Reconciling Perspectives: A Substantive Theory of How People Manage the Process of Software Development

by

William Stephen Adolph

M.Sc, Simon Fraser University, 1988
B.A., Simon Fraser University, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY in

THE FACULTY OF GRADUATE STUDIES (Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA (Vancouver)

February 2013

ABSTRACT

Software development is a 1.6 trillion dollar industry and any improvement to how we create software will have significant economic benefit. As part of software process improvement programs, many organizations adopt a methodology. While there is evidence that methodologies improve development outcomes, few developers follow specific methodologies, preferring to use ad hoc practices. What is going on here? Is there a gap between the process models and how people actually create software? Could it be that software developers do not find methodologies useful and that methodologies do not provide guidance or support for how people really create software?

To answer these questions, we performed a field study to understand how the process of software development is actually managed. We used Grounded Theory to generate a substantive theory with the intention of using it to inform methodology design. We learned the main concern of people involved in the process of software development is getting the job done and different points of view and expectations create impediments. People use a four stage process of Reconciling Perspectives to remove these impediments. When a perspective mismatch is discovered, people converge their mismatched perspectives by reaching out and negotiating a consensual perspective. Constructing the work products and evaluating them validates the consensual perspective. The process may yield accepted work products, providing objective evidence the perspective mismatch was reconciled, or may result in waste. Reconciling Perspectives is a social process moderated by social dynamics, and creating accepted work products depends on individuals’ abilities to reach out and engage in negotiations while also sheltering themselves from distracting interruptions. This creates a tension in the process that must be managed because to discover and converge perspective mismatches, people must be open to interruptions while also remaining focused on getting the job done.

This study provides additional evidence that software development is a socio-technical process. Most software methodologies are biased towards a technical description of software as a rational process of reductive transformation and do not provide guidance for social roles. We use Reconciling Perspectives to construct guidelines that can inform the design of a scalable socio-technical software development methodology.
PREFACE

Parts of this dissertation have appeared in previously published work:


This dissertation makes use of copyrighted material for which permission has been obtained:

- Figure 1 CMM Levels of Software Process Maturity from (Thomson & Mayhew, 1997, p. 6) is republished with the permission of the copyright holder John Wiley and Sons

- Figure 4 OpenUP Detail Use Case Scenarios (Eclipse, 2012) is republished under the Eclipse Public License V1.0

Permission to republish the following figures:

- Figure 28 Relationship Between Performance and Communications Frequency (Patrashkova-Volzdoska, McComb, Green, & Compton, 2003, p. 266) © IEEE

- Figure 2 Classical Waterfall (Royce 1970 pg 330) © IEEE

was obtained with the posting of this notice:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author’s approval.

This research was approved by the University of British Columbia Behavioural Research Ethics Board (BREB), H08-2387.
## TABLE OF CONTENTS

**PREFACE** ......................................................................................................................... iii

**LIST OF TABLES** ............................................................................................................. ix

**LIST OF FIGURES** ......................................................................................................... x

**DEDICATION** .................................................................................................................. xiv

**Chapter 1** **INTRODUCTION – WHAT IS GOING ON HERE?** .................................. 1

1.1 Motivation for this Study ............................................................................................... 3

1.2 Software Engineering .................................................................................................... 5

1.2.1 Software Process Improvement ................................................................................. 10

1.2.2 Software Methodologies ....................................................................................... 12

1.3 A Gap in Software Engineering Research ...................................................................... 18

1.3.1 Software Development as a Social Process .......................................................... 20

1.4 Research Objective and Guide to this Thesis ............................................................. 23

**Chapter 2** **LITERATURE REVIEW PART 1: FRAMING THE STUDY** ................ 26

2.1 Grounded Theory and the Traditional Literature Review ............................................. 26

2.2 Who Uses Software Methodologies? ............................................................................ 27

2.3 Better Methodologies, Better Software? ....................................................................... 30

2.4 Adoption Models – What Motivates People to Use a Methodology? ......................... 33

2.5 The Social Nature of Software Development ............................................................. 35

**Chapter 3** **METHODOLOGY** .................................................................................. 40

3.1 Philosophical Stance .................................................................................................... 40

3.1.1 The Positivist Paradigm ....................................................................................... 41

3.1.2 The Interpretivist Paradigm ................................................................................. 42

3.1.3 Personal Reflections ............................................................................................ 44

3.2 The Design of a Grounded Theory Study .................................................................... 46

3.2.1 Why Grounded Theory? ..................................................................................... 47

3.2.2 Grounded Theory Overview ................................................................................. 48

3.2.3 Which Version of Grounded Theory? ................................................................... 50

3.2.4 The Grounded Theory Analysis Model ............................................................... 51

3.2.5 Theoretical Sampling ......................................................................................... 60

3.2.6 Theoretical Sensitivity ......................................................................................... 61

3.2.7 Memoing ............................................................................................................ 62

3.2.8 Sorting and Writing ............................................................................................ 63

3.2.9 Literature Search .................................................................................................. 64

3.2.10 Grounded Theory Rigor and Quality ................................................................. 65

3.3 The Study ...................................................................................................................... 68

3.3.1 The Inception of the Project ............................................................................... 68

3.3.2 The Research Problem ....................................................................................... 69
# Chapter 4

## THE THEORY OF RECONCILING PERSPECTIVES

### 4.1 Introduction

### 4.2 Overview

### 4.3 The Organizational Context of Reconciling Perspectives

#### 4.3.1 The Process of Software Development

#### 4.3.2 The Organizational Ecosystem

#### 4.3.3 A Central Concern: Getting the Job Done

#### 4.3.4 Acquirers and Suppliers

#### 4.3.5 Perspectives

#### 4.3.6 Perspectives Mismatches

### 4.4 Reconciling Perspectives

#### 4.4.1 Process Flow

#### 4.4.2 Outcomes

#### 4.4.3 Multiple Instances

#### 4.4.4 Tension in the Process

#### 4.4.5 Converging

#### 4.4.6 Validating

#### 4.4.7 Reaching Out

#### 4.4.8 Negotiating Consensus

#### 4.4.9 Bunkering

#### 4.4.10 Evaluating

#### 4.4.11 Cycle Time

#### 4.4.12 Scope

#### 4.4.13 Communications

#### 4.4.14 Social Dynamics

---

### 3.3.3 Ethics ................................................................. 70
### 3.3.4 Tools .................................................................. 70
### 3.3.5 Field Access ....................................................... 71
### 3.3.6 Study Scope ......................................................... 72
### 3.3.7 Site Description .................................................... 73
### 3.3.8 Data Collection ..................................................... 79
### 3.3.9 Interviews ............................................................ 80
### 3.3.10 Participant Observation .......................................... 82
### 3.3.11 Document Analysis ............................................... 83
### 3.3.12 Sampling ........................................................... 84
### 3.3.13 Analysis ............................................................... 87
### 3.3.14 Generating Concepts - Open Coding ...................... 88
### 3.3.15 Discovering the Central Concern: Selective Coding ... 90
### 3.3.16 Relating the Concepts: Theoretical Coding ............... 93
### 3.3.17 Memoing .............................................................. 94
### 3.3.18 Sorting and Writing .............................................. 95
### 3.4 Trusting Reconciling Perspectives .................................. 95
### 3.4.1 Threats to Credibility ............................................ 98
### 3.4.2 Member Checks and Feedback .................................. 100
### 3.5 Limitations of this Study .......................................... 101
Chapter 5  LITERATURE REVIEW PART 2: SITUATING RECONCILING PERSPECTIVES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Reconciling Perspectives and Social Cognition Theory</td>
<td>198</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Perspectives and Schema Theory</td>
<td>199</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Technology Frames of Reference</td>
<td>200</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Mental Models</td>
<td>201</td>
</tr>
<tr>
<td>5.2</td>
<td>The Organization as an Ecosystem of Communities</td>
<td>201</td>
</tr>
<tr>
<td>5.3</td>
<td>Are Software Projects Impeded by Perspective Mismatches?</td>
<td>202</td>
</tr>
<tr>
<td>5.4</td>
<td>Converging</td>
<td>204</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Reaching Out</td>
<td>205</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Negotiating Consensus</td>
<td>206</td>
</tr>
<tr>
<td>5.5</td>
<td>Validating and Bunkering</td>
<td>209</td>
</tr>
<tr>
<td>5.6</td>
<td>Tension in the Process</td>
<td>212</td>
</tr>
<tr>
<td>5.7</td>
<td>The Social Dynamics of Managing Communications</td>
<td>216</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Personal Strength</td>
<td>217</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Experience</td>
<td>218</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Leadership</td>
<td>220</td>
</tr>
<tr>
<td>5.8</td>
<td>The Influence of Social Processes on Software Development</td>
<td>225</td>
</tr>
</tbody>
</table>

Chapter 6  RECONCILING PERSPECTIVES AND SOCIO-TECHNICAL APPROACHES TO SOFTWARE ENGINEERING

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Software Development as Social Process</td>
<td>231</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Software Methodologies Are Not Used Because They Are Not Useful</td>
<td>233</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The Agile Manifesto and Social Processes</td>
<td>237</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Scrum as Social Process</td>
<td>238</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Agile Manifesto and Reconciling Perspectives</td>
<td>241</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Scrum and Reconciling Perspectives</td>
<td>242</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Are Social Processes Sufficient?</td>
<td>244</td>
</tr>
<tr>
<td>6.2</td>
<td>Software Development as a Socio-Technical Process</td>
<td>245</td>
</tr>
<tr>
<td>6.3</td>
<td>Refocusing Software Methodology Design</td>
<td>246</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Define Social Roles and Supporting Practices</td>
<td>247</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Favour Shorter Cycle Time</td>
<td>250</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Support Social Scalability</td>
<td>251</td>
</tr>
<tr>
<td>6.4</td>
<td>Is a Socio-Technical Methodology Sufficient?</td>
<td>257</td>
</tr>
</tbody>
</table>

Chapter 7  CONCLUSION

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Reconciling Perspectives: Summary</td>
<td>262</td>
</tr>
<tr>
<td>7.2</td>
<td>Explaining Why People Don’t Use Software Methodologies</td>
<td>267</td>
</tr>
<tr>
<td>7.3</td>
<td>Using Reconciling Perspectives to Inform the Design of Software Development Methodology as Socio-Technical Processes</td>
<td>269</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Define Social Roles and Supporting Practices</td>
<td>270</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Favour Shorter Cycle Time</td>
<td>271</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Support Social Scalability</td>
<td>271</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Team Training</td>
<td>271</td>
</tr>
</tbody>
</table>
7.4 Using Grounded Theory for Software Engineering Research ............................. 272
7.5 Contribution of this Study ............................................................................ 273
7.6 Future Research ......................................................................................... 275
    7.6.1 Elaborating Reconciling Perspectives .................................................... 276
    7.6.2 Create and Deploy a Socio-Technical Software Development Methodology .... 279
    7.6.3 Are Software Developers and Pilots the “Same Breed of Cat”? ................ 280
    7.6.4 What Can We Learn from Others? ....................................................... 281
    7.6.5 Creating Opportunities for Practitioners to Conduct Software Engineering
              Research .................................................................................................. 282
Bibliography ....................................................................................................... 283
Appendix A Contact Letter ................................................................................... 297
Appendix B Consent Form .................................................................................... 302
Appendix C Interview Protocol .......................................................................... 306
Appendix D Participant Observation Protocol .................................................... 309
Appendix E Sample Coded Interview ................................................................ 311
Appendix F Sample Participant Observation notes ............................................ 313
Appendix G Wiki Concept Page ........................................................................ 319
Appendix H Wiki Indicator .................................................................................. 325
Appendix I The Theory Emerges for the First Time .......................................... 327
Appendix J Glossary ............................................................................................ 329
LIST OF TABLES

Table 1: Hirschheim and Klein's Four Paradigms of System Development (Hirschheim & Klein, 1989, p. 1210) ................................................................. 19
Table 2: Methodology Usage (from Fitzgerald 2000) ........................................ 28
Table 3: Example Indicators and Codes ............................................................. 53
Table 4: Strategies Executed in This Study to Create an Audit Trail .................. 68
Table 5: Interview Subject Description ................................................................ 81
Table 6: Gasson’s Comparison of Trustworthiness Criteria between Positivism and Interpretivism ................................................................. 97
Table 7: Reconciling Perspectives Failure Path Flows ........................................ 106
Table 8: Reconciling Perspectives Properties ..................................................... 107
Table 9: Process of Software Development as Described by Interview Participants ........................................................................................................ 114
Table 10: Job Areas Where Perspective Mismatches May Occur ....................... 124
Table 11: Strategies for Negotiating Consensus ............................................... 155
Table 12: Scope Categories ............................................................................... 174
Table 13: Properties of Social Dynamics ........................................................... 180
Table 14: Reconciling Perspectives Leadership Roles ....................................... 190
Table 15: Three Stages of Mental Model Convergence (McComb, 2007, p. 103) .................................................................................................................... 208
Table 16: Types of Experience (Reagans, et al., 2005, p. 870) ......................... 219
Table 17: Reconciling Perspectives Leadership Roles ....................................... 221
Table 18: Four Themes for the Integration of Communities (Ferreira, et al., 2012) ............................................................................................................... 226
Table 19: Reason Why Rational Software Methodologies are Not Useful ....... 236
Table 20: Eight Project Contextual Factors (Kruchten, 2011) ......................... 253
Table 21: Reconciling Perspectives Failure Paths ............................................. 264
Table 22: Social Dynamics Properties ............................................................... 266
Table 23: Leadership Roles and Responsibilities .............................................. 267
Table 24: Reasons Why Software Methodologies are Not Useful .................... 268
LIST OF FIGURES

FIGURE 1: CMM LEVELS OF SOFTWARE PROCESS MATURITY (THOMSON & MAYHEW, 1997, P. 6) ......................................................................................................................... 11
FIGURE 3: SPEM CONCEPTUAL INPUT-PROCESS-OUTPUT MODEL (OMG, 2005, P. 3-2).. 15
FIGURE 4: OPENUP DETAIL USE CASE SCENARIOS (ECLIPSE, 2012) ......................... 16
FIGURE 5: THE GROUNDED THEORY PROCESS ................................................................ 49
FIGURE 6: CONCEPT INDICATOR MODEL COMPARING INDICATOR TO INDICATOR ...... 52
FIGURE 7: CONCEPT INDICATOR MODEL COMPARING SUBSEQUENT INCIDENTS WITH CONCEPT .................................................................................................................. 52
FIGURE 8: UML MODEL INDICATORS, CONCEPTS, CATEGORIES ................................. 54
FIGURE 9: CONSTANT COMPARISON AND CONCEPT EMERGENCE ............................ 57
FIGURE 10: CONSTANT COMPARISON GENERATION OF PROPERTIES ...................... 58
FIGURE 11: CONSTANT COMPARISON FURTHER ELABORATION OF PROPERTIES .... 59
FIGURE 12: SITE 1 OFFICE LAYOUT ................................................................................. 74
FIGURE 13: SITE 2 OFFICE LAYOUT ............................................................................... 76
FIGURE 14: SITE 3 OFFICE LAYOUT ............................................................................... 78
FIGURE 15: RECONCILING PERSPECTIVES .................................................................... 105
FIGURE 16: MULTIPLE OVERLAPPING INSTANCES OF RECONCILING PERSPECTIVES... 109
FIGURE 17: THE SOFTWARE DEVELOPMENT PROCESS ................................................. 113
FIGURE 18: CONVERGING: REACHING OUT ................................................................. 127
FIGURE 19: CONVERGING: NEGOTIATING CONSENSUS ............................................ 128
FIGURE 20: VALIDATING: BUNKERING ........................................................................... 130
FIGURE 21: VALIDATION: EVALUATING ......................................................................... 136
FIGURE 22: RECONCILING PERSPECTIVES: RECURSIVE OR NESTED FLOW LEADING TO ACCEPTED WORK PRODUCTS ................................................................. 142
FIGURE 23: RECONCILING PERSPECTIVES: WASTE CREATED AS A RESULT OF FAILING TO REACH OUT ......................................................................................... 151
FIGURE 24: RECONCILING PERSPECTIVES: WASTE CREATED AS A RESULT OF CONVERGENCE STALLING ..................................................................................... 158
FIGURE 25: RECONCILING PERSPECTIVES: ITERATIVE FLOW LEADING TO ACCEPTED WORK PRODUCTS ......................................................................................... 163
FIGURE 26: WASTE RESULTING FROM ISOLATION FLOW ............................................ 165
ACKNOWLEDGEMENTS

If I have seen further it is only by standing on the shoulders of giants.
– Sir Isaac Newton

While only my name appears on the cover of this dissertation, I could not have achieved this work without the help, guidance, and support of giants and I want to take a moment to thank all those giants who made significant contributions to this dissertation.

