatory to all projects delivering code to any staging environment. A document called BVT Change Request has been created and all teams must specify their test case inputs and select which tests make sense for their suite of applications. Figure 6.11 shows this template document.

All BVT tests must pass in all environments and are now part of the release sign-off documentation. It is a pre-requisite for approval by management in order for a release to be deployed to production. Figure 6.12 shows an example of this document.

The environment configuration tests started a change in the organizational views towards test automation. This was accomplished by creating a simple but tangible test suite that ran in an automated fashion and displayed results daily to involved stakeholders. The support from project team leads and their powers of influence with the

BVT Change Request Template									
1									
2	<b>Project Team</b>	<b>Deployment environment</b>	<b>Deployment Date</b>		Email completed form to BISBVTSupport				
3		Test							
4		Acceptance			Date submitted:				
5		Production							
6									
7									
8	<b>Change Type</b>	<b>BVT changes</b>							
9		(Select from the pull down)		(put your info here)		(put your info here)		(put your info here)	
10	new	<b>File Comment</b>	<b>File Name</b>		<b>Path to File</b>		<b>Value</b>	---	
11	changed	<b>File Existence</b>	<b>File Name</b>		<b>Path to file</b>		---	---	
12	drop	<b>File Version</b>	<b>File Name</b>		<b>Path</b>		<b>Version</b>	---	
13		<b>File/Folder Permissions</b>	<b>name</b>		<b>Path</b>		<b>Account</b>	<b>Permissions</b>	
14		<b>in COM+</b>	<b>Interface Name</b>		<b>Path</b>		<b>Version</b>	<b>Identity</b>	
15		<b>Registry Value</b>	<b>Name</b>		<b>Path</b>		<b>Value</b>	---	
16		<b>Subsystem Test</b>	<b>Gateway URL</b>		<b>Subsystem name</b>		<b>Function</b>	<b>Transaction Type</b>	
17		<b>Verify IIS: Application Pool</b>	<b>AppPoolName</b>		<b>running/stopped</b>		<b>UserName</b>	<b>Server</b>	
18		<b>Verify IIS: Properties</b>	<b>AccessFlags</b>		<b>AuthFlags</b>		<b>UserName</b>	<b>serverBindings</b>	
19		<b>Web page online</b>	<b>web Name</b>		<b>URL</b>		---	---	
20		<b>Windows Service</b>	<b>Name</b>		<b>running/stopped</b>		<b>Start type</b>	<b>Registry Name</b>	
21		<b>XML file tests</b>	<b>File Name</b>		<b>path</b>		<b>Node/Attr Name</b>	<b>Value</b>	
22		<b>Account in Group</b>	<b>Account name</b>		<b>Group Name</b>		<b>Domain or Local</b>	<b>Server</b>	
23		<b>DB - Member of Role</b>	<b>database:</b>		<b>Member Name</b>		<b>Role name</b>	---	
24		<b>DB - Stored Procedure</b>	<b>database:</b>		<b>SP name</b>		<b>Role name</b>	---	
25		<b>DB - Table</b>	<b>database:</b>		<b>Table name</b>		<b>Role name</b>	<b>Permissions</b>	
26				example:					
27	new	XML file tests	File Name	web.config	path	xxxapp1\ve\$\apps\myproj (don't forget to tell us which server!)	Node/Attr Name	Env_Db	Value
28									
29									
30									

Figure 6.11: BVT Change Request

Enterprise Management Team helped the environment configuration tests to move from project-specific solutions to an enterprise wide solution.

A verbatim quote from an e-mail sent by the test lead states that:

*“Before the configuration BVT’s: One or more testing environments would be unstable each day causing delays in development and testing. During a period of 3 months, environment outages were tracked and over 50 hours of development and testing time were lost and business confidence in the environments/systems was very low. In addition, identifying and addressing the cause of the outage required 2 to 5 people to get involved to search for and address the issue. The primary root cause was very difficult to determine. After configuration BVT’s: BVT’s initially focused on monitoring and “flagging” environment configuration changes. The BVT’s were executed on a daily basis and on command. The impact of the BVT’s is significant. Problem root cause identification is immediately available and resolution activities are focused (1*

	A	B	C	D	E	F	G	H	I	J
1	Project Description									
2	Project Lead									
3	Planned Release Date									
4	Change Awareness #									
5										
6		Project Name -- RELEASE								
7										
8	Development and Test Management	Quality Criteria	Actuals			Agreed Upon Values (Environment)			Responsible Party	Supporting Evidence
9			Date	Date	Date	Test	Act	Prod		
10		Unit Testing Code Coverage							Project Team	
11		Integration Testing Code Coverage							Project Team	
12		Requirements Coverage							Project Team	
13		Defect Count (Critical/Major)							Project Team	
14		Peer-based Code Reviews (Incl at least 1 for Data Access)							Project Team	
15	Requirements Review (most							PMO (Requirements)		
16										
17	Deployment and Release Management									
18		Solution BVT							Project Team	
19		Training Plan							Project Team	
20		Bill of Materials							Project Team	
21		Deployment Plan							Project Team	
22		Release Notes							Project Team	
23		Communications Plan (include Automated Release package							Project Team	
24								Project Team		
25								Project Team		
26										
27	Operations and Support									
28		Operations Plan Manual (includes							Project Team, TSI,	
29		Disaster Recovery Plan (DRP)							Project Team	
30										
31										
32	Approvals and Signoffs									
33		DRP Signoff					N	N	Y	TSI
34										

Figure 6.12: Release sign-off document

- 2 people) and usually are corrected in minutes. It was identified that 97% of all instability problems were associated with environment configuration and database issues/changes (not system code/functionality). Test environment downtime has been reduced to 0 - 10 minutes per week. Most importantly, overall organization confidence in the test environments has significantly increased - to the point where development and test results are trusted and used for benchmarking purposes.”

As such, we have fulfilled our researcher-client agreement, by showing by example that test automation is indeed a worthwhile investment.

## 6.6 Implications for practice

From a test automaton adoption point of view, we provide both technical and organizational suggestions/lessons learned.