First, to my supervisory committee and if, in the academic world, there is a concept of a “dream team” then I had the “dream team”. It is unusual for a middle age man to return to school to pursue research and I want to thank my entire committee for the flexibility and support they gave me as a “mature” student. Thank you to my supervisor Philippe Kruchten who gave me the inspiration and flexibility to pursue my passion to explore new territory and the guidance to keep it under control. Philippe was always there with an insight, a paper, and often an introduction that guided the emergence of our theory. Philippe was also there when it came to navigating university rules and regulations and was supportive when I hit some personally challenging moments.

Every grad student needs a supervisory committee member like Wendy Hall, who is dedicated to guiding new scholars through the maze of methodology and research and who gives them the hope they may one day finish. Wendy provided me with wonderful guidance on classical grounded theory and helped bring greater scholastic discipline to this dissertation.

Finally Karl Aquino who lead one of the best seminars in Organizational Behaviour I have attended and introduced me great things about Organizational Behaviour and was there when I needed a question answered over coffee.

To the organizations that gave me permission to collect data at their sites, thank you and thank you to those who helped me negotiate access to field sites: Mike McCallister, Barbara Hunt, Finlay Canon, Lawell Kiing, and Michael Vax. This research would have been impossible without your help.

To all those who participated in this study, thank you for permitting me to peer into your daily lives and understand how you saw the process of software development.
I do not believe it is possible to construct a dissertation on one’s own and I need to thank two people who helped me write this one. First, thank you to my transcriptionist Steve Noble who quickly and accurately typed up my recorded interviews. I simply cannot imagine myself being able to do the same. Thank you, also, to Susan Patch who copy edited my writing and helping me write better grammar and created my diagrams.

Thank you to my employer Rally Software for giving me the leave to complete this dissertation.

To all those who provided feedback and comments on the theory: Tim Dudra, Mark Kilby, Christopher Avery, Alistair Cockburn, and the anonymous reviewers at Empirical Software Engineering and the Journal of Systems and Software, thank you. Your insightful feedback helped make this a much better dissertation.

Thank you to the authors of Microsoft Word and Mac OS who provided me with numerous opportunities for unscheduled breaks and gave me the discipline to start taking frequent check points of my work.

This research was generously supported by: the Agile Alliance, the Eclipse Foundation, and MITACS.

Thank you to my colleagues and friends who gave me encouragement that I was doing something worthwhile and helped me through some of those dark moments when I was thinking “why the <bleep> am I doing this?” and was ready to throw in the towel.

Finally, and not least, thank you to my wife Lesya and daughter Sophia, who have patiently put up with lost family time as I strived to complete this dissertation. It has not been easy for them, and I want you both to know I appreciate the sacrifices and accommodations you have made for me. Thank you and I love you both dearly.
DEDICATION

In memory of my parents Marcelle and Don
The Blind Men and the Elephant

A HINDU FABLE.
From the poems of John Godfrey Saxe (1872)

I.
It was six men of Indostan
To learning much inclined,
Who went to see the Elephant
(Though all of them were blind),
That each by observation
Might satisfy his mind.

II.
The First approached the Elephant,
And happening to fall
Against his broad and sturdy side,
At once began to bawl:
“God bless me!—but the Elephant
Is very like a wall!”

III.
The Second, feeling of the tusk,
Cried:“Ho!—what have we here
So very round and smooth and sharp?
To me ’t is mighty clear
This wonder of an Elephant
Is very like a spear!”

IV.
The Third approached the animal,
And happening to take
The squirming trunk within his hands,
Thus boldly up and spake:
“I see,” quoth he, “the Elephant
Is very like a snake!”

V.
The Fourth reached out his eager hand,
And felt about the knee.
“What most this wondrous beast is like
Is mighty plain,” quoth he;
“T is clear enough the Elephant
   Is very like a tree!”

VI.
The Fifth, who chanced to touch the ear,
   Said: “E’en the blindest man
Can tell what this resembles most;
   Deny the fact who can,
This marvel of an Elephant
   Is very like a fan!”

VII.
The Sixth no sooner had begun
   About the beast to grope,
Than, seizing on the swinging tail
   That fell within his scope,
“I see,” quoth he, “the Elephant
   Is very like a rope!”

VIII.
And so these men of Indostan
   Disputed loud and long,
Each in his own opinion
   Exceeding stiff and strong,
Though each was partly in the right,
   And all were in the wrong!
“it’s getting everybody to understand things in a similar manner. Get everybody on the same page. I think that’s – that’s the biggest impediment is people walk away with misunderstandings and you haven’t clarified. And they just run with their assumptions or what the way they understand things. And then we either have to go and correct them or back out of it or redo um something. And I think that’s the thing that causes us the most pain. I spend the most time on in terms of preventing and mitigating” – Site 1 Subject 1 follow-up

“…there just weren’t enough conversations taking place” – Site 2 Subject 3

“So, it just means time-wise ah you can't bug them about every little detail. They've got to do a lot of context switching because they have so many features they're working on” – Site 3 Subject 6
Chapter 1

INTRODUCTION – WHAT IS GOING ON HERE?

Software development is a risky and expensive proposition, and talking with software developers quickly reveals the quagmire that software development can become. Late deliveries, defective products, cost overruns, and ruined careers and businesses are some of the debris resulting from failed or less-than-successful projects. This is expensive debris, considering that global software development is a 1.6 trillion US dollar industry (Bartels, Holmes, & Lo, 2006). Improving the productivity of software development teams and the quality of delivered software will result in significant economic returns.

Software methodologists argue the benefits of following a software development methodology, and there are studies that demonstrate a positive correlation between software development methodology adoption and team effectiveness in terms of product quality and productivity (Diaz & Sligo, 1997; Harter, Krishnan, & Slaughter, 2000; Krishnan, Kriebel, Kekre, & Mukhopadhyay, 2000). On the other hand, industry data also demonstrates that the effect of software methodologies on software development productivity is limited (Jones, 2000; Sawyer & Guinan, 1998), and the contribution of software methodologies to team effectiveness is frequently questioned (Cockburn, 2003; Dybå, Moe, & Arisholm, 2005; Fitzgerald, 1998). The level of methodology adoption is also questionable. One well known software development celebrity summed up the situation as “the picture of software designer deriving his design in a
rational, way from a statement of requirements is unrealistic. No system has ever been
developed this way, and probably none will” (Parnas & Clements, 1985, p. 251).

What is going on here? Is it possible that the low rates of software methods adoption are a
strong indicator that software practitioners do not believe their needs are being addressed by
software methodologies? While there are anecdotes about methodology usage, or lack thereof,
there is a gap in our empirical knowledge about the interface between software developers and
methodologies. In an effort to narrow this gap, we undertook this field study of how people
manage the process of software development.

In this section, we explain our motivation for conducting this research, starting with sub-section
1.1 which explains my personal motivation as part of the research team. Sub-section 1.2 defines
software engineering and distinguishes its goal from other disciplines such as Computing
Science and Information Systems. Sub-section 1.2.1 explains software process improvement,
and sub-section 1.2.2 defines what we mean by software methodology and the paradox of
software methodology usage. Sub-section 1.3 explores the apparent gap between software
engineering research and how software development is actually practiced and, finally, sub-
section 1.4 explains our research objective, our research question, and provides an overview of
this dissertation.
1.1 Motivation for this Study

All together, the principal researchers have designed and managed large software projects in telecommunications and aviation for more than 50 years, and we have often wondered why software methodologies are not well received or used on software projects. In our professional careers, we have participated in various projects where a lack of software engineer discipline severely challenged the project. Software engineering discipline in these situations means following some agreed on set of technical practices for creating working products: Are we conducting reviews? Who is doing the testing? What level of test coverage? How and when will we integrate components? What is a component? What are our goals, and milestones? When is it done? Without some agreement on a set of technical practices for creating software, planning and project management become bad examples of management by walking around – walking around and pestering development staff asking “when are you going to be done?”

We continue to see people who are creating software work long desperate hours to push a challenged project out the door, and we have done the same ourselves. We have seen harried project managers chewed out by executive staff during project status meetings. We have seen projects push back delivery dates time and time again, sometimes with serious consequences for the firm. We have seen individuals demoted for challenged projects, and we have seen careers halted and broken. Certainly the root cause for these situations cannot be solely attributed to the lack of software engineering discipline; in many of these situations, issues such as poor product

1 At the time of writing this dissertation, Canadian mobile phone maker RIM had announced yet another delay of their next generation handset because of challenges integrating the software. Most analysts do not believe the company will survive - EL Akkad, O. (2012, June 28 2012). RIM to chop 5,000 jobs, delays launch of BlackBerry 10. The Globe and Mail.
management and marketing were significant challenging factors. However, as software engineers, we should be able to reduce the contribution that the lack of software engineering discipline makes towards challenging a project. We have personally been part of far too many crisis management teams and conducted far too many post-mortems where we were left thinking “this could have been avoided”.

Software methodology is the foundation of good software engineering discipline because it establishes a shared common vision for how individuals participating in a software project will work together to deliver a product. Software methodologies generally describe individual roles, collaboration between roles, the work products required, how the quality of the work products is judged, and the processes to create and manage those work products. Our personal experience is backed by studies that demonstrate a positive correlation between software development methodology adoption and team effectiveness in terms of product quality and productivity (Diaz & Sligo, 1997; Harter, et al., 2000; Krishnan, et al., 2000).

Yet people are reluctant to adopt methodologies, and see the arrival of the methodologist black belts (Snee & Hoerl, 2003) as a traumatic event, like the arrival of a tax auditor, rather than a supportive experience. Or, in situations where methodologies have been adopted, their adoption may have been done more for show because, when the inevitable schedule pressure comes, our experience is that people abandon methodological practices. We would assume that, if people found the practices helpful and supportive, they would embrace them even more tightly when under pressure rather than abandoning them altogether.
Many individuals see software methodologies as “too heavy”; that is, they have too much overhead and are not worth the cost. A frequent comment regarding software methods is that “*We could never get it done if we had to develop software doing that*”. Industry data suggests that the effect of software methods on software development productivity is limited (Jones, 2000; Sawyer & Guinan, 1998). Furthermore, the contribution of software methods to team effectiveness is frequently questioned (Cockburn, 2003; Dybå, et al., 2005; Fitzgerald, 1998).

Since 1998, I have made a very good living as a consultant helping organizations adopt the current fad technology – use cases, object-oriented software development, patterns, and agile methods – to transform the engineering organization. In all of my engagements, I was brought in because of a concern the engineering group could not deliver on its commitments and the assumption was if we trained the engineering group on the current fad technology then the situation would improve. On the first assessment visit to a client site, it usually became clear that there was no technology that would solve their problems because their issues were social. In my personal opinion, the desire for new technology deployment was, in reality, an attempt to foster a social change without formally admitting that their organization had a social problem. Therefore, in my opinion, we cannot research this question strictly as a technical issue; we must also investigate the social or so called “people” side of the equation.

### 1.2 Software Engineering

Engineering is about building things, from consumer products to industrial infrastructure, that are safe to use and economical for their purpose. Few people think twice about crossing a bridge,
living in a condominium tower, or flying to visit friends and relatives, which is a testament to our success at applying scientific methods to building things. According to Britannica.com ("engineering," 2012), ABET (once known as the Accreditation Board for Engineering and Technology) defined “engineering” as:

“The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation or safety to life and property”.

Other engineering disciplines emerged as a profession in response to the failure of constructed works such as collapsing bridges and exploding boilers (Parnas, 2002); software engineering emerged forty years ago during a NATO scientific conference (Naur & Randell, 1968) in a response to the realization that large software systems were frequently delivered late – if delivered at all – were over budget with feature deficiencies, and had suspect quality.

The IEEE defines software engineering as (IEEE, 1990):

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1).
Barry Boehm (1976), one of the grand patriarchs of software engineering, captures the essence of software engineering as “Software engineering is the means by which we attempt to produce all of this software in a way that is both cost-effective and reliable enough to deserve our trust” (p. 1226).

Software engineering is about how we build software; it is distinct from Computer Science, where the primary concern is algorithms and heuristics, and from Information Systems, where the concern is managing and exploiting organizational information. In short, we can characterize these disciplines by the types of questions about software that they answer: Information Systems answers the question “What should it do?”, Computing Science answers the question “How will it do it?”, and Software Engineering answers the question “How do we build it?”

Software engineering enjoys the same challenges as other engineering disciplines in terms of the complexity of the problem domain and managing a complex development process. What makes software development unlike other engineering disciplines, however, is the flexibility possible through software and the nature of discrete systems (Booch, 1991). The near unlimited flexibility of software is immediately obvious to anyone who has seen a Hollywood blockbuster movie lavishly enhanced with computer generated imagery (CGI): cities sinking into the sea as the Earth comes to an end; space ships engaged in epic battles of good versus evil; and elegant blue creatures living in a symbiotic relationship with the life force of their planet battling a greedy mining corporation. Thirty years ago, when I started in this industry computers were expensive and memory-limited, which constrained the design space and the design options a programmer could consider. A good programmer knew how to save a few bytes of memory and
shave a few milliseconds off a loop. The rapid evolution of micro-electronics has removed these constraints such that the only constraints on software development are imagination and the ability to manage the realization of that imagination. The lack of physical constraints is one source of complexity that differentiates software development from other engineering disciplines.

The other source of complexity that differentiates software development from other engineering disciplines is that software systems are discrete systems. Most other engineering disciplines work with continuous systems where the systems’ responses (outputs) vary continuously and predictably with its inputs. Software is a discrete system with “memory” or “state” where outputs vary with both input and state, and small variations in output cannot be assumed from small variations in input. This is why software systems tend to fail catastrophically, and do not “degrade gracefully”. Fredrick Brooks (1982) eloquently likened this trait of computer programming to the magic of legend: “If one character, one pause, of the incantation is not strictly in proper form, the magic does not work” (p. 8).

Other engineering disciplines apply science to create theoretical models that can predict the behaviour and performance of an engineered system before it is built. Our great grandparent engineers used such models to predict the load that a bridge could support before it was built, or to predict the pressure a boiler could withstand and the amount of steam it would generate. Such models will likely elude us because we cannot even predict in general terms if a program will halt ("Why is Turing's halting problem unsolvable?," 2006). Brooks observed that complexity is the essence of software, and the engineering models that allow other disciplines to abstract away
complexity will not work when such complexity is the essence of the problem (Brooks, 1987).

Despite software’s essential complexity, and these theoretical limitations to managing complexity, the software industry has done rather well because every day software is constructed, airplanes fly safely and efficiently through the air despite being uncontrollable without auto-pilot assistance, phone calls are routed, cars start, tractors plow fields with a precision no human operator can match, power is reliably generated and distributed, credit is granted, holidays are booked, billions in retail transactions are executed, and people play Angry Birds\(^2\) on hand held devices. This is an industry that is creating enormous benefit and wealth despite the theoretical challenges. As software engineers, we obviously have done something right and, like all other engineers, we always want to do better.

Understanding and improving the ways in which software is developed is a prominent part of software engineering research (Fuggetta, 2000), and includes (Sjoberg, Dybå, & Jorgensen, 2007):

1. the development of new, or modification of existing, technologies (process models, methods, techniques, tools, or languages) to support software engineering activities, and

2. the evaluation and comparison of the effect of using such technology in the often very complex interaction of individuals, teams, projects, and organizations, and various types of task and software systems.

\(^2\) Angry Birds is a registered trademark of Rovio Entertainment Ltd.
1.2.1 Software Process Improvement

Any improvement to a 1.6 trillion US dollar industry will result in significant economic benefit. A survey of software process improvement experience reports suggests that organizations engaged in software process improvement are enjoying a return on investments of 400 to 500% (van Solingen, 2004). To answer the question “what is a good software process and how do we improve ours”, several models of software process improvement have evolved over the last 30 years: SPICE (Emam, 1997), ISO 9000 (Kehoe & Jarvis, 1996), and Six Sigma (Holpp, 2002). Arguably, the most widely known software process improvement models are the so-called “maturity models” represented in the Capability Maturity Model (CMM) (Paulk, 2002; Watts, 1989) and its successor, the Capability Maturity Model Integrated (CMM-I) (Team, 2002).

The assumption of a maturity model is that a mature organization possesses an organization-wide ability to manage software development (Paulk, Curtis, & Chrissis, 1993). An immature organization improvises and moves from one crisis to another – often called firefighting – and routinely experiences schedule and budget over-runs and product quality issues. There is a high degree of variance in an immature organization, and processes are not repeatable. As an organization matures, variances are reduced, and the software process becomes better defined, and is more consistently implemented and followed throughout the organization. Roles, work products, and processes are well defined and updated when improvements are developed. Budgets and schedules are based on historical performance data and are realistic. There is an objective quantitative means for judging product quality, and for analyzing problems with both the product and the process.
CMM (and its successor, CMM-I) is a multi-stage model characterizing an organization’s software development “maturity level”. Organizations are assessed as being at one of five levels of software development maturity: Initial (1), Repeatable (2), Defined (3), Managed (4), and Optimizing (5).

The foundation of a mature software organization is it follows a defined software process, one that is defined by a documented rational software development methodology.

Figure 1: CMM Levels of Software Process Maturity (Thomson & Mayhew, 1997, p. 6)
1.2.2 Software Methodologies

If software engineering is about “how we build it”, then software methodologies are the rules and guidelines for building it. Many in the field of software development tend to use the terms “software method”, “software methodology”, and “software process” interchangeably, and it is therefore important that we clarify our usage of the terminology. The IEEE (2010) defines a software methodology as (p. 216):

1. a system of practices, techniques, procedures, and rules used by those who work in a discipline.

2. specification of the process to follow together with the work products to be used and generated, plus the consideration of the people and tools involved, during a development effort.

This definition is further elaborated in a note: “A methodology specifies the process to be executed, usually as a set of related activities, tasks and/or techniques, together with the work products that must be manipulated (created, used or changed) at each moment and by whom, possibly including models, documents and other inputs and outputs. In turn, specifying the models that must be dealt with implies defining the basic building blocks that should be used to construct”

In his popular software engineering text, Robert Pressmen (2001) used the term “methods” and defined software engineering methods as providing “the technical how-to’s for building software. Methods encompass a broad array of tasks that include requirements analysis, design,

---

3 In the IEEE vocabulary the term “method” is not related to software development practices.
program construction testing and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques” (p. 21). David Parnas described software methodologies as the equivalent of the Philosopher’s Stone⁴, a rational software process that would give us a good reason for doing what we’re doing. Parnas and Clements (1985) expressed their ideal vision for a software methodology, writing “ideally we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof” (p. 251). Parnas and Clements also stated clearly that the Philosopher’s Stone did not exist because the complexity of software makes an ideal rational process impossible. Despite this, they argued it is still beneficial to approximate or, as they put it, “fake it”.

Most software methodologies try to approximate or “fake” a rational process by following a reductionist framework describing the successive transformation of work products as they flow from one stage to the next (Wastell, 1996). The classic waterfall embodies this reductionist framework and models software development as a linear series of processes where the outputs of the prior process flow down as inputs to the next (Royce, 1970).

---

⁴ The Philosopher’s Stone is the legendary alchemist magnum opus that could turn base metal into gold. The analogous software methodology could turn an ad hoc software development process into a purely rational mathematical process.
In the classic waterfall model, the outputs from the upstream process must pass through a “stage gate” where they are verified and validated before they can flow through as inputs to the next process. The benefit of this model is that subsequent stages receive inputs that are complete and have been verified and validated so they are not progressing the development with suspect and ambiguous inputs. For example, a software development project cannot move forward from the Software Requirements process to the Analysis process until the established standards for the output of the Software Requirements process are complete, verified, and validated.

While modern software engineering methodologies have moved towards more iterative and incremental models of software development, such as the Unified Process (UP) (Jacobson, Booch, & Rumbaugh, 2012) and its variants, they still model software development as a technical process of reductionist transformation. Each process stage can be described in terms of an Input-Process-Output model (IPO), where inputs arrive and are then processed by transforming them into outputs. This model enables methodology designers to specify the inputs
to a process, the outputs, the quality of those outputs, the steps for transforming the inputs into outputs, and the individuals responsible for animating the process.

This IPO meta-model is standardized in the *Software Process Engineering Meta-model* (SPEM) (OMG, 2005). Many of the modern UP software methodologies, including the Rational Unified Process (RUP\(^5\)) and OpenUP, are built using SPEM as a meta-model. Figure 3 is a meta-model for this IPO process:

![Figure 3: SPEM Conceptual Input-Process-Output Model (OMG, 2005, p. 3-2)](image)

Figure 4 is an example of a process description for a methodology using the SPEM meta-model, in this case OpenUP (Kroll & MacIsaac, 2006). This description shows the process (or steps) of “Detail Use-Case Scenario” which takes a Use Case as its input and its output is a Glossary, a Use Case (likely updated), and a Use-Case Model. The Analyst is primarily responsible for animating this process (performing the steps associated with this process), and may be assisted by the Architect, Developer, Stakeholder, and Tester.

\(^5\) RUP is a registered Trademark of IBM Corporation
SPEM standardizes software methodology engineering to avoid the proliferation of idiosyncratic methodologies that make learning and adapting methodologies difficult (Henderson-Sellers, 2003). This is an example of developing a technical standard to rigorously and unambiguously define methodologies and avoid the differences in points of view resulting from ambiguity caused by the use of different terminology and different representations of practices in different software methodologies.

The problem with software methodologies and the efforts to standardize their definition is it seems no one actually follows them unless coerced to do so (Fitzgerald, 1997; Huisman & Iivari, 2006; Nandhakumar & Avison 1999; Ward, Taylor, & Bond, 1996). Time and time again, when I ask developers why they do not follow the corporate software development methodology, or
adopt a “rational” methodology, the response I get is “...because if we ever did it like that we would never get it done”. Pressman also voiced the perception that “many perceive quality assurance activities are time consuming rather than time saving” (Pressman, 1996, p. 16). In contrast to Parnas’ idealized “faked” rational process, my observation of the methodology-in-use is that it is usually an ad hoc collection of practices agreed to on a situational basis by those immediately engaged in the software development process. The “faking” part is more often the arbitrary and coercive creation of work products (e.g. detailed documentation) that produces a veneer or fiction of a rational process that conceals the irrational process (Nandhakumar & Avison 1999). The strategy of “faking” a rational process is not serving development teams well in practice, and Nandhakumar and Avison (1999) concluded from their study that “formal methodologies are too mechanistic to be of much use in the detailed, day-to-day organization of developers activities” (p. 187). Further, they asserted the risk of faking such a process is that “Seeking to impose methodologies to improve the productivity of the developers' task may be counter productive” (p. 188).

An argument may be made that we are taking methodologies too literally, and that methodologies should be viewed as metaphorical, providing a means by which the process of software development can be understood (Hughes, 1998). I have spent much of my career interpreting methodologies for my clients and helping them take their methodologies less literally; however, my ability to perform this service for my clients is drawn from my twenty five
years of industry experience and not from any guiding set of software engineering theories. Whenever I am asked how I knew what to do, my answer is always “it just seemed obvious”\textsuperscript{6}.

Regardless of whether we should take methodologies as a literal and specific guide for creating software, or use them metaphorically, software engineering researchers should be concerned by this gap between software engineering research and how practitioners develop software in the field.

### 1.3 A Gap in Software Engineering Research

In an editorial, Nimal Jayartna and Ian Sommerville (1998) lamented that, after a decade of software engineering methodologies and techniques becoming more formal, there was little evidence that these techniques were leading to order of magnitude improvements in the practice of software engineering. I am in full agreement with their assertion that formal methods, models, and theories do not help us make sense of a disorderly and chaotic real world. Force fitting rationality onto disorder and chaos not only creates gaps, it also creates distortions and unrealistic expectation for how software should be constructed.

Hirschheim and Klein (1989) dimensionalized this gap, building on Burrell and Morgan’s (1979) social paradigms to identify four ideal types of information systems development approaches. The Burrell Morgan paradigms classify social theory along two dimensions. The first dimension is the objective-subjective dimension, which asks if we view the world as one in which

understanding is best achieved through the application of scientific method or through direct experience. The second dimension is the order-conflict dimension, which asks if we view the social world as one stressing stability and order or one stressing conflict and change.

The resulting four paradigms of information systems development approaches are described in Table 1.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Systems Development Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Functionalist</em> (objective-order)</td>
<td>The developer is viewed as an expert, and systems development proceeds by the application of formal concepts through planned intervention with rationalistic tools and methods.</td>
</tr>
<tr>
<td><em>Social Relativist</em> (subjective-order)</td>
<td>The developer is viewed as a facilitator, and systems development proceeds by subjective understanding and cultural sensitivity through adapting to internal forces of evolutionary change.</td>
</tr>
<tr>
<td><em>Radical Structuralist</em> (objective-conflict)</td>
<td>The developer is viewed as a partisan, and systems development proceeds by seeking to raise ideological consciousness through organized political action and adaptation of tools and methods to different social classes of interest.</td>
</tr>
<tr>
<td><em>Radical Humanist</em> (subjective-conflict)</td>
<td>The developer is viewed as an emancipator, and systems development proceeds by improving human understanding and the rationality of human action through emancipation of suppressed interests and the liberation from unwanted natural and social constraints.</td>
</tr>
</tbody>
</table>

*Table 1: Hirschheim and Klein's Four Paradigms of System Development*  
*(Hirschheim & Klein, 1989, p. 1210)*

Engineers who have been trained to view reality from a rational and objective point of view may have some discomfort with viewing software development from a perspective other than functionalist and, given that the nature of engineering is to find order, the conflict paradigms
may seem just that – radical. In my personal opinion, we are doing a disservice to ourselves and to our clients if we ignore the other paradigms because the most successful companies I have worked with all fitted – perhaps unknowingly – into the radical humanist paradigm and viewed their services and products as “emancipating”. This table demonstrates that there are more ways to view software development than as a functionalist rational process, and not doing so limits our ability to improve the software development process because we are ignoring dimensions of the issues that cannot be addressed by the application of natural science principles.

1.3.1 Software Development as a Social Process

It is immediately clear to an observer – watching requirements elicitation, Joint Application Development (JAD) meetings, planning meetings, problem solving meetings, lunch discussions, one-on-one consultations, and design negotiations over the espresso machine, and Nerf dart wars – that software development is a social process. Other researchers have made the same observation: “Software Engineering is primarily a social and creative process, where the creativity, skill, and co-operation of developers, users, and procurers determine the quality and effectiveness of the developed software” (Fuggetta, 1999, p. 135). The social processes are not a side effect of the software development process; the social processes are the control mechanism – as was highlighted by Nandhakumar and Avison’s “The evidence suggests that social controls, such as norms promoting collaboration with colleagues, professional design practices and established routines appeared to be a more significant influence on developers' work practice at LMC than the requirements of a methodology” (Nandhakumar & Avison 1999, p. 188). Put in modern agile shorthand: “people trump process” (Cockburn & Highsmith, 2001).
A social process is the way in which we interact with each other (Ginsberg, 1939), and numerous studies demonstrate that individual abilities and team social factors are significant cost drivers for software engineering projects, often swamping all other factors (Boehm, 1984; Cockburn, 2002; Cockburn & Highsmith, 2001; Curtis, Krasner, Shen, & Iscoe, 1987; Jones, 2000; Lister & DeMarco, 1987; Sawyer & Guinan, 1998). Caper Jones’ (2000) data demonstrate that high management and staff experience contribute 120% to productivity while effective methods/processes contribute only 35%. Cost drivers associated with personal factors from the Constructive Cost Model (COCOMO) “reflect the strong influence of personal capability on software productivity” (Boehm et al., 1995, p. 86). Earlier, Boehm (1981) concluded: “Personnel attributes and human relations activities provide by far the largest source of opportunity for improving software productivity” (p. 666).

If social factors are the biggest cost drivers and explain variances in the productivity of software development teams, research that helps us identify and understand social processes in software engineering should yield significant benefit to the industry. Unfortunately, most software engineering research is focused on tools and production methods, which have limited ability to account for and manage the variance in software projects (Glass, Vessey, & Ramesh, 2002; Sawyer & Guinan, 1998; Shaw, 2003; Sjoeberg et al., 2005; Zannier, Melnik, & Maurer, 2006). Comparatively little empirical research has been undertaken about the social processes that influence software development. Shaw’s (2003) analysis of papers accepted by the International Conference on Software Engineering (ICSE) shows that the majority of accepted papers “…reports an improved method or means of developing software that is, of designing, implementing, evolving, maintaining, or otherwise operating on the software system itself.”
Papers addressing these questions dominate both the submitted and the accepted papers” (Shaw, 2003, p. 727).

Procedures or techniques and tools and notation accounted for the majority of accepted papers (69%), while less than 10% of the accepted papers described empirical or qualitative models. Another survey of the software engineering research literature (Glass, et al., 2002) reported a very small percentage (in many cases less than 1%) of research papers explored organizational issues or employed research methods useful in understanding social behavior.

While more researchers are investigating the influence that personal attributes and human relationships have on software projects (Dittrich, John, Singer, & Tessem, 2007), Curtis, Krasner, Shen, and Iscoe’s (1987) retelling of the 13th century story of a man who, after losing his keys, crosses the street to search under a lamppost because the light is much better there, still sums up the state of software engineering research. This should not come as a surprise to us because we are engineers and not sociologists. Engineers – especially at the research level – are engineers because they enjoy solving technical problems. The activity of developing tools and methods is what we are educated to do and what we enjoy doing. If we are to better understand the factors that have the greatest influence on software development performance, our research must include study of social processes. We are looking specifically for guidance about which social theories are relevant so that we can use them to inform software engineering policy.
1.4 Research Objective and Guide to this Thesis

At the beginning of this section, we asked the question “What is going on?” in reference to the paradox that, while software engineering research suggests that rational software methodologies are useful, few practitioners willingly apply them, and practitioners often believe that methodologies are actually detrimental to getting the job done in a timely and cost effective manner. We further argued that this reveals a gap between software engineering research and the practice of software development. Our research objective, therefore, is to understand how people actually manage the process of software development and to generate a theory that explains the process. Such a theory will provide us with a model for reasoning about software processes that can be used to inform the design of software methodologies.

Theory is the foundation of science because theory provides explanations about and understanding of phenomena as well as testable predictions. The Oxford dictionary defines theory as “a supposition or a system of ideas intended to explain something, especially one based on general principles independent of the thing to be explained” (theory). Mature disciplines have a well-established base of theories that codify and rationalize well understood knowledge, and provide a basis for identifying and interpreting new work (Hall, Baddoo, Beecham, Robinson, & Sharp, 2009). Theory is more than an academic exercise for researchers, and it is important for practitioners also because, according to Schön, practitioners do not consider that they have formed a satisfactory account of phenomena in any practice situation until they have framed it in terms of their overarching theory, a rationale according to which they explain the situation and choose their acts (Schon, 1983). As a young discipline, the development and use of theories in
software engineering is limited (Hannay, Sjoberg, & Dybå, 2007), so there is benefit in a study that can generate useful software engineering theory from field data.

We chose Grounded Theory (Glaser & Strauss, 1967) as our research method because Grounded Theory is useful for answering the question “What is going on here?” by generating a substantive theory that explains how people manage their central concern. We can then use the generated theory to inform policy for the design of software methodologies. Therefore, as we are interested in understanding the process by which people create software, our research question for this study is:

“How do people manage the process of software development?”

As this is an engineering dissertation, my objective is not to generate theory as an abstract piece of knowledge; rather, I want to use the generated theory to make useful recommendations for the design of software methodologies.

This dissertation is presented in seven sections: the first is this section, which explains my motivation for conducting this research. The second section is the preliminary literature review that frames and establishes the context for this study. Section three explains the use of classical grounded theory for this study, and how field data were acquired. Section four presents the generated substantive theory of *Reconciling Perspectives*. Section five compares *Reconciling Perspectives* to extant theory to further elaborate the theory. Section six uses *Reconciling Perspectives* to...
Perspectives to recommend policy for the design of software engineering methodologies. Finally, section seven summarizes the results and offers a roadmap for future research.

Throughout this dissertation, I employ a typographical convention to identify the conceptual elements of the theory. Conceptual elements are presented using italicized fonts such as Reconciling Perspectives, or Cut-Off. There are places in this dissertation where grammatical considerations force me to adjust the spelling of concepts such as Getting the Job Done and Get the Job Done. Both of these names represent the same concept and concept names are sufficiently differentiated there should be no confusion regarding the name.
Chapter 2

LITERATURE REVIEW PART 1: FRAMING
THE STUDY

2.1 Grounded Theory and the Traditional Literature Review

Barney Glaser advocates performing the literature review after a substantive theory emerges: “do not do a literature review in the substantive area and the related areas where the research is to be done” (Glaser, 1998, p. 67). This is not an embargo against a literature search prior to study; rather, Glaser is mitigating the risk that extant theory may bias the researcher and potentially lead to the researcher forcing the data to fit a pet extant theory. In response, I used this first literature review to broadly “frame our research” to verify that my chosen area of study is one that is valuable to the industry, and to discover existing empirical research to support my arguments for the necessity of this research. As a substantive theory emerged, I followed Barney Glaser’s recommendation and conducted a second literature review to compare the emerging substantive theory with extant theory and situate it within the known theoretical landscape.

This first literature search was driven by what I regarded as a gap between software engineering theory and practice. In my personal experience, few developers in the field rigorously follow a software methodology, although many software engineering theorists and researchers highlight the benefits of software methodologies as part of a software process improvement. One of the first questions is: are my observations and beliefs simply an anomaly, or is there a real dissonance between software engineering theorists and the industry? If there is dissonance, then
what are some of the reasons for it? What empirical research has already been conducted? What questions remain?

2.2 Who Uses Software Methodologies?

Software engineering authors, such as Pressman (2001), Booch (1996), Kroll (2006), Kruchten (2004), and Humphrey (1989), identify the benefits of applying software methodologies, including: predictability, repeatability, and higher quality products. There are numerous experience reports and studies demonstrating the benefits of adopting or enforcing a software methodology as part of a software process improvement initiative (Agrawal & Chari, 2007; Basili & Green, 1994; Diaz & Sligo, 1997; Dion, 1993; Harter, et al., 2000; Ward, et al., 1996).