### 6.6.1 Technical lessons-learned

1. NUnit proved to be an effective, flexible, and easy-to-use testing harness for the execution of tests other than unit tests. It took approximately 60 hours to develop the environment configuration tests.
2. By running environment configuration tests on an on-going basis, the root causes of environment instability and configuration problem areas started to emerge.
3. The configuration necessary to run applications is no longer hidden in long documents in source control or in deployment scripts; it is visible and readily available in a suite of automated tests.
4. Using XML to mimic the data transfer objects proved to be an excellent strategy for this data-centric application. It is easy to create XML test inputs for new functionality added to the system, or to add new scenarios based on errors and bugs being found in production. As well, the tool created by one of the team's developer to generate XML files based on existing database-saved requests in the staging environments is also used to debug problems. In cases where the team had production issues, this tool is used to extract the data in the production environment. The same scenario is then used in the lower staging environment where it was easier to debug and log the application tests. This has enabled developers to find faults in a much more efficient manner. The important aspect here is to have a declarative representation of the test case data outside of the test code, be it XML or another declarative data storage tool.

5. The GUI scenario-based Functional Regression Tests unfortunately did not yield positive results. We share similar experiences to Berner *et al.* [17] in regards to the cost and inefficiency of GUI functional testing. According to these authors “extreme caution is needed if script-based automation of GUI based tests is attempted. These tests require more effort to create, and deliver a large number of false positives, are difficult to maintain, and are therefore executed less often than expected.”

The Rational Functional Tester scripts have been of little use to the team. IBM Rational Functional Tester (RFT) IDE is not particularly intuitive to use. In various occasions RFT tried to convert our project to Visual Studio 2005, losing the configuration for the applications under test. Issues with the playback scripts happened constantly due to custom JavaScript in the system under test and required manual code changes to the test playback code. Even after these code changes the scripts were unstable, throwing “ObjectNotFound” errors related to HTML DOM objects intermittently. The upgrade to Rational Functional Tester 7.0 required that the scripts be moved to Visual Studio 2005, which caused many migration issues. Since the scripts were being under-utilized, they were simply decommissioned.

#### 6.6.2 Contextual Organizational suggestions

1. Start by addressing your team’s biggest hurt area. Make your testing execution visible by automating it. Keep reusability in mind. If possible make it reusable by other teams so you can demonstrate more earned value.
2. Share your code and experiences with other teams that can benefit from it and provide insight into their problems, they will also help you gain visibility.
3. Expect resistance. The truth is that automated testing requires special permissions, so admit it, get it working on a local environment where you do have enough

security privileges, and plan to do some convincing for using it on higher staging environments. Do not take this resistance personally, nicely present the benefits of your tests to those being skeptical and they will see that it will actually help them do their job better too.

4. Keep maintainability in mind. The effort involved in maintaining a running test suite is one of the causes of why automated tests are abandoned by teams [17, 33]. Make it easy to add new test cases to your suite without having to change or redeploy code.
5. If you have a manual testing team, get them involved. There is a difference between testing and automating. Manual testers have the skill of testing, while you may just be a good developer that can automate stuff [33]. By involving these people you will get the most out of your time.
6. Manns and Rising's patterns provided invaluable techniques to diffuse these new ideas in a context where test automation was not given support.

### 6.6.3 Generic Lessons-learned

In companies with concurrent projects and legacy systems, not all teams are allowed to utilize test automation and automated deployment techniques in all staging environments. But they all still share these environments and need to validate server configurations to run their applications. Due to differences in delivery schedules, teams deploy builds to staging environments at different times (manually or through scripts). This leads to staging environment configurations being overwritten or lost and code from other teams starts to fail after being tested and deployed. The fine line between enterprise-wide and project-specify configuration management is grey. The introduction of a suite of automated environment configuration tests that verify the environment on demand, has



helped us identify many deployment dependencies and assist teams in resolving configuration issues in staging environments in a timely manner. It also abstracts environment configuration management into a live and evolving test suite that shows failure and is easy to maintain. The creation of a simple test strategy that used existing free test tools, with generic test drivers that use declarative descriptions as test case inputs, enables future re-use. This reuse escalates the use of this test strategy to an enterprise testing framework, proving the need and usefulness of the techniques we implemented.

Thus, in situations where test automation is not supported by upper management a team could invest a relatively small amount of time to:

- Investigate their biggest hurt area cause by the lack of test automation. If not hurt areas exist, then pick the area that could benefit the most from test automation.
- Create a simple testing strategy that uses existing free testware tools and create generic test drivers that use declarative test case inputs. This will allow future re-use.
- Automate the test execution using existing OS tools, like Microsoft Task Scheduler and expose the test execution results in an easy to access location, such as a simple website.

By demonstrating a live-working testing suite that is automatically running and producing simple results, a team may gain just enough traction to engage change. A team may find out that “no” means “I am not sure what test automation really is and what benefits it can really bring.”

To the best of our knowledge, we could not find related experiences on testing strategies that focus on verifying environment configuration. But, before testing that systems are running functionally, we need to ensure they are in a hospitable environment.

## 6.7 Limitations

In this case study we report on an action research project conducted to address specific issues with staging environment instability and lack of support for test automation. Care should be taken in drawing general conclusions based on our results. However, we believe that our findings can be valuable for organizations facing similar changes, especially when aggregated with other findings [64]. The environment configuration testing techniques presented here were geared towards systems deployed to Microsoft Windows servers. The back-end function testing techniques are only applicable to applications developed using object-oriented practices that expose interfaces. Accurately applying these techniques to other operating systems and other architectural designs would require more time and resources that we could afford during this Masters Degree.

## 6.8 Research goals addressed

This chapter represented an action research project with the goal of addressing two issues unresolved by the adoption of IID: staging environment instability and lack of support for test automation. These issues were identified during the interviews and questions&answers sessions conducted in case study 1 and case study 2.

This chapter described a test strategy that successfully introduced test automation solutions to legacy applications and alleviated configuration management issues. We accomplished this by delivering:

- A suite of back-end NUnit functional tests that used XML-driven data inputs leveraged the maturity of maintenance applications. This strategy was able to introduce a change in the perspective of the cost-value relationship of test automation for legacy applications.

- A suite of automated environment configuration tests that verify staging environments on demand. This test suite has helped this government agency identify many deployment dependencies, and assists teams to resolve configuration issues in staging environments in a timely manner. It also abstracts environment configuration management into a live and evolving test suite that shows failure and it is easily maintainable.

# Chapter 7

## Conclusion

In this thesis, we presented three case studies that were conducted to understand and influence the adoption of an IID approach by a group of IT projects in a large bureaucratic government agency. The improvement efforts were motivated by several concerns identified by the business clients of those projects: poor software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays.