Given the advocacy for the use of software methodologies, there is a surprising lack of research about software methodology usage. Kautz, Hansen, and Jacobsen (2004) observed, in their study of software methodology usage in Denmark, that “[t]here is little empirical documentation about the actual use of development methods”. Griffin and Brandyberry’s (2010) review of systems development methodology usage research found only 14 quantifiable academic studies between 1986 and 2001. Of these 14 studies, five were conducted in the UK, one each in New Zealand and Finland, and three in Asia. Only two were conducted in the largest and most influential software development industry, the US.

Ward, Taylor, and Bond (1996) conducted a mail survey of IT/IS benefits with the top UK companies. Of the 250 surveys mailed out, 60 responses were received, of which 52% of
respondents indicated they followed a software methodology, and 15% of respondents indicated they did not use a methodology. Fitzgerald’s frequently cited survey (1997) seemingly contradicts Ward’s results, and supports the anecdotal view of many professionals that the cost of methodologies are not worth the benefits. Fitzgerald’s study also employed a postal survey, mailed to 776 individuals in UK software organizations. Of the 162 responses received, some of which were followed up by field interviews, 60% of organizations were not using a formalized method. Table 2 is reproduced from Fitzgerald’s data (2000).

<table>
<thead>
<tr>
<th>Methodology Usage</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizations not using any methodology</td>
<td>60</td>
</tr>
<tr>
<td>Organizations using a formalized commercial methodology</td>
<td>14</td>
</tr>
<tr>
<td>Organizations using internal methodology based on a commercial one</td>
<td>12</td>
</tr>
<tr>
<td>Organizations using internal methodology not based on a commercial one</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2: Methodology Usage (from Fitzgerald 2000)

The differences between the two studies may result from the different questions asked in the surveys. Ward and colleagues (1996) inquired about the benefits IT/IS organizations received from their technology investments, whereas Fitzgerald’s (1997) inquiry focused on the usage and benefits from commercially-available methodologies. Fitzgerald also observed software methodologies were extensively modified, leading Fitzgerald to distinguish between the “formalized method” and “the method in action”.

Supplementing the academic studies are industry-based surveys such Version One’s annual Agile Adoption survey (VersionOne, 2009), which showed 84% of respondents used Agile to
some degree. The intensity of Agile adoption was low, with 54% of the responding companies reporting that only five or fewer teams followed Agile practices. The survey did not correlate reporting company size to the number of teams following Agile practices. The survey is a web survey of some 2570 individuals from 88 countries. Surveys such as this one present a number of validity issues, not the least of which is that they are self-reporting and voluntary; we simply do not know what the response rate is because we are uncertain about the population from which the self-reporting sample is taken.

The few empirical studies of software methodology usage available support our anecdotal assertion that software methodologies are either not used or not followed rigorously. Nandhakumar and Avison’s (1999) case study of a large IT firm offers an explanation for why software developers may not be inclined to use a software methodology. Using participant observation of a large UK company, Nandhakumar and Avison suggested “Formal methodologies are too mechanistic to be of much use in the detailed, day-to-day organization of developers’ activities”. Furthermore, they asserted that “methods were a necessary fiction to present an image of control or to provide a symbolic status possibly at the risk of degrading developer productivity”.

Huisman and Iivari’s (2006) study of South African software managers and developers supports Nandhakumar and Avison’s (1999) assertion. Their industry survey revealed software managers’ perceptions of software development methodologies are more positive than those of software developers. From their results, Huisman and Iivari concluded “systems development
methodologies reflect management's agenda” and the low level of support provided by systems development methodologies is a likely cause for the low rate of software methodology usage.

2.3 Better Methodologies, Better Software?

It would be simple to dismiss software process improvement as simply an academic ideal that does not contribute value to real practitioners; however, there are many reports describing practitioner’s experiences with the benefits of a software improvement process, where the adoption and the disciplined use of software methodologies is a key practice. Diaz and Sligo (1997) reported Motorola Government Electronics’ Division adopted CMM, which resulted in a nearly eight-fold decrease in defects and a 280% improvement in productivity. Dion (1993) calculated the software process improvement effort at Raytheon Equipment division returned $7.70 for every dollar invested, and yielded a two-fold increase in productivity. Basili and Green (1994) described how a software improvement effort at NASA’s Flight Dynamic Division resulted in a 75% reduction in defects and a 50% reduction in project costs over eight years. Galin and Avrahami (2006) reviewed 19 papers reporting experiences of professionals involved in over 400 projects, and reported that software process improvement programs improved performance consistently across seven different project metrics.

In a major IT firm, Harter, Krishnan, and Slaughter (2000) empirically investigated the relationship between process maturity, quality, cycle time, and effort for the development of 30 software products that were part of a materials requirements planning (MRP) system. With collected data, they validated their model relating cycle time, effort, and product quality to
process maturity. Their study revealed a virtuous relationship between quality and development efforts, where a 1% improvement in process maturity resulted in a 0.611% decrease in development effort, a 0.454% decrease in cycle time, and a 1.589% increase in product quality. Agrawal and Chari (2007) studied 37 high maturity projects (CMM level 5) to find the determinants for their effort, cycle time, and quality. Using linear regression, Agrawal and Chari found project size was the only significant determinant of time, effort, and product quality for high maturity projects. From the result of their study, we can infer successful software process improvement efforts reduce the sources of variation in the process to only the size of the project.

Although these papers report the benefits of organizations reaching higher levels of process maturity, there are those who question the benefits of the maturity approach to software process improvement. Jeff Voas’ (1997) commentary on the obsession with software process improvement metaphorically compares the assumption that quality software processes results in quality software to assuming that clean pipes can only produce clean water. While it is certainly difficult for dirty pipes to yield clean water, he argues there is no link between a quality software process and quality software. Pour dirty water into clean pipes, and it is still dirty water that pours out.

Furthermore, while software methodologies are an intrinsic part of a software process improvement effort, it is unclear from these papers the contribution the use of software methodologies made towards the results reported. Sawyer and Guinan’s (1998) study of 40 teams at a commercial software development house reported “the use of software methods and

---

7 During a panel on software process improvement, a colleague once suggested “you can make lead life jackets and still be ISO-9000 certified”
automated development tools provide no explanation for variance in either software product quality or team performance” (p. 560). Cockburn’s (2003) reflection study on people and methodologies in software development also down-played the importance of software development methodologies. Cockburn posed the question “do software methods matter at all?”. In his analysis of 23 industry articles chronicling his observations over a number of projects, one of the conclusions Cockburn drew was that “people are a first order non linear component in software development” leading him to suggest methodologies “the working conventions people choose to follow” (p. 68).

Project data collected by Jones (2000), and Boehm (1984), are supportive of Cockburn’s assessment. Jones’ data reveals high management and staff experience contributed 120% to productivity while effective methods/process contributed only 35%. Staff *inexperience* degrades productivity -177%, while ineffective methods/processes degrade productivity only -41%. Boehm’s results (2000), reflected in the COCOMO (Constructive Cost Model) estimation model, place personnel/team capabilities as the most significant cost drivers, affecting productivity by 353%, and process maturity as the twelfth driver, affecting productivity only 143%.

Dybå (2005) performed a quantitative survey of 120 software organizations to understand the key factors for software improvement success. His study led him to conclude “rather than trying to imitate technical procedures, software organizations should focus their SPI efforts on creating an organizational culture within which these procedures can thrive. This differs substantially from that found in most of the existing SPI literature, which focuses almost entirely on software engineering tools and techniques” (p. 421).
The reluctance to adopt formal methodologies has, in the past, been written off as ignorance on the part of the developers, slow technology transfer, or even sloth (Stelzer & Mellis, 1998). It is truly difficult to believe that, as a whole, well-educated and experienced software developers are simply ignorant or even lazy. Fitzgerald’s (1997) study indicates this is not the case because those who reported not using methodologies had a high degree of knowledge about them. His findings imply that software developers made an explicit decision not to use methodologies because they did not see adoption of a formal methodology as valuable. Fitzgerald’s study raises questions about the motivations for adopting software methodologies.

Several software engineering researchers (Davis, 1985; Hardgrave & Johnson, 2003; Khalifa & Verner, 2000; Riemenschneider, Hardgrave, & Davis, 2002) have constructed or used social psychology models derived from of the Theory of Reasoned Action (TRA) (Ajzen & Fishbein, 1980) to understand the factors driving technology adoption. TRA predicts and explains action by relating an individual’s intention to take an action to his or her attitude towards a behavior, the subjective norm; that is, an individual’s perception of what his or her peers think he or she should or should not do.

Davis (1985) introduced a derivative of TRA in the form of the Technology Adoption Model (TAM), specifically adapted for modeling the acceptance of information systems. TAM replaced TRA’s attitude construct with two technology acceptance measures: Perceived Usefulness, and Perceived Ease of Use. Davis later compared the explanatory power of TRA and TAM in a
study of user acceptance of computer technology (Davis, Bagozzi, & Warshaw, 1989). The models were compared by studying 107 MBA students’ use of a word processing package. In this study, Perceived Usefulness strongly influenced peoples' intentions to adopt technology, explaining more than half of the variance in their intentions to use the word processor. Perceived Ease of Use had a small but significant effect on intentions, while Attitudes only partially mediated the effects of these beliefs on intentions, and subjective norms had no effect on intentions.

Khalifa and Verner (2000) modified Triandis’ (1979) extension of TRA to explain usage of software methodology rather than intention to use. Their model related facilitating conditions and developers’ beliefs with regards to the perceived consequences of using a specific software methodology in terms of both product quality and process quality. Khalifa and Verner used a questionnaire for 82 senior software developers from Australia and Hong Kong. Their results indicated that the extent and use of a software methodology was actually influenced by facilitating conditions and developers’ beliefs concerning the effect of the software development methodology on the quality of the software development process and team size.

Reimenschnieder and colleagues’ (Riemenschneider et al., 2002) survey of 128 developers from Fortune 1000 companies, after the introduction of a new software methodology, compared five theoretical models including TAM, TAM2, and the Theory of Planned Behavior (TPB) (Ajzen, 1985). The authors discovered that, if a methodology was not regarded as useful by developers, then its prospects for successful deployment were severely undermined. The intention to use a methodology was driven by three factors: organizational mandate, compatibility with how the
developers worked, and the opinion of peers towards using the methodology. A criticism of Reimenschnieder and colleagues’ work is that, although it focuses on the elements of software methodologies that are regarded by developers as useful, they have not sought perspectives of developers on what makes a software methodology useful.

Hardgrave and Johnson (2003) combined TAM and TPB to explain acceptance of Object Oriented Systems Development (OOSD) methodologies by individual developers. Based on responses from 150 developers, they claimed their model explained 60% of a developer’s acceptance of OOSD. Their constructs included the perceived usefulness of OOSD to the organization, the perceived personal ease or difficulty of performing OOSD, and the perceived influence (subjective norm) of others over the use of OOSD.

The theme emerging from these adoption models is that the intention to use a software methodology is driven both by developers’ perceptions of a methodology’s usefulness, and their perceptions of how their peers perceive a methodology’s usefulness. Low levels of adoption and low levels of conformity imply that developers do not find software methodologies useful. If software methodologies do not serve their needs then “what are the needs of those developing software?”

2.5 The Social Nature of Software Development

A partial answer to the question “What are the needs of software developers?” may be found in Curtis, Krasner, and Iscoe’s (1988) qualitative field study of software development. Their results
suggested software development is a process that includes learning, technical communications, requirements negotiations, and customer interaction. They characterized software development as a learning process that is not served well by methodologies that only view software development processes as a series of development tasks.

Sawyer and Guinan’s (1998) study also highlighted the social nature of software development. Although they did not find a relationship between tools, techniques, and team performance, their study reported that social processes, such as informal and formal coordination, inter-team conflict resolution, and high levels of supportiveness, explained 69% of the self-reported variation in software team performance and 20% of the variance in quality of team work products. This led them to suggest “the formal coordination aspects of a methodology are valuable since they provide for an occasion to socialize.... [It is] the process of socializing at these meetings that provides the value, not the occurrence of the meeting” (p. 562).

The influence of social processes on software team performance was highlighted by Faraj and Sproull’s (2000) study of the coordination of expertise, which demonstrated a strong relationship between coordination of expertise and team performance. The study collected data from 69 teams using a survey instrument at a division of a large high tech company, and their analysis of the data demonstrated that the mere presence of expertise was not sufficient to affect performance if team members could not coordinate that expertise. Faraj and Sproull distinguished between the coordination of expertise and administrative coordination. Administrative coordination is effective for simple routine tasks such as assignment of tasks and resource allocations, but is insufficient for complex non-routine tasks, which required knowledge
and skill. Formal software methodologies embody administrative coordination and, while a necessary part of software development, administrative coordination is not sufficient because it is the social processes that enable coordination of expertise.

The Agile movement was launched by a manifesto with a first declaration that states “Individuals and interactions over processes and tools” (AgileAlliance, 2001a). The manifesto opened opportunities for researchers studying software methodologies to investigate the social dynamics of software development. There is considerable research into the use of ‘Extreme Programming’ (Beck, 2000). Many early studies were like Abrahamsson’s (2003), which evaluated the utility of what seemed like a radical departure from conventional software methodologies. Not only did Abrahamsson’s study demonstrate Agile methodologies could improve software team performance, but it also implied these improvements resulted from improved team social practices such as the XP planning game and the practice of pair programming.

Several studies have investigated how specific agile social practices and roles contribute to team performance. One example of such studies is by Williams and colleagues (Williams, McDowell, Nagappan, Fernald, & Werner, 2003), who experimented with the XP practice of pair programming as a learning technique in a post-secondary institution. While their experiment demonstrated paired students achieved higher exam and project scores, they also demonstrated paired students developed a positive attitude toward collaboration. Martin, Biddle, and Noble, (2004), conducted an ethnographic study of the role of the XP on-site customer, and found eight practices used by successful XP teams. Their study also observed the on-site customer role was unsustainable because of the demands placed on the on-site customer.
In contrast to the studies of specific XP roles and practices and their effectiveness, other researchers have studied how XP practices have holistically shaped social interaction. Robinson and Sharp (2005) examined the interaction between organizational culture and XP practices. In their ethnographic study, they observed the social interactions of three XP teams at different software development organizations. They concluded the “technical practices of XP have social interactions of consequence for the practices themselves” (p. 107). Chong (2005) took a different approach to that of Robinson and Sharpe and compared the social interactions of two software development teams using different software methodologies: one using XP, and one not using XP. Chong reported that the XP team had higher levels of consistency and standardization in their work patterns, and a higher degree of transparency than the non-XP team. Chong argued that greater standardization and transparency in teams allowed software development to become more routine, recognizable, and accessible. Whitworth and Biddle (2007) employed Grounded Theory to explore how agile practices mediated the relationship between the individual and the team. Their study provided a window into the experience of being a member of an agile software development team. They observed that Agile teams appeared more cohesive than non-agile teams. Dybå and Dingsor (2008) reviewed 36 empirical studies of Agile software development, seven of which considered the role of human and social factors. These studies demonstrate that Agile practices can improve team social practices and, as a result, software development performance. Collectively, these studies offer insight into how Agile software development practices – and specifically XP practices – contribute to create a positive social dynamic, and to the argument of the importance of social practices in software development.
None of these studies generated theory, or conceptualized practices such that they could be used to inform the design of software methodologies. None of these studies generated a theory explaining how people manage the process of software development. Such a theory could serve as a framework for explaining the social practices of software development, and could inform the design of software methodologies. As stated previously, mature disciplines have a well-established base of theories (Hall, et al., 2009), and numerous researchers (Hannay et al., 2007; Sjøberg, Dybå, Anda, & Hannay, 2008) have expressed the need for the creation and use of theory in software engineering. Therefore, the motivation behind this study is to find a theory that explains how people manage the process of software development.
Chapter 3

METHODOLOGY

Software engineering research should move the discipline of software engineering away from simple advocacy and closer to an evidence-based discipline (Sheppard, 2003). For this, I need to state what I believe constitutes knowledge, and explain how I will obtain that knowledge. This section describes the study design and how I executed the design. In sub-section 3.1, I first describe our philosophical stance with respect to the well-known knowledge paradigms used in software engineering (and information systems) research; this establishes the context for our subsequent research design. In sub-section 3.2, I describe the grounded theory method and the design of this study, and sub-section 3.3 describes my execution of the study design. Finally, in sub-section 3.4, I address the traditional issues of threats to validity and study limitations.

3.1 Philosophical Stance

Research is the pursuit of knowledge, and Plato defined knowledge as “justified true belief”. My philosophical stance is my statement of assumptions regarding what I believe constitutes evidence and what I must do to justify the evidence I present in this study as knowledge. Lincoln and Guba (2005) present four paradigms for qualitative research: positivism, post-positivism, critical theory, and constructivism. My adopted philosophical stance for this study is to follow the interpretivist paradigm and much of my explanation of my philosophical stance is drawn from Orlikowski and Baroudi’s (1991) treatment of the topic. In this sub-section, I will
explain the more traditional positivist paradigm and contrast it with the chosen interpretivist paradigm.

3.1.1 The Positivist Paradigm

Engineering prides itself on its rigorous application of the natural sciences, and much of engineering research is based on a “positivist” paradigm of knowledge where the ontological assumption is that there is one true reality which can be objectively apprehended or known independently of the researcher (dualism). The positivist epistemological stance is that reality can be objectively comprehended, which makes objectivity paramount and therefore explicitly excludes values. Positivist research methodologies rely on formal propositions, quantifiable measures of variables, and hypothesis testing where results can be independently reproduced and verified. Knowledge results from a verifiable hypothesis that that can be shown to be true regardless of the researcher, the researcher’s values, and the occasion. The positivist’s paradigm serves the natural sciences well because natural laws such as gravity have little respect for my values. Gravity operates whether I like it or not.

According to Orlikowski and Baroudi (1991), in the positivist paradigm the role of the researcher is “...to ‘discover’ the objective physical and social reality by crafting precise measures that will detect and gauge those dimensions of reality that interest the researcher. Understanding phenomena is thus primarily a problem of modeling and measurement, of constructing an appropriate set of constructs and an accurate set of instruments to capture the essence of the

8 Civil engineers burst with pride about their discipline when they tell stories such as the one about the top of the towers of the Verrazano Narrows suspension bridge in New York being 41.2mm further apart than their base because of the Earth’s curvature.
phenomenon. It is assumed, explicitly or implicitly, that there is a one-to-one correspondence between the constructs of a researcher's model and the events, objects, or features of interest in the world” (p. 9).

Using the positivist paradigm to study social processes implies that we are choosing to describe the social world in terms of natural laws where knowledge can be acquired through the collection of value-free facts (Nandhakumar & Jones, 1997, p. 110). This works well if we are interested in discovering the result of a specific intervention to answer a research question such as “Are pair programming teams more productive than individual programmers?” However, as Orlikowski and Baroudi point out, “the positivist aim to explain and predict external reality implies that people are not active makers of their physical and social reality. Such a posture is not conducive to the discovery and understanding of non-deterministic and reciprocal relationships” (Orlikowski & Baroudi, 1991, p. 12). The purpose of this study is to discover how people manage the process of software development and if we accept that organizations, groups, and social systems do not exist apart from humans then we need a different approach to understand these systems.

### 3.1.2 The Interpretivist Paradigm

In contrast to the positivist paradigm, the interpretive paradigm takes the position that our knowledge of reality is a social construction. The aim of all interpretive research is to understand how members of a social group, through their participation in social processes, “enact their particular realities and endow them with meaning, and to show how the meanings, beliefs, and intentions of the members help to constitute their social action” (Orlikowski &
Baroudi, 1991, p. 13). The interpretivist ontological assumption is that reality is a social construction and that “realities are apprehendable in the form of multiple intangible mental constructions socially and experientially based, local and specific in nature” (Guba & Lincoln, 1994, p. 110). The interpretivist epistemology holds that knowledge is obtained through the understanding of the social interaction by which the subjective meaning of reality is constructed (Walsham, 1995). The interpretivist methodology takes the position that the “variable and personal nature of social constructions suggests that individual constructions can be elicited and refined through interaction between and among investigator and respondents” (Guba & Lincoln, p. 111).

Understanding social reality requires understanding how practices and meanings are formed and informed by the language and tacit norms shared by humans working towards some shared goal (Orlikowski & Baroudi, 1991). Unlike the positivist approach, the interpretive researcher avoids imposing externally-defined categories on the social setting. Rather, the interpretivist researcher engages with the social setting, investigating how the interaction takes place from the participants’ perspective, and his or her constructs emerge from the in-depth examination of and exposure to the phenomenon of interest. The categories and themes that emerge out of this approach are “intended to closely resemble those relevant to the study's participants” (p. 14).

If reality is a social construction that is local and specific in nature then how can we use an interpretive study as a basis for informing policy and taking action? How do we prove anything? Interpretive studies do not prove or disprove hypotheses; rather, researchers try to understand phenomena through the meanings and values that people themselves assign to them, and to
produce a rich and detailed description of the phenomenon under investigation (Dawson et al., 2003). An interpretivist study is an effort to “distill a consensus construction that is more informed than any of the predecessor constructions” (Guba & Lincoln, 1994, p. 111). Interpretivist research helps us to answer the question “What is going on here?” and to create policy from a more informed position.

While interpretive studies provide the researcher with creative tools for answering questions of the form “What is going on here?”, this creativity should not come at the expense of analytical rigor. As Whittemore and colleagues (2001) put it, “Creativity without science diminishes the knowing associated with interpretive inquiry. Interpretive inquiry is not the sole creation of the investigator, as is the case with journalism and opinion. Rather, the results are the joint creation of inquirer and inquired in a given context at a given time” (Whittemore, Chase, & Mandle, 2001, p. 526). Section 3.2.10 explains the rigor in qualitative studies under the interpretivist paradigm that was applied in this study.

3.1.3 Personal Reflections

The interpretive researcher participates in the social creation of reality and cannot assume a value-neutral stance; prior assumptions, beliefs, values, and interests always intervene to shape their investigations (Orlikowski & Baroudi, 1991, p. 15). As such, the researcher, as well as his or her audience, needs to be aware of his values in the form of the experiences, biases, and reactions to the data that may shape his interpretation of the data.
Without going into an extensive autobiography, I am an experienced software developer, systems analyst, and project manager who has participated in the design and management of large software projects since starting in the industry in 1983. These projects included the development of call processing software for central office telephone switches, automated railway signaling systems, a workflow management system for a robotic image setter, and the design of a large European cellular telephone billing system. For the last 15 years, I have worked as a consultant, and I am currently employed by Rally Software as an “Agile coach”. Several of my personal friends are members of the so-called Agile “A list”.

I have a very strong belief that social dynamics are the dominant determinant of software product development productivity. Much of my consulting practice focuses on introducing technical practices that encourage strong social practices; however, I have also worked with teams that had strong social practices, and yet failed catastrophically.

At the start of this study, I was fascinated by the relationship of team dynamics and fast decision cycles and especially by the work of Colonel John Boyd as some of my early publishing reveals (Adolph, 2005; Adolph, 2006). Part of my thinking when I engaged in this study is that I would find a strong relationship between fast decision cycles and social dynamics.

I consider myself to be a practitioner more than an academic, and part of my motivation to conduct this research is that I personally believe that there is a significant gap between the software engineering research agenda and the practice of software development. Software engineering is supposedly an applied science which, for me, implies our research should be
accessible to motivated practitioners and applicable to the issues they are encountering. I doubt, for example, that many practitioners beyond the software engineering process group of large government contractors read *IEEE Transactions on Software Engineering*. Most software practitioners that I know who still read edited publications read *Better Software*, and most follow the online blogs produced by whomever they agree with. Very few read what I call refereed publications. I have a concern the pursuit of an excessive sense of rigor in our research and publication is limiting the applicability of software engineering research.

3.2 The Design of a Grounded Theory Study

A Grounded Theory is a set of integrated conceptual hypotheses systematically generated to produce a theory about a substantive area (Glaser & Holton, 2004). Glaser and Strauss (1967) proposed the method as having application for both qualitative and quantitative data. Co-discoverers Barney Glaser and Anselm Strauss called the method “grounded” because a theory is systematically obtained from a broad array of data through a rigorous process of constant comparison – it is ‘grounded’ in the data (Glaser, 1965; Glaser & Strauss, 1967). Grounded Theory differs from logico-deductive methods of inquiry because, rather than developing a theory without relying on data and then systematically seeking out evidence to verify the theoretical constructs, researchers using Grounded Theory set out to gather data and then systematically develop a mid-level substantive theory derived directly from the data (Glaser & Strauss, 1967, p. 4). The goal of Grounded Theory is to generate concepts and categories that account for a pattern of behavior which is relevant and problematic for those involved (Glaser, 1978, p. 78).
For many of us in the software engineering field, the term “theory” is a source of confusion with respect to Grounded Theory. Our strong backgrounds in physical sciences and mathematics tend to lead us to regard a theory as a universal truth, like Einstein’s Theory of Relativity. Theories do not have to be universal truths; they can vary in their generalizability. Grounded Theory generates a mid-level or substantive theory which describes processes such as those in social movements, organizations, or communities. A further challenge for engineers, who are more comfortable with an objective view of reality, is that when studying people we cannot ignore our personal values because personal values influence the ways in which people interpret reality. Grounded Theory takes into account personal values and exposure to concepts in particular disciplines through the concept of theoretical sensitivity (Glaser, 1978). However, in Grounded Theory, such pre-existing values and knowledge serve only as another source of data and are not privileged (Glaser, 1998)

3.2.1 Why Grounded Theory?

Grounded Theory is best for answering questions of the form “What is going on here?” (Schreiber & Stern, 2001). It is useful when we want to learn how people manage their lives in the context of a problematic situation, and for learning about the process of how people understand and deal with what is happening to them through time and changing circumstances. Grounded Theory is useful for research in areas that have not been previously studied or where a new perspective might be beneficial (Schreiber & Stern, 2001).
Grounded Theory would be an inappropriate form of inquiry for answering a question such as “Are pair programming teams more productive than individual programmers?” Grounded Theory is, however, appropriate if we are interested in explaining how pair programming unfolds by asking “How do individuals manage the process of pair programming?” The answer to such a question helps inform the creation of policy for managing pair programming. For the purposes of our study, low rates of software method adoption suggested a disconnection between how software methods recommend management of the software development process and the processes that people actually follow when managing it. We wanted to understand the central concern of those involved in the process, and how they resolved their central concern, with the intention of being able to inform software development policy.

### 3.2.2 Grounded Theory Overview

Grounded Theory is like agile software development in the sense that it is deceptively simple conceptually, yet rigorous and disciplined in practice. Figure 5 shows how we visualize the process of Grounded Theory. A researcher begins by collecting data on a phenomenon of interest, and analyzes the data by searching for patterns of incidents which indicate concepts that are aggregated into categories. The theoretical properties of a category are developed by comparing incidents in incoming data with previous incidents in the same category. During the analysis, the “core category” is developed. The core category is a category which captures the most variation in the data (Glaser, 1978) and addresses the main concern of the study participants. Throughout the process, the researcher writes memos, capturing his or her thoughts that support the emerging concepts, categories, and their relationships (Glaser, 1998). The process continues until the categories become “saturated”, that is, further collection of data does
not add any new properties to the existing categories – the incidents are said to be interchangeable. After saturation, the theory is compared to theories described in the literature. The literature search is deferred to late in data collection to avoid forcing pre-conceived theories onto the substantive theory being developed from the data (Glaser, 1998).

From this overview, one could regard producing a substantive theory as a straightforward, mechanical, rigorous method which enables the researcher to make easy progress; however, we found it messy, tedious, and difficult. Producing the substantive theory required creativity and
theoretical sensitivity on the part of the researcher to develop the theory. Theory development did not progress in a straightforward linear manner; rather, it involved many false leaps, blind alleys, and dead ends. Nonetheless, it was a very, very rewarding experience that enabled us to see the experience of software engineering from a very different perspective.

3.2.3 Which Version of Grounded Theory?

What confuses most novice Grounded Theory researchers is there are not one but at least three similar, yet distinct, research methods that claim the name Grounded Theory. The original method was described in *The Discovery of Grounded Theory* (Glaser & Strauss, 1967). Strauss later considered ways to formalize and proceduralize grounded theory and published *Basics of Qualitative Research* with Corbin in 1990 (Strauss & Corbin, 1990). Further adding to the potential confusion of novice researchers was Kathy Charmaz’s efforts to clarify some of the ontological and epistemological ambiguities underlying Grounded Theory with the publication of *Constructing Grounded Theory* (Charmaz, 2006).

There is a voluminous and sometimes shrill debate in the Grounded Theory community about which method(s) are true Grounded Theory. Most software engineering researchers using Grounded Theory tend to cite Strauss and Corbin (Strauss & Corbin, 1998), likely preferring Strauss and Corbin’s more structured and proceduralized approach to analysis. I chose the classical Glaserian approach for two pragmatic reasons: there was less likelihood of imposing pre-existing categories on the data (Glaser, 1992), and there were more resources available to me in terms of mentors, seminars, and websites based on the Glaserian approach. For example, our
3.2.4 The Grounded Theory Analysis Model

The appeal of Grounded Theory to the software engineering researcher is that Grounded Theory generates theory rather than just testing it. Grounded Theory also possesses a visceral appeal for software engineering researchers because theory is generated by following a rigorous method, much of which is derived from Paul Lazerfeld’s (Lazerfeld & Wagner, 1958) teaching of qualitative mathematics; Barney Glaser was one of his students.

3.2.4.1 Concept-Indicator Model

Grounded Theory is based on a concept-indicator model of constant comparisons of incidents (indicators) to incidents (indicators) as shown in Figure 6. Concepts are often behaviors, or factors affecting behaviors, which help to explain to the analyst how the basic problem of the actors is resolved or processed (Strauss, 1987, p. 33). As concepts are generated, the researcher begins comparing incidents (indicators) to the emerging concept, see Figure 7 (Glaser, 1978, p. 62). Indicators are actual data, such as behavioral actions, and events observed or described in documents and in interviews; they are often captured by the words of the participants (Strauss, 1987, p. 25). An indicator may be a word, a phrase, a sentence, or a paragraph in the data being analyzed. Glaser, in his usual mystic writing style, never quite definitely states what a concept is; however, for the purpose of our study, we loosely regarded a concept as a class which is suggested by indicators. We elaborate on this topic in section 3.2.4.2.
When reading Grounded Theory texts an impression can be created, particularly in Strauss and Corbin’s (1998) writing, that indicators represent minute data fragments, because of the frequent admonitions about line-by-line comparison and examples provided. In actuality, indicators are exactly what the term implies: they indicate some event or behavior which may be indicated by a word, a sentence, paragraph, or even a set of paragraphs. The following are some examples of indicators from our research:
<table>
<thead>
<tr>
<th>Code</th>
<th>Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brainstorming</td>
<td>“I do some brainstorming as to what they want or what they really want”</td>
</tr>
<tr>
<td>Tracing</td>
<td>“So, I spent quite a bit of time kind of tracing through code to try to figure out what each thing you know kind of did”</td>
</tr>
<tr>
<td>Experimenting</td>
<td>“...so they were experimenting with different ways to you know track issues, um, we also had problems with the branching code and tracking you know changes to multiple (unclear) code”</td>
</tr>
</tbody>
</table>

Table 3: Example Indicators and Codes

3.2.4.2 Concepts, Codes, Categories, and Properties

Concepts and categories are not well defined and it often seems that the terms are used interchangeably. Glaser and Strauss originally stated that a category “stands by itself as a conceptual element of the theory” (Glaser & Strauss, 1967). Later, Glaser said that a category is a “type of concept that is usually used for a higher level of abstraction” (Glaser, 1998, p. 38) and, in later writings, Glaser defined a concept as a “naming of an emergent social pattern grounded in the research data” (Glaser, 2001).

Using this definition, we chose to operationally model a concept as a Unified Modeling Language (UML) (Booch, Rumbaugh, & Jacobson, 1998) class with indicators suggesting concepts, and codes that were simply the indicator name. Concepts were given names that were evocative of the patterns of behavior suggested by the indicators. We further chose to regard categories as an aggregate of concepts, representing a higher level concept. Viewing concepts and categories through this model helped clarify our understanding of properties, which are referred to as “a conceptual aspect or element of a category” (Glaser & Strauss, 1967). We
chose to view properties from the UML perspective, as “attributes”. Glaser and Strauss often make references to sub-categories, which we thought of as a “whole-part” relationship between categories. Figure 8 is a low fidelity UML representation of the way we chose to reason about these relationships; our model was intended to help us reason about these fuzzily-defined relationships, not to be definitive:

![UML Model Indicators, Concepts, Categories](image)

**Figure 8: UML Model Indicators, Concepts, Categories**

Concepts are the building blocks of a Grounded Theory, and conceptualization is a distinguishing trait (Glaser, 2002). The two most important properties of conceptualization for generating Grounded Theory are that concepts are abstractions of time, place, and people, and that concepts have enduring “grab” in the sense that they are easily knowable and explainable (Glaser, 2001). To crystallize the importance of conceptualization, a favorite question of Glaser’s during a seminar discussion is “*How would you conceptualize that?*” If we cannot conceptualize, we can fail to see the common threads running between incidents and concepts.
that can be collapsed into categories. Only conceptualization results in a parsimonious and conceptually dense theory.

Conceptualization should not be a stranger to software engineers. Conceptualization is what we routinely do to create cohesive, loosely-coupled systems. We hopefully do not create completely separate system designs for a web interface, a Windows client, and a CRM system. Rather, we abstract and model all of these in terms of an external interface which is a conceptualization of the web interface, the Windows client, and the CRM system.