In particular, we described how the changes affected several existing projects (case study 1 - Chapter 4). We explored the practices in the context of a new project (case study 2 - Chapter 5). In both cases, the effects of these changes are considered quantitatively and qualitatively. We presented an action research project with the goal of addressing two issues unresolved by the adoption of IID: staging environment instability and lack of support for test automation (case study 3 - Chapter 6).

In this final chapter, we revisit the research goals as well as provide a summary of the key contributions. We present some continuing changes at the organization under study and conclude with some discussion of future work.

### 7.1 Research Goals

The primary research goal was to investigate how IID practices affected the business clients' concerns related to: poor software quality and stability, insufficient testing timelines, poor bug-fixing responsiveness, and delivery delays. From that perspective, the IID adoption goals studied were presented in the Goal Question Metric approach in Chapter 1, and are summarized in Table 7.1.

Goal 1	
Purpose	Improve
Issue	Poor and unstable
Object (process)	quality of releases
Viewpoint	IT Management and Business clients
Goal 2	
Purpose	Improve
Issue	Rushed or insufficient
Object (process)	testing timelines
Viewpoint	Business clients
Goal 3	
Purpose	Improve
Issue	Long wait and red-tape
Object (process)	bug-fixing requests
Viewpoint	Business clients
Goal 4	
Purpose	Decrease
Issue	Delays
Object (process)	Release delivery
Viewpoint	IT Management and Business clients

Table 7.1: Our GQM approach

These goals were translated into specific questions and metrics which were applied to a suite of existing projects that transitioned from waterfall to IID, and to a project that followed IID since its inception. In addressing these research goals, this work provides two key contributions: (1) an addition to the body of knowledge about the effectiveness of IID in environments with higher degrees of formality, and (2) a description of the experience the organization had in making those changes.

Comparisons between the projects studied revealed that a common set of issues were still unaddressed by the adoption IID practices due to the existing contextual constraints: staging environment instability and lack of support for test automation. The secondary research focus was to address the two issues stated above by investigating the following research questions:

- *What artifacts or processes can alleviate issues with configuration management of staging environment?* In companies with concurrent projects and legacy systems,

not all teams are allowed to utilize test automation and automated deployment techniques in all staging environments. But, they all still share these environments and need to validate server configurations to run their applications, which is the case of the projects under study in this thesis. Due to differences in delivery schedules, teams deploy builds to staging environments at different times (manually or through scripts), and this leads to staging environment configurations being overwritten or lost. As well, code from other teams starts to fail after being tested and deployed.

- *What artifacts or processes can support projects, especially legacy ones, in gaining support to invest time and money in test automation?* It is common for organizations to skip automated testing due to budget or time constraints [17], as they are not initially as important to the stakeholders as the implementation of extra features. While current literature provides many experiences and techniques to introduce test automation into software during construction, software engineers must often work with legacy applications in an adaptive maintenance mode that lacks automated tests.

In addressing these two research goals, this work provides another contribution: (3) an explanation of a test strategy that successfully introduced test automation solutions to legacy applications and configuration management issues.

Each of these three key contributions is briefly discussed in the following section.

## 7.2 Key Contributions

In addressing our research goals, this work has made three key contributions:

### 7.2.1 An addition to the body of knowledge about the effectiveness of IID in environments with higher degrees of formality

In more specific terms, we found that IID paired with investments in formal test teams and refactoring allowed existing projects to:

- Improve the software quality and stability of three quality-challenged, complex projects.
- Provide five areas of improvements to business clients: reestablishment of business involvement, better distribution of acceptance testing effort, introduction of a formal testing team, less pushback on necessary changes, and improved communication and management of expectations.
- Provide only limited improvements to bug-fixing responsiveness because of the existing backlog of bug reports.

We found that IID practices allowed a new project (three complex systems) to:

- Provide support for managerial decisions that resulted in this project being the first project in six years to deliver results both on-time and on-budget.
- Avoid some of the quality and stability issues experienced by former projects, such as the projects studied under case study 1.
- Provide significantly better bug-fixing responsiveness than projects that migrated to IID.

In both case studies, IID practices allowed the detection of bugs significantly earlier in the development cycle.

7.2.2 A description of the experience the organization had in making process changes  
Transitioning to an IID approach presented some important challenges and lessons-learned, such as:

- During the transitional stages software quality and stability actually got worse, as experienced by others [35, 44, 83].
- The government agency under study had large scale system changes happening at the same time as the methodology transition. The team members indicated that there was simply “too much going on all at once.” This made it difficult to differentiate problems due to the IID introduction or due to other technical or contextual issues.
- The initial IID adoption was very much tool-centric. The project teams felt that a process-centric approach would have resulted in quicker improvements.
- Business clients felt that they could have benefited from formal training in regards to what IID means and what it entails from their point of view.
- Due to existing quality issues, it took time for existing projects that migrated to IID to see real improvements to software quality and stability. These existing projects did not provide the same bug-fixing responsiveness as the project that followed IID since inception, as existing bugs needed to compete against new ones in the priority queue to get fixed. It is important to be realistic about the expectations surrounding the IID adoption in regards to solving existing issues. Although it took time, IID proved to be a very worthwhile investment.
- The shift to IID presented some paradigm shifts to business clients. During early iteration end demos the business clients tended to focus more on missing or “to be implemented” features than on the actual features presented.



- The business clients of the first fully IID-developed system delayed hands-on acceptance testing until bigger portions of the systems were completed. These clients were used to receiving an end-to-end system to test in the former Waterfall process. The business clients did not want to waste time mimicking data to test an atomic part of the application. They felt that once the mock part was ready it would need to be tested again. This delayed the discovery of some importance issues.
- During the first weeks after daily stand-up meetings were introduced. These meetings took too long and were considered cumbersome by the development team.

### 7.2.3 A test strategy that successfully introduced test automation solutions to legacy applications and configuration management issues

The third major contribution is a test strategy that successfully introduced test automation solutions to legacy applications and alleviated configuration management issues. We accomplished this by delivering:

- A suite of back-end NUnit functional tests that used XML-driven data inputs leveraged the maturity of maintenance applications. This strategy was able to introduce a change in the perspective of the cost-value relationship of test automation for legacy applications.
- A suite of automated environment configuration tests that verify staging environments on demand. This test suite has helped this government agency to identify many deployment dependencies and assist teams resolve configuration issues in staging environments in a timely manner. It also abstracted environment configuration management into a live and evolving test suite that is easily maintainable.