Naming a concept is sometimes problematic. While we tended to use in vivo inspired codes to capture incidents (see Table 3), the codes could not capture the full scope of the emerging concept. This is where Glaser (2002) again asserts the importance of the constant comparative method “The pattern is named by constantly trying to fit words to it to best capture its imageric meaning. This constant fitting leads to a best fit name of a pattern, to wit a category or a property of a category. Validity is achieved, after much fitting of words, when the chosen one best represents the pattern” (p. 315).

In our study, at a descriptive level we saw “tracing through the code library”, “spikes”, and “JAD sessions” as part of the concept of “Scouting”. But from the participants’ experiences, we viewed “Scouting” as a unifying pattern for an indeterminate activity undertaken to find answers in an environment of uncertainty and often under resource constraints. In all of these activities, participants were engaged in exploration to find answers, often encountering many blind alleys and dead ends. “Scouting” was developed as a concept that categorized items where the purpose
was to obtain information but where there was uncertainty about the resources and time required to obtain that information. All of these activities were characterized not only by the excitement of discovering new knowledge, but also by the frustration of managing “Scouting” in a resource-limited environment. Moving from the descriptive to the conceptual also helps create a dense parsimonious theory (Glaser, 1998).

3.2.4.3 Constant Comparison

Constant comparison is a systematic way to generate concepts from incidents. In the words of Grounded Theory co-discover Anselm Strauss (1987) “Initially, indicators are compared to indicators, where the analyst is forced to confront similarities, differences and degrees of consistency of meaning among the indicators. Through this laborious and often tedious process concepts are generated and indicators are then compared to the emerging concepts. This sharpens and clarifies the concept to achieve the best fit to the data. Meanwhile further properties of the category are generated until the codes are verified and saturated yielding nothing much new. In this model the concepts have “earned their way into the theory by systematic generation from data.” (p. 25)

Constant comparison has been another distinguishing trait of Grounded Theory because it is much more than simply a search for patterns in the data. It is a continuous process of generation and verification which helps purge the emerging theory of preconceived concepts because concepts must earn their way into the data. New data is constantly compared against previously observed data and developed concepts. Concepts that are not sustained by the data (single
indicators or weak patterns) soon drop out. If followed rigorously, constant comparison makes Grounded Theory self-correcting, a feature that cushioned us more than a few times.

Comparison of the indicators from Table 3, specifically “Brainstorming”, “Tracing”, and “Experimenting”, gave rise to the concept of “Scouting”, an activity to discover a path through unknown or unfamiliar territory. This is illustrated in Figure 9. Further comparison began to develop dimensions for this concept; for example, “Brainstorming” and “Experimenting” versus “Tracing”: the purpose of “Brainstorming” and “Experimenting” is to generate candidate solutions, whereas “Tracing” is forensic, trying to discover how something works and what side-effects a solution may have. This gives a qualitative range for the purpose of “Scouting”, from “forensic” to “generative” (Figure 10).

![Figure 9: Constant Comparison and Concept Emergence](image.png)
As the concept begins to emerge, subsequent indicators are compared to the concept. For example:

| Workshopping | “during those workshops we don’t reduce any-any scope or anything like that. So it’s more of a like—okay we want to do this so what would it involve—to do this” |

When we compared “Workshopping” to the existing “Scouting” concept, it helped us develop the category by adding another dimension to it, “plurality”: is “Scouting” done by individuals as in “Tracing”, or is it conducted as part of group exercise such as “Brainstorming” or “Workshopping” (Figure 11)?
Subsequent indictors further dimensionalized the “Scouting” concept to include:

- Time Continuity – whether scouting took place as a short term continuous effort or as a frequently suspended long term effort.
- Ownership – who declared the scouting effort complete, the scouter or someone else.

### 3.2.4.4 Saturation and Interchangeable Indicators

In comparing incident to incident, when coding data the researcher begins to see a pattern and to develop concepts that fit it (Glaser, 1998). In continuously comparing further incidents to concepts, the researcher starts seeing the same pattern over and over again in different ways (Glaser, 1998). Saturation occurs when, as more data are collected and analyzed, nothing new is added to a category. Saturation occurs because the incidents are interchangeable; indicators for events and behaviors may vary in detail or be repetitious, but add up to the same thing analytically, and further collection of data does not further elaborate the category.
We discovered that categories tended to saturate very quickly. With less than five interviews, we started only seeing minor variations in the indicators for “Scouting”. Codes such as “JAD Session”, “Walk-Through”, and “Code Read” all seemed to be variations of existing indicators and, while they strengthened our confidence in the category, additional collection of data added nothing new.

3.2.5 Theoretical Sampling

In any research project, sampling strategies and the adequacy of the selected sample affect the validity of the study. For a Grounded Theory study, the population under study is the set of indicators that constitute the phenomena rather than individuals experiencing the phenomena. Therefore, the sampling strategy must make sufficient indicators available to develop a conceptually dense substantive theory. Grounded Theory employs theoretical sampling where the analyst “jointly collects, codes, and analyzes his data and decides what data to collect next and where to find them in order to develop his theory as it emerges” (Glaser, 1978, p. 36). In other words, the analytical process and concept development direct the investigator to find further incidents to develop properties and dimensions.

Theoretical sampling is different from selective sampling, which uses a pre-determined selection process. By sampling individuals, a researcher has the potential to repeat samples of the same indicators, without adequately sampling the indicators in the problem space to produce a grounded theory. Theoretical sampling is driven by emerging categories and hypotheses, and is therefore an ongoing process that cannot be pre-determined. It is a critical practice supporting the development of an emerging theory that is tightly intertwined and supportive of iterative and
concurrent collection of data. “By responding to the need for theoretical completeness, theoretical sampling directs the researcher to new data sources which constantly generate the parsimony and scope of the theory as it accounts for how the main concern is constantly resolved” (Glaser, 1998, p. 158). According to Glaser (1998) “taking a random sample of a set number of people does not make sense in grounded theory” (p. 159).

3.2.6 Theoretical Sensitivity

Glaser (1978) in his usual elaborate and mystic style describes theoretical sensitivity as the researcher’s insight to the research situation. It is often referred to as a creative aspect of Grounded Theory because it involves the researcher acknowledging experience and expertise developed in the field of study. The Grounded Theory researcher requires creativity to “see” the recurring patterns of behavior and events because his or her reflections about what is happening in the data drives theory development. Theoretical sensitivity relates to the choice and appropriateness of imagery evoked by concepts and their names. A Grounded Theory researcher must develop theoretical insight and make something of that insight (Glaser & Holton, 2004) but theoretical sensitivity is just one more source of data, which cannot be privileged over other sources. A researcher cannot impose their preconceived ideas and biases on the emerging theory.

Theoretical sensitivity makes Grounded Theory research very much a solitary effort. The insight and perspective of the individual researcher is what gives insight into the data. We are aware of instances where individuals have collaborated to develop a Grounded Theory. Often these groups have resorted to pre-defining codes such that all individuals code data in the same way. This is forcing pre-conceived concepts on the data.
3.2.7  Memoing

Memos capture, make real, and preserve the ideas emerging from an analyst’s pre-conscious processing as the data is analyzed (Glaser, 1998). Whenever an idea emerges about a concept, or about the connections between concepts, analysts should stop coding and write a memo because memoing is about ideas the analyst is developing from the data. Ideas are fleeting and fragile, whereas the data will always be there. Although typically based on description, memos help raise description to a theoretical level through the conceptual rendering of the material. The writing of theoretical memos is the core stage in the process of generating theory. If an analyst skips memoing then he/she is not doing Grounded Theory (Glaser, 1978).

In this study, I took Glaser’s advice on memoing “to the max” and wrote memos whenever an idea “popped” into my head. During early open coding, I wrote a large number of what were rather speculative memos. An arbitrary word or phrase in a transcript could trigger an uncritical “aha moment”, and this large number of memos produced concepts that were not grounded in the data and even some that were flights of fancy. This is a healthy process because, as long as I remembered that concepts must earn their way into the theory through constant comparison, such a process explicated my biases. My memos fell into three broad categories:

1. Concept Memos: As concepts were developed during constant comparison, the memos captured the concepts. Initially, concept memos simply served as a glossary, but as I became more mature in the process, the memos became much richer, describing concepts in dimensional terms, and in terms of their properties and relationships to other concepts. Our theory is built from concept memos.
2. Process Memos: I kept extensive process memos that served as an audit trail of our activities and decisions. Much of this section is derived from my process memos.

3. Speculative Memos: I created speculative memos when I was inspired by an idea. These memos may not make a contribution to the theory, but they captured interesting insights that emerged from the data and revealed my biases.

Memos are not pretty things. They are written in the passion of the moment; they can be short “jots” or long rambling prose, and they are full of misspellings and grammatical errors. They are a tool for the analyst to raise description to the theoretical level through conceptual rendering, and not for immediate sharing of ideas (Glaser, 1998). While memos are written ad hoc, they are not undisciplined. There is a need to use some caution to avoid smuggling extant theory in through the back door (Glaser, 1998, p. 181). In one case, I used Herbert Simon’s concept of satisficing (Simon, 1947) as short hand to characterize how people manage the process of software development by trying to find a reasonable solution without wasting resources to find the best solution. The concept of satisficing does not belong in the emerging theory because I would otherwise be imposing a concept from an existing theory onto the substantive theory. In the pure sense, a Grounded Theory stands on the data on which it is grounded, rather than being derived from an existing theory. There is room for insights and reflections about the theory in the discussion sections of papers and dissertations.

3.2.8 Sorting and Writing

Analysis develops concepts and the relationships between those concepts, captured and expressed in memos, which inform the theory (Glaser, 1998). The analyst sorts the memos to
organize them into the outline of a theory for writing and presentation. There are no pre-conceived outlines because the outline is generated through the sorting of the categories and properties in the memos into similarities, connections, and conceptual orderings (Glaser & Holton, 2004).

The classical approach to sorting is to dump written memos on a dining room table and manually arrange them topologically, looking for connections, and the strength of those connections. I entered my memos into a Wiki, and used the hierarchy or tree view in the Wiki as the “table top”; we then simply moved memos up and down the memo hierarchy to arrange them.

3.2.9 Literature Search

Glaser (1978) argued that extensive reading in the substantive area, in the form of a literature review, could unduly influence analysis by forcing extant theoretical concepts onto the collection and analysis of data. Thus, to undertake an extensive review of the literature before developing a core category violates a basic premise of Grounded Theory – that theory emerges from data, not from extant theory. An extensive literature review could also run the risk of clouding the researcher’s ability to remain open to developing a completely new core category that has not figured prominently in the research to date, thereby thwarting theoretical sensitivity (Glaser & Holton, 2004). On the other hand, researchers, and particularly doctoral students, must situate their work in the context of the extant literature; Glaser’s position cannot be used to avoid important claims of the significance of the work and its potential contribution.

__________________________________________

9 Any researcher with a toddler (or a cat) may not find this method practical.
Glaser’s (1998) exhortations to delay the literature review should not be viewed as an outright ban on reviewing the literature. Traditional logico-deductive research starts with existing theory with the intention of affirming or denying a theory, or extending an existing theory. Accurately placing the proposed research within the existing knowledge landscape does not preclude looking at a topic differently or exploring an area where the theoretical landscape is not well developed. The literature search could arguably be used to increase the theoretical sensitivity of the analyst. We therefore interpreted Glaser’s (1998) argument as one against prematurely introducing extant theory.

Our literature search took place in two phases. The first was a “framing phase” that identified the general space we were interested in exploring in terms of the problem, and why the problem is valuable. Our search identified empirical software engineering research papers that explicated where researchers had gone before, and methodology papers that explicated how Grounded Theory was being employed in the software engineering community. The second, more traditional, phase of the literature search was conducted after the theory had emerged and stabilized, and pulled in extant theory to compare and contrast with our theory.

### 3.2.10 Grounded Theory Rigor and Quality

Qualitative research is often viewed with discomfort in software engineering because of misunderstandings regarding validation and verification. We have observed several situations in which Grounded Theory researchers were asked to “validate” their theory using quantitative mechanisms. This demonstrates a misunderstanding of Grounded Theory because unless the researchers bypassed the constant comparative method and fabricated their concepts and
relationships then the theory is grounded in the data and is a valid explanation of how people solve their problems.

Positivism’s criteria for research rigor and quality, with which software engineering researchers may be familiar, do not fit with qualitative tenets. Just as we are trying to answer a different type of question with interpretive approaches, we need to follow different criteria for rigor. For a grounded theory we need to adopt different criteria for evaluating quality and rigor of which can be framed with two questions:

1. Does the theory accurately represent the data and was not forced on the data?
2. Does the theory have “grab”? (Glaser, 1978) Will people find it interesting and does it add to the known body of knowledge?

Glaser’s (1998) quality criteria of fit, workability, relevance, and modifiability for judging grounded theory is similar because in part because it focuses on determining if the theory fits the data and was not forced:

- Fit is another word for validity; does the core category adequately express the pattern in the data which it purports to conceptualize?
- Workability means do the categories and the way they are related through the hypotheses sufficiently account for how the main concern of the participants in a substantive area is continuously resolved?
- Relevance makes the research important because it deals with the main concerns of the participants involved. Relevance, like good concepts, evokes instant grab – capturing the attention.
• Modifiability is significant because theory is never right or wrong, it just gets modified as new data is compared to it.

• Parsimony and Scope: the theory accounts for much variation in the data with as few concepts as possible (Hall & Callery, 2001)

Glaser (1978) emphasizes these criteria are met because the theory is generated from the data (pg 236). If the researcher follows the constant comparative methods and lets “concepts earn their way into the theory” and does not force the data, then the emergent theory will fit, and have grab. A grounded theory that fits the data and has relevance will work because it parsimoniously integrates the fewest concepts that account for much of the variation in the data.

Any research process requires attention to transparency and keeping good, well organized notes and records about the research process. Can evidence of the effort and the reasoning process be presented? We employed several practices to demonstrate an audit trail:
Through our process memos, we maintained a research diary of our activities and thoughts during this project and decisions regarding choices during the analysis phase.

We reviewed transcription and analysis of interviews and with participants to incorporate their comments in the data analysis.

We kept our work auditable by maintaining rich, thick, and well organized descriptions – all searchable within our Wiki.

We kept descriptions of the field sites, participant observations episodes, and of interview subjects.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thick Description</td>
<td>Through our process memos, we maintained a research diary of our activities and thoughts during this project and decisions regarding choices during the analysis phase.</td>
</tr>
<tr>
<td>Member Checks</td>
<td>We reviewed transcription and analysis of interviews and with participants to incorporate their comments in the data analysis.</td>
</tr>
<tr>
<td>Auditable</td>
<td>We kept our work auditable by maintaining rich, thick, and well organized descriptions – all searchable within our Wiki.</td>
</tr>
<tr>
<td>Applicability</td>
<td>We kept descriptions of the field sites, participant observations episodes, and of interview subjects.</td>
</tr>
</tbody>
</table>

Table 4: Strategies Executed in this Study to Create an Audit Trail

The next section describes the execution of our study design and provides evidence we did not force concepts on our data and therefore *Reconciling Perspectives* as a substantive theory fits the data, is relevant, works and is modifiable.

### 3.3 The Study

#### 3.3.1 The Inception of the Project

I had never done sociological research before, so I followed Fred Brooks famous words to “*be prepared to throw one away*”. I actually threw two studies away: the First Report (Adolph, Hall, & Kruchten, 2008), and an unpublished field study. Both of these early studies were small and
provided the opportunity to learn, fail safely, and obtain feedback to hone my skills as a sociological researcher.

### 3.3.2 The Research Problem

Grounded Theory is a method that permits researchers to determine an actual problem that exists for the participants in a substantive area, rather than what professional researchers may believe is the participant’s problem (Glaser & Strauss, 1967). In a Grounded Theory study, the researcher works with a general area of interest rather than with a specific problem (McCallin, 2003). This does not mean that there is no specific problem, but as Glaser (1998) writes “This problem and its processing will emerge in the initial stages of the research. And it will emerge if it is not derailed by what the researcher thinks is relevant beforehand and forces it on the study” (p. 116). In commenting on problem emergence in his own study ‘awareness into dying’ (Glaser & Strauss, 1965), Glaser (1998) stated that “we had no idea that awareness was a problem that the medical team had to handle constantly and that they did so by generating awareness contexts” (p. 119).

For this study, I worked with the very broadly defined problem statement of “How do people manage the process of software development?” While I have some preconceived ideas about potential problems (such as coordination, communication, adaptation), I left the problem statement open so that I could determine if my preconceived problems mattered to other software developers. Had I gone into the field with a rigorously defined research problem, I likely would never have learned that “Reconciling Perspectives” was a way people “Got the Job Done”.