## 7.3 Future Work

The work presented in this thesis mainly focused on quantitative and qualitative data about how IID practices affected several concerns identified by the business clients of the projects we studied. The rich amount of data gathered leaves considerable room for further research. Consequently, possible future directions this research could take include performing root cause analysis of the bugs from case study 1 and case study 2; an evaluation of which RUP work products were deemed useful by the project teams; and an exploration of the longer term affects of the IID adoption in this organization.

### 7.3.1 Bug root cause analysis

Our work indicates that IID practices were able to increase the software quality and stability of the releases being delivered by a suite of existing projects. It also showed that IID practices allowed the detection of bugs earlier in the development cycle. But our work did not explore if IID practices changed, improved, or worsened some of the bugs' root causes. For example, have IID practices diminished the amount of bugs caused from wrong requirements?

### 7.3.2 An evaluation of RUP work products

One of the most common criticisms of the Rational Unified Process is the amount of documentation it requires. On the other hand, many believe that Agile methods require no documentation. In case study 2, the risk list document was a key work product in enabling stakeholders to make key decisions, but little was mentioned about the other work products. An interesting study could analyze how many times work products are accessed in source control and how many times they are changed in the lifecycle of a project.

### 7.3.3 Longer term effects of the IID adoption

According to Pettigrew, change needs time to “reveal its untidiness” [67]. The research conducted in this thesis was concluded approximately one-and-a-half years after the projects from case study 2 moved into maintenance mode. Other concepts and practices were introduced in a bottom-up fashion by leveraging some of the improvements provided by the IID adoption, while some existing IID practices faced cultural challenges. Consequently possible future directions this research could take include further analysis and investigation of these concepts and challenges.

New practices and concepts adopted after the IID adoption include:

- Test Driven Development (TDD),
- Continuous Integration,
- Daily stand-up meetings that also include the business clients, and, whenever possible, project sponsors,
- User stories (in replacement of use cases).

Some of the IID practices that were facing cultural challenges include:

- Iteration length.

The organization held the six-week iteration cycles for production releases. On the other hand, projects have internal iteration cycles (that lead to non-production releases) that range from two to four week iterations (which may require a two week code freeze when it is ready for production release). But, in April of 2009 upper management decided to move the production release schedule from 6 weeks to 10 weeks. Many of the legacy applications had problems migrating to the faster release cycles introduced by the IID adoption.

- The role of the middle manager.

Middle managers faced challenges in regards to the adoption of new techniques. It required skillful leveraging of known improvements coming from the RUP adoption. The hierarchial structure still operates very much in a command and control fashion.

# Appendix A

## Appendix

### A.1 Pettigrew's Contextual Framework

According to Andrew Pettigrew:

*“Episodic views of change not only treat innovations as if they had a clear beginning and a clear end but also, where they limit themselves to a snapshot time-series data, fail to provide data on the mechanisms and processes through which changes are created. Studies of transformation are, therefore, often preoccupied with the intricacies of narrow changes rather than the holistic and dynamic analysis of changing.”*

Pettigrew's states that *“causation is neither linear nor singular.”* To address this, his framework suggests that to conceptualize changes we need linkages between the content of the change, its context and its process. Pettigrew defines:

**Context** as the antecedent conditions of change, the internal structure, cultural, and political context within which transformation occurs, as well as broad features of the outer context of the firm from which much of the legitimacy for change is derived. This encapsulates the “why” of change [67].

**Process** as a sequence of individual and collective events, actions, and activities unfolding over time in context. This encapsulates the “how” of change [68].

**Content** as the particular areas of transformation under examination. This encapsulates the “what” of change [67].

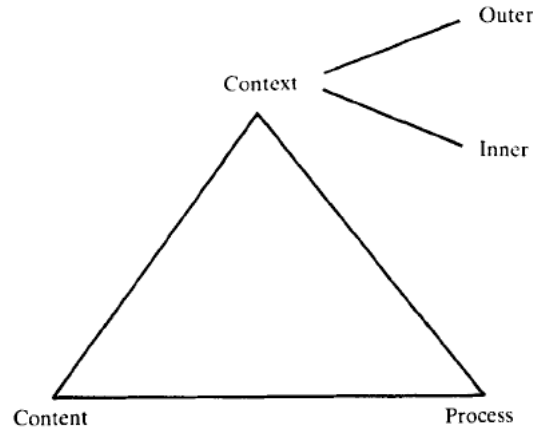


Figure A.1: Andrew Pettigrew's Contextual Analysis

Figure A.1 illustrates the relationship between these three variables. The process of change refers to the actions, reactions, and interactions from the various interested parties as they seek to move an organization from its present to its future state. This framework focuses on actors and systems in motion. The process is both constrained by context and shapes context, either by preserving or altering it. Pettigrew suggests that process research is best characterised in terms of cycles of induction (collecting generic data and learning from emerging themes in the data) and deduction (collecting data to answer a specific hypothesis)[68].

## A.2 Canonical Action Research

Davidson *et al.*'s principles of Canonical Action Research (CAR) [24] were implemented and will be presented in chapter 6. These principles attempt to marry rigor and relevance of action research conducted in system-related research, “*although they can be hard to achieve in a single CAR project.*” The CAR process is iterative and collaborative. It focuses on both organizational development and the generation of knowledge. The CAR model has five overarching principles:

- Researcher-Client Agreement (RCA).

This agreement includes:

- High-level descriptions of the research goals,
- Roles and responsibilities of the researcher and client,
- Ethical and anonymity requirements,
- A list of expected evaluation reports.

Neither the researcher nor the client should take over the CAR project. Clients and researchers should work together in *“roles that are culturally appropriate given the particular circumstances of the problem context”* [24]. A researcher-client agreement specifies both the commitment of the researcher to improve some particular aspect of the organizational situation and the commitment of the organization to share details and lessons-learned about the CAR project. The RCA should also specify project objectives, evaluation measures, data collection, and analysis methods to be used. These “success measurements” can and should be revisited and refocused as necessary. A RCA was signed between us and the industrial partner in August of 2007 (more details presented in chapter 6).

- Cyclical Process Model (CPM) with five distinct phases: diagnosis, action planning, intervention (action taking), evaluation, and reflection.

The sequential fashion of the five distinct phases enforces a level of systematic rigor, *“but our experience suggests that some iteration between stages may be needed. For example, supplementary planning must be necessary if an intervention cannot be completed as intended”* [24]. Thus, variations of the phases are allowed but should be explained in the context of the research.

During the diagnosis phase, the researcher is responsible for conducting an independent evaluation of the problems under research and to update the RCA accordingly if objectives need to be changed. Action plans should derive from the diagnosis of the problem in-hand. The intervention (action taking) should ensure that enough data is collected for future analysis and evaluation. During evaluation, the results from the intervention need to be matched against the original research objectives. The final reflection phase decides if more process cycles are needed, and defines if the project has fulfilled the exit criteria: either by reaching research goals, or by some other reasonable justification.

- Theory

The review of related literature helps the researcher to narrow the research goals and to focus data collection activities. The research goals must be of interest to the client organization but also to the research community. CAR projects may start as *“theory-free action learning”* but during diagnosis a grounded theory may emerge. If it does not, then explicit theorizing is required and the planned intervention and evaluation phases must implement the chosen theoretical framework.

- Change through Action.

This principle reflects the *“indivisibility of action and change, with intervention seeking to produce change”* [24]. In order to assess if change happened, it is necessary to understand and document the current organizational situation - the “as-is” state. The intervention must be appropriate to address the problem in-hand to move the organization to a “to-be” state. Similar to the view of Pettigrew [67] Davidson *et al.* cite the work of Hult & Lennung that state: *“in action research, a deliberate attempt is made not to divorce phenomena from the environment which gives them meaning The researcher using action research will be studying intercon-*



*nections, interdependencies and the dynamics of a total functioning system rather than isolated factors” [24]. The timing and nature of interventions should be documented for use during evaluation.*

- Learning through Reflection

This principle addresses the call for reports with implications for practice and further research. Davidson *et al.* suggest that progress reports, final research activities, and outcomes must be reported clearly to the client. Lessons-learned during the CAR project should be documented as soon as they are realized to avoid “retrospective bias.” Researchers should specify further actions that could improve the project outcomes. Researchers should present both practical lessons-learned as well as implications for the research community.

### A.3 Case study 1 and 2 - Examples of Normality Test

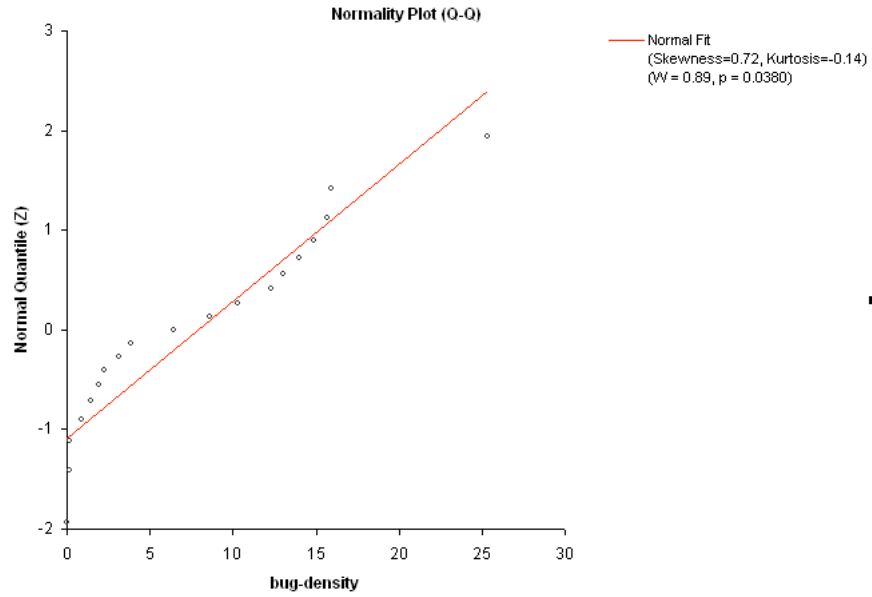


Figure A.2: Case study 1 - Normality Plot (Q-Q)