69
3.3.3 Ethics

Most universities have research ethics boards which review and approve all research involving human subjects. While the potential for making grievous indiscretions when undertaking human psychological research makes it clear why such controls are necessary, those of us engaged in software engineering research may regard the ethics process as an annoyance, or as another bureaucratic hoop we must jump through. In general, software developers are not an underprivileged or easily abused minority; however, Singer and Vinson (2002) explain why an ethics review process is necessary even for software engineering research. This research was approved by UBC Behavioural Research Ethics Board: certificate H08-2387. The contact letter and study consent form are in appendices B and C.

3.3.4 Tools

Contrary to Glaser’s (1998) advice not to record interviews, I digitally recorded all of the interviews, and the recordings were later transcribed. I do not have the quality of memory required to reconstruct an interview strictly from quick jotting of field notes. This did cause me some grief later, when I literally interpreted the “line by line” coding instruction, because I had a large volume of data. Nonetheless, I still believe, despite Glaser’s recommendation, that recording and transcribing interviews is the most effective way to capture interview data.

While I did not use specific qualitative research tools, I did make use of computer support for managing my data. All interviews and field notes were transcribed into MS Word and were then stored in a Wiki, and I used the MS Word “comment” feature to tag incidents and write short
memos. As concepts emerged, memos describing them were written up in the Wiki and then manually linked to the source document(s). Appendix E is an example of a coded page.

I trialed several tools for our First Report and field study, and found the tools limiting at best because they tended to distance me from the data. Manually transcribing and linking the data between the MS Word documents and the Wiki, while tedious, enabled me to immerse myself in the data and paced my progress. Tools can create the illusion that more is somehow better because the tools make it appear easier to quickly handle larger volumes of data, while distancing one from that data. This can result in a limited or cursory analysis of large volumes of data, and in equating quality of research with number of interviews.

3.3.5  Field Access

Field research means going into the field and immersing oneself in the environment (Emerson, Fretz, & Shaw, 1995). My dissertation supervisor and I are fortunate to have good personal connections with many software engineering organizations, which meant that I did not have to conduct any “ambush” interviews and I had the opportunity to be on site for extended periods of time. Mulhall (2003) raised some concern about selecting field sites simply because they are accessible to the researcher; however, when the choice is between ambush interviews and an opportunity for an extended engagement, then accessibility wins. In the selection of potential field sites, I also used judgment to obtain some variation in the types of organizations participating in the study. I was able to collect field data from three different organizations: an onsite customer support field office, a small product company, and a software research and development center for large multi-national software product vendor.
Once I had access to a site, I made a concerted effort to maintain a regular visitation schedule, usually once a week for local sites and once a month for out-of-town sites. There were times when I collected data faster than I could analyze it. To fully engage in concurrent data collection and analysis, taking a pause in the visitation schedules would have been appropriate; but once a researcher is out of sight, he is out of mind, especially for busy software development teams. I found that simply being on site, even if I were not explicitly engaged in data collection, helped build trust and familiarity, which made it safe for people to approach me and participate in the study. I was pleasantly surprised by the openness and candidness of many interviewees, as well as by the openness of the organizations.

In accordance with ethics guidelines, I officially sought the permission of the subject organizations to gain access to their site prior to commencing the study at their site. Appendix A is an example of the contact letter.

### 3.3.6 Study Scope

All organizations participating in this study were product development companies whose focus was creating a software product or services that was sold or leased to clients. This is in contrast to Information Technology groups where the focus is creating software that supports internal processes. All study participants were drawn from the engineering side of the organizations and no participants from the business side of the organization participated in this study.
3.3.7 Site Description

Although most teams participating in this study claimed to follow Scrum (Schwaber & Beedle, 2002), the reality was that all were operating in a mixed-methods environment. The Site 1 team was embedded within a non-Agile organization. The product development teams at Site 2 were using Scrum, while their product support counterparts tended to only follow elements of the Scrum method. The teams at Site 3 who participated in this study followed Scrum but also worked within the context of a formal stage-gated software life cycle. At the enterprise level, Site 3 was starting an Agile transition.

3.3.7.1 Site 1: Small Onsite Field Office

Located in the interior of British Columbia, this site was an embedded project office for a Scandinavian vendor of gaming software. The project office had anywhere from 5 to 9 people on site supporting a large gaming operator. Although there was a permanent local team, the size of the team varied with the arrival and departure of staff from the Scandinavian offices. Developers at Site 1 both created new software, and maintained existing software created by their company. While they were physically located at the gaming operator’s site, they were distinct from the gaming operator. Their culture was also distinct; they represented a fast moving software development and maintenance organization hidden away in the basement of a very large and bureaucratic quasi-government gaming operator. The gaming operator’s office was plush, large, bright, and uniform. In contrast, the project office was one room in an underused hallway with yellowing linoleum on the floor. The project office was more reminiscent of someone’s old recreation room or a church basement than a software development center.
The office was windowless and desks were organized around the walls of the room. Unlike the gaming operator, the desks were not uniform and were older style office desks, some with peeling laminate. When they held their daily scrum meetings, the team simply turned their chairs and faced each other. At one end of the office was a foos ball table and dart board. The team had a tradition of going for coffee and doughnuts at 10:30 every morning. Figure 12 is a sketch from my field notes of the Site 1 office.

Figure 12: Site 1 Office Layout
The staff were young, all employees, and for most of them this was their first real job. Every few weeks or so, a couple of engineers from their head office would rotate into their office and work with them for a few weeks. The project manager was an experienced “renaissance woman” with a very strong personality who was very protective of “her people”. A strong characteristic of this team is that they took great pride in distinguishing themselves from the operating company, seeing themselves as the “can do” people.

I must disclose that I knew the project manager prior to this engagement and became personal friends with the project manager as this study unfolded. Data collection was abruptly terminated when the gaming organization encountered an event that resulted in conflict between the gaming organization and the field office of the parent company.

3.3.7.2 Site 2: Moderate Size E-commerce Vendor

Located in Vancouver, British Columbia, this home grown company was in a rapid growth phase, increasing their staff from less than 30 people to over 150 people in than 3 years. During this time, they had gone from being a specialized niche player to challenging major vendors such as Dynamo ATG and IBM. At the time of this study, the company’s software development group consisted of approximately 70 people, in addition to a few temporary contractors. The software group was organized into two sub-groups: product development (PD), and product services (PS). Product development reported to the Chief Technology Officer (CTO) and was responsible for ongoing development of the product. Product services reported to the Chief Operating Officer (COO) and customized and deployed the software for clients. The team was moderately young and very energetic. Many team members had eastern European ancestries, as
the company made significant use of an outsourcing house in Ukraine. Many of the senior leads were English, having been brought to Canada after their office in London was closed.

The office, located on two floors of a class “B” office tower, was bright, and had good quality modular furniture. The walls were covered with white boards; in fact, the walls were white boards – large information radiators covered with project schedules, system architecture diagrams, story boards, and marketing strategies. Teams sat together, with office space being arrange in “U” shaped pods that seated 9 people. Figure 13 is a sketch of the layout from my field notes.

![Figure 13: Site 2 Office Layout](image)

The PD team had been given a budget by their CTO to decorate their office, and all the PD pods were decorated with elaborate science fiction themes, from the flying saucers of the Invaders to
the X wing fighters from Star Wars. Walking into the PD area, one was greeted by a life-size replica of Bender, the kleptomaniac robot from Futurama. One group didn’t follow the sci-fi theme, created the illusion of a Caribbean resort in their pod. The PD teams had the frantic energetic feel of a stereotypical Silicon Valley startup, complete with surprise NERF\textsuperscript{10} Dart wars. In the middle of the office, the team was building a “soap box” racer for an upcoming charity event.

Unlike the PD team, the PS team was on a separate floor, and there were no flying saucers or X-wing fighters. The impression I had was that this environment was more stressful and serious. The desks were more conventional cubicles and the PS team seemed almost to be hard pressed for space. The desks were often empty because the PS team frequently worked at customer sites. One could almost imagine the PS team as the “forgotten”. The PS team smiled and laughed much less than then PD team, and I never observed a NERF Dart war breaking out in the cubicle farm of the PS team. The PS team was focused on rigid fixed deliverables set by customer demand, while the PD team had considerably more flexibility.

I must disclose that two years prior to using this site as a field research site, I had a business relationship with the principals of this company, and the CTO at the time of this study is a personal friend. I must also disclose that, during the two years that elapsed from when my business relationship with this site ended to when I engaged Site 2 into this study, I maintained a number of personal relationships with people in the company and I therefore have insight into this organization that other researchers would not be privileged to. This includes knowledge of a

\textsuperscript{10} NERF is an official trademark of Hasbro Corporation
near catastrophic project failure to which interview subjects from this site make numerous references.

### 3.3.7.3 Site 3: Large Business Intelligence Vendor

Site 3 is a large multi-national’s Business Intelligence R&D office in Vancouver, with some 1500+ people in the downtown office, and over 300,000 people in locations around the world. The R&D center is in a wholly-owned building in a trendy downtown Vancouver neighbourhood. The office itself is open, with modern European-style furniture, and artwork on the walls. Many posters from marketing campaigns are posted around the office. All desks are modular and uniformly organized in hexagonal-shaped pods – see Figure 14.

![Site 3 Office Layout](image)

**Figure 14: Site 3 Office Layout**

In contrast to other sites, there were minimal personal decorations, other than family photos or perhaps a pennant of a favourite sport team. The interior décor was rough sawn timbers. The
organization of the office was, at best, confusing, and it was easy to get lost. Unlike the two other sites, security was prominent at this organization.

The staff had a higher percentage of experienced individuals, and there were many people who had over 10 years with the company and over 25 years industry experience. A significant influence on the team was that their company had been acquired only a year earlier by a large Business Intelligence vendor and now the cultures were being integrated. While many saw the merger as positive, they were still in the early stages of the merge, and there were still uncertainties regarding how the merger would unfold.

### 3.3.8 Data Collection

I engaged in the field study from February 2009 through January 2010, collecting data (using semi-structured interviews and participant observation) and analyzing data over the entire 12 month period. In total, I conducted 18 interviews and spent some 42 days observing participants at work.

I incorporated both semi-structured interviews and participant observation because interview data can only provide a partial description of phenomena and accesses only what participants say they do. Participant observation allowed me to observe what people actually did, and it was an important mainstream data collection method rather than a mere supplement to the interviews. Some of my most informative data came from observing meetings and informal ad hoc conversations with participants. I considered the participant observation invaluable because it also demonstrated how participants interacted. Without participant observation, it is likely that I
would have only generated a theory of how people create software and not how they managed it. However, because Grounded Theory is intended to determine in part how individuals make meaning of a situation, perceptions can only come from interviews. One question that becomes quite important during the interviews is asking “Why?”.

I collected all data, and performed the detailed analysis of the data, while members of my supervisory committee provided guidance in clarifying, elaborating, and consolidating the emerging categories. During the many false starts and blind alleys, my committee members provided “sober second thought” to restart the analysis by simply asking generative questions of me, such as “What is the data really indicating?” and “What is really going on here?”. Some of the most productive collaborations occurred when my committee members suggested alternative concepts to those I was proposing. These collaborations helped sharpen the emerging concept, and pointed to a need for further data collection.

3.3.9 Interviews

Research ethics did not permit me to directly recruit interview subjects. When a site agreed to participate in the study, I scheduled a kick-off meeting with the participant groups to explain the study and, specifically, the ethics of informed consent. People were then invited to contact the principal researcher if they wished to directly participate in the study. Some 53 individuals from all three sites participated in this study according to our count of signed informed consent forms. Appendix B is an example of the interview consent form used in this study. While most of the participants were content to have me observe them or engage in casual conversations, many also approached me to participate in interviews. The only real problem with interviewing was finding
time in a busy individual’s schedule, and a free conference room. Study participants varied in age and experience from front line developers five years out of school to seasoned individuals with 19 years experience. Most subjects were directly involved with the creation and delivery of software, and had job titles such as project manager, software development manager, business analyst, quality assurance engineer, and developer. Table 5 lists the individuals who participated in the study interviews. All study participants were drawn from the engineering side of the organizations. The closest we came to interviewing the business side of any organization was a project manager and a business analyst.

<table>
<thead>
<tr>
<th>Subject Id</th>
<th>Role</th>
<th>Experience</th>
<th>Education</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1 Subject 1</td>
<td>Branch/Project Manager</td>
<td>13 years</td>
<td>BA</td>
</tr>
<tr>
<td>Site 1 Subject 2</td>
<td>Developer</td>
<td>5 years</td>
<td>MSc Computing Science</td>
</tr>
<tr>
<td>Site 1 Subject 3</td>
<td>Developer</td>
<td>7 years</td>
<td>BSc Computing Science</td>
</tr>
<tr>
<td>Site 2 Subject 1</td>
<td>Team Leader/Mentor/Developer</td>
<td>19 years</td>
<td></td>
</tr>
<tr>
<td>Site 2 Subject 3</td>
<td>Business Analyst</td>
<td>5 years</td>
<td>BASc, Systems Engineering</td>
</tr>
<tr>
<td>Site 2 Subject 4</td>
<td>Software Development Manager</td>
<td>10 years</td>
<td>MSc Software Engineering</td>
</tr>
<tr>
<td>Site 2 Subject 5</td>
<td>Team Lead / Mentor / Developer</td>
<td>6 years</td>
<td>MSc Mechanical Engineering</td>
</tr>
<tr>
<td>Site 3 Subject 1</td>
<td>Team Lead / Developer</td>
<td>8 years</td>
<td>BMath</td>
</tr>
<tr>
<td>Site 3 Subject 2</td>
<td>Team Lead / Developer</td>
<td>13 years</td>
<td></td>
</tr>
<tr>
<td>Site 3 Subject 3</td>
<td>Software Development Manager</td>
<td>13 years</td>
<td>BASc</td>
</tr>
<tr>
<td>Site 3 Subject 4</td>
<td>Architect</td>
<td>14 years, 5 as a developer</td>
<td>self-taught software developer</td>
</tr>
<tr>
<td>Site 3 Subject 5</td>
<td>Development Manager</td>
<td>10 years</td>
<td>BA English literature</td>
</tr>
<tr>
<td>Site 3 Subject 6</td>
<td>Lead Architect / Project Leader</td>
<td>16 years experience</td>
<td>BSc</td>
</tr>
<tr>
<td>Site 3 Subject 7</td>
<td>QA Manager</td>
<td>12 years</td>
<td>BASc Electrical Engineering</td>
</tr>
<tr>
<td>Site 3 Subject 8</td>
<td>Lead Program Manager</td>
<td>12 years</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Interview Subject Description

11 Site 2 Subject 2 was unable to participate in the interview cycle
On average, I spent approximately one hour formally interviewing each subject. With three of the subjects, I was able to conduct a formal secondary or follow-up interview to clarify and probe more deeply into new categories that emerged from their interviews. With most of the other subjects, follow-up interviews were much more ad hoc, often conducted as casual conversations in the subject’s work area, after which I quickly jotted down field notes. All interviews, with the exception of the follow-up interviews, started with the question, “Tell me in your own words how you create software here at site x.” Subsequent interview questions were guided by the subjects’ answers to this question. Appendix C is a copy of the initial interview protocol. Subjects were often asked to provide stories of both successful and less-than-successful projects in which they had participated. As the study progressed, events from participant observation generated further interview questions. Interviews were digitally recorded and then transcribed. Observation field notes were written in lab notebooks and then later scanned.

3.3.10 Participant Observation

My observation data was collected either by observing meetings or by simply sitting quietly in a cubicle and observing what was taking place around me. The second style of observation was more passive, and consisted of simply being on site and watching people’s day-to-day interactions. I was frequently invited to regularly scheduled status and planning meetings, as well as to ad hoc problem solving meetings. I watched people asking questions, observed their patterns of movement, and noted who the “go to” people were. There were also many opportunities for informal conversations between myself and the study participants. The settings provided opportunities to observe interactions between people as they stood in front of large
“information radiators” and conducted ad hoc problem solving meetings and negotiations. These observations were all captured in my field notes.

Because participant observation allowed observations of what people actually did, it served as an important mainstream data collection method. I considered the participant observation invaluable because it demonstrated firsthand how participants interacted. I was able to observe phenomena that might not have been described in interviews, from frustration with the unreasonable demands of an important client to the intense negotiation of a sprint planning meeting. I was able to observe the individuals who served as a group’s “go to” people and how they put aside their tasks to help others. I observed spontaneous problem-solving meetings, and numerous NERF Dart wars. Participant observation also provided access to data on the business side of the organization, either through informal conversations or by observing representatives of the business during status meetings. Appendix D is a copy of our participant observation protocol and Appendix F is an example of my field notes taken during a weekly status meeting.

3.3.11 Document Analysis

Two of the organizations (Site 1 and Site 2) made their internal documentation, project data, and status reports available to me. At Site 1, I had unlimited access to their online bulletin board, and was able to review their process manuals and weekly status reports as well as their corporate organization charts. At Site 2, for the product development team, most project data such as organization charts, project time lines, and release plans where posted up on the large whiteboards that covered the walls. At Site 3, I was not given access to internal company documentation. Document analysis helped me understand the context of the organization (e.g.
the organizational charts), and the event history derived from project plans and status reports. From these, I was able to create questions to ask interview subjects about what happened at those events.