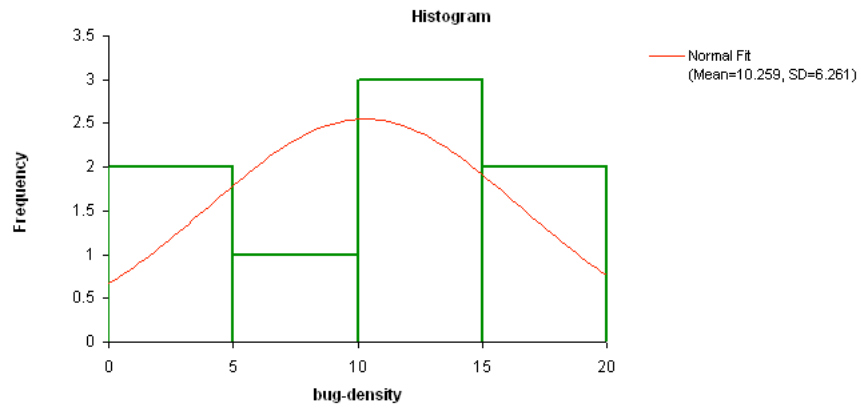


Figure A.3: Case study 1 - Pre-RUP histogram

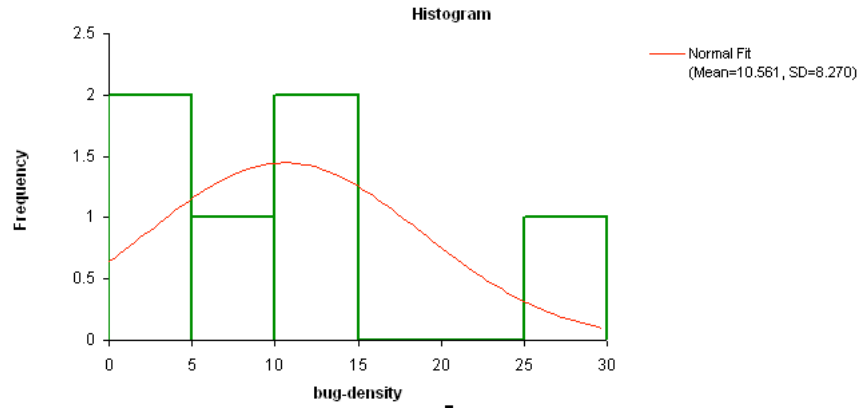


Figure A.4: Case study 1 - Transition histogram

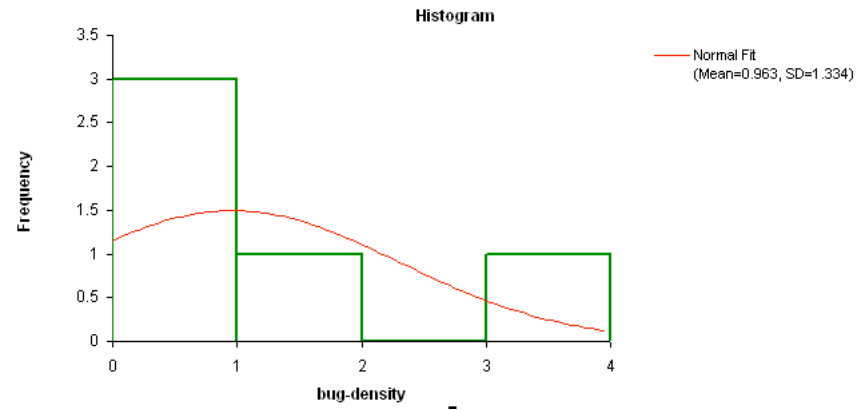


Figure A.5: Case study 1 - Partial RUP histogram

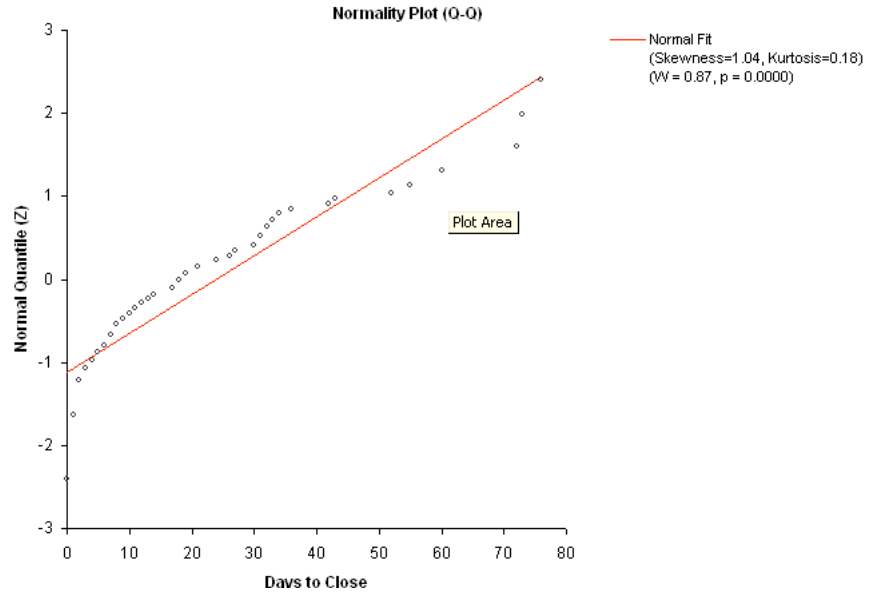


Figure A.6: Case study 2 - Normality Plot (Q-Q) for High Attention category

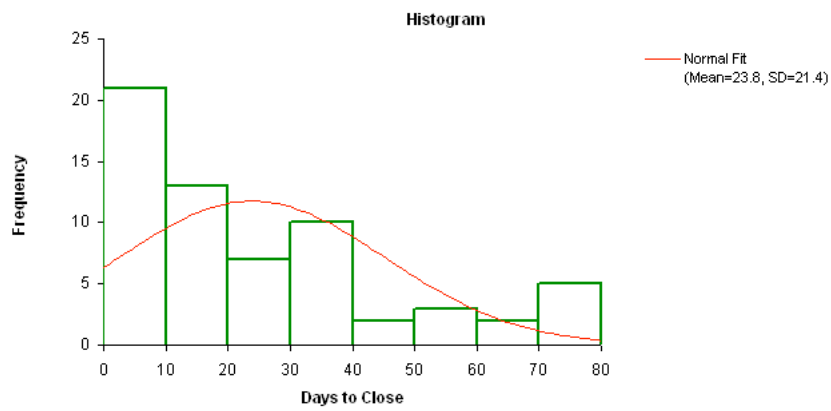


Figure A.7: Case study 2 - High Attention category histogram

## A.4 Ethics Approval

In this appendix, I present a copy of the Ethics Approval for the case studies conducted.

## A.5 Co-Author Permissions

In this appendix, I include permissions from my collaborators to use co-authored work from our papers in my thesis.

## Bibliography

- [1] Ibm rational functional tester. On-line: <http://www-306.ibm.com/software/awdtools/tester/functional> Downloaded: October 8, 2008.
- [2] The question man. On-line: [http://www.ajr.org/article\\_printable.asp?id=676](http://www.ajr.org/article_printable.asp?id=676) Downloaded: July 31, 2009.
- [3] What to avoid. On-line: <http://www.ajr.org/article.asp?id=678> Downloaded: July 31, 2009.
- [4] What to do. On-line: <http://www.ajr.org/article.asp?id=677> Downloaded: July 31, 2009.
- [5] Ieee, software engineering book of knowledge, 2007. Online: <http://www.swebok.org/index.html> Downloaded: May 1st, 2009 23:00pm.
- [6] Ibm rup white papers. 2009. On-line: <http://www.ibm.com/developerworks/search/searchResult> Downloaded: September 23, 2009.
- [7] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods. review and analysis. In *Espoo 2002 - VTT Publications*, volume 478, page 107, 2002.
- [8] S. Ambler. Agile adoption rate survey. On-line: [www.ambysoft.com/surveys/agileMarch2007.html](http://www.ambysoft.com/surveys/agileMarch2007.html). Downloaded: August 28, 2009.
- [9] S. Ambler. Agile and the rup. On-line: <http://www.ambysoft.com/unifiedprocess/>. Downloaded: August 28, 2009.

- [10] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, 2002.
- [11] IEEE Standard Association. Ieee std 610.12-1990 ieee standard glossary of software engineering terminology -description, 2004. Online: <http://standards.ieee.org> Downloaded: May 1st, 2009 23:00pm.
- [12] J. Barnes. *Implementing the IBM rational unified process and solutions: a guide to improving your software development capability and maturity*. IBM Press, 2007.
- [13] Paul A. Beavers. Managing a large "agile" software engineering organization. *AG-ILE 2007*, pages 296–303, 13-17 Aug. 2007.
- [14] Kent Beck and Cynthia Andres. *Extreme Programming Explained : Embrace Change (2nd Edition)*. Addison-Wesley Professional, November 2004.
- [15] M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. Online: <http://agilemanifesto.org> Downloaded: May 1st, 2009 23:00pm.
- [16] Hilary Berger. Agile development in a bureaucratic arena—a case study experience. *International Journal of Information Management*, 27(6):386–396, 2007.
- [17] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 571–579, New York, NY, USA, 2005. ACM.
- [18] Joseph A. Blotner. Agile techniques to avoid firefighting at a start-up. In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, pages 1–ff, New York, NY, USA, 2002.



ACM.

- [19] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.
- [20] R. Chillarege. Software testing best practices. ibm research. 1999. Online: <http://www.chillarege.com/authwork/TestingBestPractice.pdf> Downloaded: May 2nd, 2009 1:00AM.
- [21] G. Cloke. Get your agile freak on! agile adoption at yahoo! music. In *AGILE 2007*, pages 240–248, Aug. 2007.
- [22] Alistair Cockburn. Incremental versus iterative development. On-line: [http://alistair.cockburn.us/index.php/Incremental\\_versus\\_iterative\\_development](http://alistair.cockburn.us/index.php/Incremental_versus_iterative_development). Downloaded: July 31, 2009.
- [23] Mike Cohn. Patterns of agile adoption. On-line: <http://www.agilejournal.com/content/view/734/111/> Downloaded: May 4th, 2009.
- [24] R. M. Davison, M. G. Martinsons, and N Kock. Principles of canonical action research. *Information Systems Journal*, 14(1):65–89, 2004.
- [25] Doug DeCarlo. *eXtreme Project Management: Using Leadership, Principles, and Tools to Deliver Value in the Face of Volatility*. Jossey-Bass A Wiley Imprint, 2004.
- [26] T. Dingsoyr, T. Dyba, and P. Abrahamsson. A preliminary roadmap for empirical research on agile software development. In *Agile, 2008. AGILE '08. Conference*, pages 83–94, Aug. 2008.
- [27] Elfriede Dustin. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Professional, 2002.

- [28] Jutta Eckstein. *Agile Software Development in the Large - Diving Into the Deep*. Dorset House, New York, 1983.
- [29] Amr Elssamadisy. *Patterns of Agile Practice Adoption*. Lulu.com - InfoQ, 2007.
- [30] Hakan Erdogmus. Agile's coming of age ... or not. *Software, IEEE*, 24(6):2–5, Nov.-Dec. 2007.
- [31] John Favaro. Managing requirements for business value. *IEEE Software*, 19(2):15–17, 2002.
- [32] Mohamed E. Fayad. Software development process: a necessary evil. *Commun. ACM*, 40(9):101–103, 1997.
- [33] M. Fewster and D. Graham. *Software Test Automation*. ACM Press, New York, 1999.
- [34] Helle Damborg Frederiksen and Lars Mathiassen. A contextual approach to improving software metric practices. *IEEE Transactions on Engineering Management*, 55(4):602–616, 2008.
- [35] Chris Fry and Steve Greene. Large scale agile transformation in an on-demand world. *AGILE 2007*, pages 136–142, 13-17 Aug. 2007.
- [36] Robert L. Glass. The standish report: does it really describe a software crisis? *Commun. ACM*, 49(8):15–16, 2006.
- [37] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable Software*, pages 493–410, New York, NY, USA, 1975. ACM.

- [38] H. Grewal and F. Maurer. Scaling agile methodologies for developing a production accounting system for the oil & gas industry. In *AGILE 2007*, pages 309–315, Aug. 2007.
- [39] Standish Group. The chaos report (1994). January 2006. Online: [http://www.standishgroup.com/sample\\_research/chaos\\_1994\\_1.php](http://www.standishgroup.com/sample_research/chaos_1994_1.php) Downloaded: January 21, 2006 18:00pm.
- [40] F. Hartman. *Dont park your brain outside*. Project Management Institute, 2000.
- [41] 2006 Heather Scofield Published on September 15. Alberta’s boom starts to rival china’s; province’s growth seen as sustainable. On-line: <http://www.uofaweb.ualberta.ca/chinainstitute/nav03.cfm?nav03=50480&nav02=43112&nav0> Downloaded: August 28, 2009.
- [42] Michael Hirsch. Making rup agile. In *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, pages 1–ff, New York, NY, USA, 2002. ACM Press.
- [43] Ivar Jacobson, Grady Booch, and James E. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.
- [44] Rich Jochems and Shane Rodgers. The rollercoaster of required agile transition. In *AGILE 2007*, pages 229–233, 2007.
- [45] Magne Jrgensen and Kjetil Molkken-stvold. How large are software cost overruns? a review of the 1994 chaos report. *Information and Software Technology*, 48(4):297 – 301, 2006.
- [46] Osamu Kobayashi, Mitsuyoshi Kawabata, Makoto Sakai, and Eddy Parkinson. Analysis of the interaction between practices for introducing xp effectively. In *ICSE '06*:

- Proceeding of the 28th international conference on Software engineering*, pages 544–550, New York, NY, USA, 2006. ACM.
- [47] Adam Kolawa. Wrox programmer to programmer: Regression testing. On-line: <http://www.wrox.com/WileyCDA/Section/id-291252.html> Downloaded: February 7, 2008.
- [48] Venkatesh Krishnamurthy. Benefits of a top-down agile adoption strategy. On-line: <http://www.agilejournal.com/content/view/407/33/> Downloaded: May 4th, 2009.
- [49] Per Kroll and Bruce MacIsaac. *Agility and Discipline Made Easy: Practices from OpenUP and RUP (Addison-Wesley Object Technology (Paperback))*. Addison-Wesley Professional, 2006.
- [50] Philippe Kruchten. Situated agility. On-line: <http://www.lero.ie/download.aspx?f=Kruchten+080614+Situated+Agility+XP2008.pdf>. Downloaded: August 28, 2009.
- [51] Philippe Kruchten. Voyage in the agile memplex. *Queue*, 5(5):38–44, 2007.
- [52] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003.
- [53] Craig Larman. *Agile & Iterative Development, A Manager's Guide*. Addison Wesley, 2004.
- [54] Jeanne Lewis and Kevin Neher. Over the waterfall in a barrel - msit adventures in scrum. *AGILE 2007*, pages 389–394, 13-17 Aug. 2007.
- [55] 2003 M. L. Perla Published on July 1. Value proposition. On-line: <http://www.marketingprofs.com/3/perla8.asp>. Downloaded: August 28, 2009.

- [56] Mary Lynn Manns and Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley Professional, 2004.
- [57] Robert Martin. *UML for Java(TM) Programmers*. Prentice Hall, 2003.
- [58] Frank Maurer and Grigori Melnik. Agile methods: Crossing the chasm. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 176–177, Washington, DC, USA, 2007. IEEE Computer Society.
- [59] Steve McConnell. The 10 most powerful ideas in software engineering. On-line: <http://www.cs.uoregon.edu/events/icse2009/postconf/>. Downloaded: October 30, 2009.
- [60] Steve McConnell. *Code Complete, 2nd Edition*. Microsoft Press, 2004.
- [61] Parastoo Mohagheghi, Reidar Conradi, and Jon Arvid Borretzen. Revisiting the problem of using problem reports for quality assessment. In *WoSQ '06: Proceedings of the 2006 international workshop on Software quality*, pages 45–50, New York, NY, USA, 2006. ACM.
- [62] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.
- [63] Geoffrey A. Moore. *Crossing the Chasm, Marketing and Selling High-Tech Products to Mainstream Customer (revised edition)*. HarperCollins Publishers, New York, 1999.

- [64] Margaret Myers. Qualitative research and the generalizability question: Standing firm with proteus. *The Qualitative Report*, 4:30 paragraphs, 2000.
- [65] J. Packlick. The agile maturity map a goal oriented approach to agile improvement. In *AGILE 2007*, pages 266–271, Aug. 2007.
- [66] White paper. Rational unified process - best practices for software development teams. On-line: [http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpr](http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpr). Downloaded: August 28, 2009.
- [67] Andrew M. Pettigrew. Context and action in the transformation of the firm. *Journal of Management Studies*, 24(6):649–670, 1987.
- [68] Andrew M. Pettigrew. What is a processual analysis? *Scandinavian Journal of Management*, 13(4):337 – 348, 1997. Reflections on Conducting Processual Research on Management and Organizations.
- [69] Caryna Pinheiro, Frank Maurer, and Jonathan Sillito. Adopting iterative development: The perceived business value. In *XP*, pages 185–189, 2008.
- [70] Caryna Pinheiro, Frank Maurer, and Jonathan Sillito. Moving towards agility in a bureaucratic environment up as a bridge between waterfall and agile processes. *AGILE 2008*, page Workshop presentation, 2008.
- [71] Caryna Pinheiro, Frank Maurer, and Jonathan Sillito. Improving quality, one process change at a time. In *31st International Conference on Software Engineering (ICSE 2009)*, volume 31 of *31st International Conference on Software Engineering - ICSE Companion 2009*, pages 81–90. IEEE, 2009.

- [72] Charlie Poole, Jamie Cansdale, and Gary Feldman. Nunit. On-line: <http://www.nunit.org/index.php> Downloaded: October 10, 2008.
- [73] Mary Poppendieck and Tom Poppendieck. *Lean Software Development - an Agile Toolkit*. Addison Wesley, 2006.
- [74] Thomas Pyzdek. Non-normal distributions in the real world. On-line: <http://www.qualitydigest.com/dec99/html/nonnormal.html>. Downloaded: December 8, 2008.
- [75] Maryam Razari and Lee Iverson. A grounded theory of information sharing behavior in a personal learning space. In *CSCW'06 - Computer Supported Cooperative Work*, pages 459–468, New York, NY, USA, 2006. ACM.
- [76] Everett Rogers. *Diffusion of innovations, 3/e*. The Free Press, New York, NY, 1983.
- [77] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
- [78] H. J. Rubin and I. S. Rubin. *Qualitative Interviewing: The Art of Hearing Data (2nd Edition)*. SAGE Publications, 2005.
- [79] Aaron Ruhnow. Consciously evolving an agile team. *AGILE 2007*, pages 130–135, 13-17 Aug. 2007.
- [80] Koch. A. S. *Agile Software Development - Evaluating the Methods for Your Organization*. Artech House, Boston, 2005.
- [81] Outi Salo. Improving software process in agile software development projects: Results from two xp case studies. In *EUROMICRO '04: Proceedings of the 30th EU-*

- ROMICRO Conference (EUROMICRO'04)*, pages 310–317, Washington, DC, USA, 2004. IEEE Computer Society.
- [82] Célio Santana, Cristine Gusmão, Liana Soares, Caryna Pinheiro, Teresa M. Maciel, Alexandre Marcos Lins de Vasconcelos, and Ana Cristina Rouiller. Agile software development and cmmi: What we do not know about dancing with elephants. In *XP*, pages 124–129, 2009.
- [83] Thomas R. Seffernick. Enabling agile in a large organization our journey down the yellow brick road. *AGILE 2007*, pages 200–206, 13-17 Aug. 2007.
- [84] Chris Spagnuolo. Agile adoption in the gis industry. On-line: [http://www.directionsmag.com/article.php?article\\_id=2711](http://www.directionsmag.com/article.php?article_id=2711). Downloaded: August 29, 2009.
- [85] Megan U. From waterfall to agile - how does a qa team transition? *AGILE 2007*, pages 291–295, 13-17 Aug. 2007.
- [86] H. Rombach V. Basili, G. Caldiera. The goal question metric approach. *Encyclopedia of Software Engineering*, 2:528–532, 1994.
- [87] James F. Walsh. Preliminary defect data from the iterative development of a large c++ program (experience report). In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 178–183, New York, NY, USA, 1992. ACM.
- [88] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Softw.*, 17(1):70–79, 2000.
- [89] Zachary Wong. *Human factors in project management*. John Wiley & Sons, 2007.



- [90] Wolfgang Zuser, Stefan Heil, and Thomas Grechenig. Software quality development and assurance in rup, msf and xp: a comparative study. In *3-WoSQ: Proceedings of the third workshop on Software quality*, pages 1–6, New York, NY, USA, 2005. ACM.