### 3.3.12 Sampling

In Grounded Theory, the sample under study is that set of indicators that constitute the phenomena rather than the individuals experiencing the phenomena (Glaser & Strauss, 1967). The sampling strategy must promote the development of sufficient categories to support a conceptually dense substantive theory. Grounded Theory employs theoretical sampling where the analyst “jointly collects, codes, and analyzes his data and decides what data to collect next and where to find them in order to develop his theory as it emerges” (Glaser, 1978, p. 36). In other words, the analytic process and developing categories provide the direction for the investigator to find further incidents to develop the properties and dimensions of the concepts. Theoretical sampling is driven by the emerging categories and hypotheses; therefore, it is an ongoing process that cannot be pre-determined. Theoretical sampling supports the development of an emerging theory that is tightly integrated.

I found starting the sampling process to be problematic and analogous to the bootstrap problem. Without any data to analyze, the evolving theory has nothing to guide the sampling process and indicate where to start. I used “judgmental” sampling (Marshall, 1996) to “boot up” the process and recruit my first interview candidates, with the goal of beginning with a variety of views, and framing the topic by asking interview subjects to tell me stories about how they managed the process of software development. This first phase generated many, many open codes capturing
indicators of how people practiced software development. Clustering the codes quickly saturated (no new data) the categories directly related to how people create software. At a conceptual level, people pretty much create software the same way; however, we were more interested in how people manage the process of software development.

I started the data collection process by seeking data that described the software development process. The early interviews were initially dominated by questions about how individuals believed they created software. I asked subjects to tell me their stories of when they thought things worked, and also when things did not; from these interviews I started to construct a descriptive model of software development. The term that came up over and over again in all interviews was the idea of software development as a negotiation. What I was seeing as an observer were numerous formal and ad hoc problem solving meetings, and the category that began to direct my data collection was “Scouting”. Software development was a process of scouting unfamiliar and uncertain territory, and I began to look for data that elaborated on the concept of software development as a scouting process and data that explained how people managed this process.

What also started to influence my sampling strategy was the diversity of the software development organizations themselves. Numerous groups collaborated in the software development process and were highly diverse with respect to their role in the process, their development practices, and even their geographic distribution. Interview subjects and casual conversations were pointing me towards communications between the diverse groups. I began to seek subjects from these different groups, such as project managers, business analysts and
quality assurance personal, to collect data to elaborate the emerging categories of scouting, negotiation, and communications.

As I followed through, collecting more and more data on scouting, diversity, communications, and negotiations, scouting became eclipsed by other categories such as personal strength, sheltering, and perspective. As I compared different interviews, it first struck me how some subjects had a “big picture” view of the world when telling their stories, while others focused on their local concerns. Observationally, I noticed how some people seemed to stay close to their group, while others seemed to move between groups like butterflies. I also noticed how some people seemed exposed to such roving butterflies, were being interrupted by them, and were accepting work that diverted them from their previously committed work. What was beginning to emerge was the category of “Bunkering”, where individuals were managing their communications by seeking shelter in the bunker.

I began to collect data to elaborate this category. It was clear from my observation that some individuals sheltered others by intervening when butterflies appeared; it was also clear from some interviews that this sheltering could be so tight that no information could enter or escape the bunker; some even describing the bunker as a “black hole”. Three of the early interview participants played this sheltering role, and I conducted follow up interviews with them that were guided by these new properties. In subsequent interviews, I sought out subjects who could further elaborate these categories; most of these subjects were either senior individuals or technical management.
3.3.13 Analysis

I followed the Grounded Theory practice of coding and analyzing the data as I collected it (Glaser, 1992). As new data were collected, they were compared to existing categories. I used what I learned from the analysis to adapt my interview and observation protocols. Insights I had while coding the data and clustering the codes were captured in memos. I used Atlassian Confluence (a Wiki) to manage the data and memos. Appendix G is an example of a conceptual memo.

As I began data analysis, I generated concepts using open coding. I explored meaning in the data by looking for similarities and differences within and between interviews and field notes on participant observation, using them to categorize and label the data. While the various flavors of Grounded Theory have claimed specific practices and nomenclature for analysis, I found that they all followed the same basic approach of fracturing and integration. I used line by line coding to fracture the data as Jeon (2004) suggested, exploring all possible aspects of the issues and ideas in the data and developing descriptive in vivo codes as labels for the meanings of these issues and ideas (Jeon, 2004, p. 253). I developed categories and their properties and integrated them by testing relationships, thus developing the theory.

There are three coding phases in classical Grounded Theory: open coding, selective coding, and theoretical coding (Glaser & Strauss, 1967). Open coding (also referred to as substantive coding) generates codes that can be clustered into concepts and categories, while selective coding involves identifying the core category, the category that explains the greatest variation in the data and around which the emerging theory is built. Theoretical coding conceptualizes how
substantive codes relate to each other as hypotheses to be tested and incorporated into the substantive theory.

3.3.14 Generating Concepts - Open Coding

Open coding generates concepts from the data that will become the building blocks for the theory. During open coding, concepts are generated by asking generative questions (Glaser, 1978, p. 57) “What is this data a study of?”, “What category does this incident indicate?”, “What is actually happening in the data?”.

My first approach to open coding was to read the interview transcripts to get an overall impression – something Glaser advises against, advocating “line by line coding…. ensures the grounding of categories beyond impressionism” (Glaser, 1978). I then followed Glaser’s advice and ensured I was carefully coding line by line. It is not that every line will contain an incident, incidents may be discovered in a word, a line, or across many lines; rather, I carefully inspected every line for incidents. Open coding appears simple, and yet it is a difficult and tedious enterprise; I sometimes took nearly a week to properly code a one hour interview transcript, although some interviews were coded in about two days.

During open coding, concepts are generated by asking the generative questions previously noted (Glaser, 1978, p. 57) such as “What is this data a study of?”, “What concept does this incident indicate?”, “What is actually happening in the data?”. For example, when reading the following passage in an interview transcript:
“Otherwise, we have no control. And ah we’ll never make our dates. Another example of a project that ran rampant and it ah technically was on time and on budget, but there was so many change requests that it almost tripled in budget through the process of the project. And we um at the beginning of the project they cut out all the scope and every single piece of it got put in back through the course of the project. The project was supposed to take six months; it took a year and a half. It was supposed to be under $500,000; it took up $1.5 million. Um, things like ah they didn't do a legal review until it was way just before they launched” – Site 1 Subject 1

This quotation illustrates a project that had run rampant because the original project leadership abdicated responsibility; it was not until new leadership was put in place that control of the project was regained and the project delivered. I captured this passage as an indicator, tagging it with the in vivo code “Otherwise we have no control”. When I reflected on “What concept does this incident indicate?”, what immediately came to mind was “Leadership” or, in this example, the lack of it. I immediately jotted down a conceptual memo referring to leadership as something that steers a project towards completion. These codes were initially jotted in the transcript using the MS Word comment feature (see Appendix E for an example). I then compared these codes to concepts in the Wiki.

More indicators were developed with further reading of the interview transcript, and each new indicator was compared to previous indicators and concepts, once again asking the questions “What is going on here?” and “What concept does this incident indicate?”. Field notes captured during participant observation were also coded, initially on the field note itself, and then by comparing and adding to the contents of the Wiki. Appendix H is an example of how I
represented indicators using the Wiki. The Wiki comment feature was also handy for capturing memos associated with a specific set of indicators.

Open coding generates categories very quickly; it is easy to become enamored with categories that support pet or preconceived ideas, and also to become overwhelmed by the number of categories. Although the categories are exciting and interesting, they need to be integrated into a theory that explains how people resolve their main concern. I recognized that the constant comparative method was invaluable for helping me prune categories because categories must “earn their way into the theory” (Glaser & Holton, 2004, p. 12). Many exciting categories simply fell by the wayside because I could not find other incidents to support them or because there was not enough variation to differentiate the category from others I had already identified.

3.3.15 Discovering the Central Concern: Selective Coding

Selective coding involves identifying the core category that best explains how study participants resolve their central concern. Without a core category, there is no grounded theory; there may be, at best, an interesting collection of themes but not a well-integrated set of hypotheses that explain how the various categories operate. Furthermore, without a core category to subsequently focus the researcher’s efforts, the researcher’s energy can be dissipated into a thin inconsequential theory (Glaser & Holton, 2004)

The core category is the one category among all the categories generated during coding that, in addition to other qualities, integrates the categories and is centrally relevant. Glaser (1978) provides some 11 criteria for choosing the core category – most important to us seem to be the
criteria of being “most connected”. The core variable reoccurs frequently in the data and comes to be seen as a stable pattern that is more and more related to other variables. The generation of theory occurs around the core category. According to Glaser (1978), the first delimiting rule of Grounded Theory is that “only variables related to the core are included in the theory” (p. 93).

I regard the core category as being center of gravity that integrates closely related concepts, rather than having a loosely related set of concepts that fail to explain how a problem is being managed. I continued to develop categories relevant to the core variable and to drop more distant, less relevant, categories. These activities aided my theoretical sampling by focusing our subsequent data collection.

I initially performed selective coding by “pre-sorting” my categories to visualize which categories may “orbit” around a candidate core category. At first, I wrote the names of my categories on index cards and placed them on my dining room table. I liked this approach because of its ease and tactileness and because, in my professional consulting, I advise organizations to use similar approaches for managing their work backlogs. Nonetheless, most organizations do not have two cats and a toddler in their households and I had limited success with this approach. I switched to working with the “tree view” of my concepts in my Wiki. While this was a less satisfying and more tedious approach, it was also more robust in the face of disruptive forces.

There were many false starts for selective coding. One of our early false starts was considering the concept of “Scouting” as the core category. “Scouting” had much appeal as a core category
because it confirmed what many of us believe to be true about the process of software development: it is a wicked problem (Rittel & Webber, 1973) where a solution requires an unpredictable exploration of unknown and possible dangerous territory. “Scouting” was certainly a recurring pattern in the data; it met some of the criteria for a core category, and could be seen as a method for Getting the Job Done. On the other hand, the “Scouting” category did not explain most of the behaviors or events in the data; “Scouting” did not explain why people could go dark, or how experience and leadership influenced the process, and it definitely did not explain the efforts that people made to “get everyone on the same page”. As I directed our data collection to develop the properties of Scouting, I saw more and more incidents in the data that highlighted the need to manage communications between individuals and groups.

After more false starts, the concept of “Bunkering” began to emerge as another candidate core category. I conceptualized “Bunkering” as a boundary-setting process for how individuals and groups protected themselves from complexity and uncertainty by limiting their scope of concern. I invested considerable effort developing the “Bunkering” category because it seemed to explain all the data I was collecting; however, the resulting theory failed Glaser’s (1967, p. 111) quality criteria of parsimony. The theory of Bunkering was, in reality, many related theories, such as one describing the process of software development, another describing how people engage in a software project, and another describing the influence of leadership and personal relationships on how people engage in a software project. A factor contributing to the lack of parsimony was my lack of conceptualization; the categories I was using to try to build the theory were too literal. I resolved this issue working with my committee members to ask the generative question: “What is actually happening in the data?”. From this reasoning, the category “Reconciling Perspectives”, a
process that participants use to reconcile their different points of view and negotiate compromises, was constructed. It was surprising, once we constructed a core category that “fit”, how quickly we could integrate the theory around it, like objects being drawn into an orbit. It made us recall Glaser’s (1967) observation that the theory will integrate if we do not force our pre-conceived concepts on it.

### 3.3.16 Relating the Concepts: Theoretical Coding

Open coding breaks the data open and generates categories as the building blocks for the theory. But a grounded substantive theory is not a loose collection of categories. A substantive theory explains how people manage a problem through an integrated set of hypotheses. Glaser (1978) offers analysts coding families for organizing and associating substantive categories. At first glance, it appears that these coding families contradict Glaser’s (1998) repeated admonitions about forcing and pre-conception. According to Glaser (1978), when a relationship between hypotheses is implicit, it is weak. The use of theoretical codes explicates the nature of relationships; therefore, theoretical codes are not paradigms imposed on the emergent theory, they are accessed based on cues in the data; theoretical coding families sharpen relationships.

After selective coding, I conceptually “draped” the emerging theory over the theoretical codes. Executing the draping technique was fairly straightforward, just taking the result of selective coding and attempting to sketch the relationships between categories. At first I draped Bunkering over the “Six C’s”: Causes, Contexts, Contingencies, Consequences, Covariances, and Conditions, which Glaser (1978) refers to as the “bread and butter” theoretical codes of Sociology and states should be the first codes to keep in mind when coding data (p. 74). When I
draped Bunkering over the Six C’s, the result was a very “clunky” theory relating Personal Strength to Communications (Covariance) and its influence on the software development process (Conditions and Consequences). In Glaserian terms, it lacked parsimony because it took too many concepts to explain what was going on. The theory of Bunkering did not connect well with the concept of Perspectives and why it was important to manage Communications.

Stepping back, I realized Bunkering was one aspect of a process. When I draped the emerging theory over the Basic Social Process (BSP) framework that was centered around Reconciling Perspectives as a core category, everything dropped into place very quickly and we had an explanation of our data that both fit and was parsimonious. As a theoretical code, the prime distinction between a BSP and other theoretical codes is that “BSP’s “are processual or as we say they process out. They have two or more clear emergent stages” (Glaser, 1978, p. 97). A process is something that occurs over time, and it involves change over time which clearly distinguishes the different stages. Appendix I is a diagram from my notes capturing the first emergence of Reconciling Perspectives.

3.3.17 Memoing

There is no grounded theory without memoing – and I memoed extensively during this study, capturing both process and concept memos. Initially, I captured all memos by creating Wiki pages, and I quickly discovered that “ideas” did not emerge only when I had access to my laptop, and that I am not the type to carry a notebook with me. However, my iPhone is always with me and I made extensive use of its voice memo capability, capturing some 1072 voice memos. I used the voice memos to reflect on and guide the creation of categories and how they integrated
into the emerging theory. Easily three quarters of my memos were what could, at best, be called “flights of fantasy”; often the thought about an interview or observation would yield data that supported a pet idea. These flights of fantasy were useful because they gave me the freedom to explore ideas with the confidence that constant comparison would bring me back to Earth.

3.3.18 Sorting and Writing

In *Doing Grounded Theory*, Glaser (1998) writes “Sorting is the last stage of the grounded theory process that challenges the researcher’s creativity…. Writing is merely a write up of the sorting piles” (p. 197). For me, sorting was a challenging process, and writing was most certainly not just “merely a write up of the sorting piles”; I found myself constantly sorting and re-sorting my memos. I sorted memos by re-organizing them in the Wiki tree hierarchy, moving memos around, and creating new “parent” memos that collected and consolidated other memos. Turning simple jots, that standing on their own are completely meaningless to anyone other than myself, into meaningful clear prose was not a trivial exercise because writing up a memo made it real and identified gaps in my reasoning. In reality, the process of writing became my final sort.

3.4 Trusting Reconciling Perspectives

Is *Reconciling Perspectives* valid? I have explained Glaser’s criteria for determining the validity of a grounded theory (fit, relevancy or grab, workability, and modifiability). In the previous subsection, I described how the study design was executed to generate a theory that is grounded in the data. In the results section conceptual elements of the theory are annotated with examples
from the collected data. For those more familiar with positivist methodologies, study limitations and addressing threats to validity explains the extent to which the results of the study are valid.

In the positivist research paradigm, threats to validity address methodological issues concerning the validity of the theory constructs, and therefore the extent to which the conclusion is warranted and the extent to which the theory can be generalized. These criteria for validity are not meaningful for interpretive studies and for judging the validity of theory that is generated from the data. Gasson offers a comparison of these criteria to criteria that software engineering researchers may be more familiar with in Table 6 (Gasson, 2003):
<table>
<thead>
<tr>
<th>Issue of Concern</th>
<th>Positivist World View</th>
<th>Interpretive World View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representativeness of findings</td>
<td><strong>Objectivity</strong>: findings are free from researcher bias</td>
<td><strong>Confirmability</strong>: conclusions depend on subjects and conditions of the study rather than the researcher</td>
</tr>
<tr>
<td>Reproducibility of findings</td>
<td><strong>Reliability</strong>: the study findings can be replicated, independently of context, time or researcher</td>
<td><strong>Dependability/Auditability</strong>: the study process is consistent and reasonably stable over time and between researchers</td>
</tr>
<tr>
<td>Rigor of method</td>
<td><strong>Internal validity</strong>: a statistically- significant relationship is established, to demonstrate that certain conditions are associated with other conditions, often by “triangulation” of findings</td>
<td><strong>Internal consistency (credibility)</strong>: the research findings are credible and consistent, to the people we study and to our readers. For authenticity, our findings should be related to significant elements in the research context/situation</td>
</tr>
<tr>
<td>Generalizability of findings</td>
<td><strong>External validity</strong>: the researcher establishes a domain in which findings are generalizable</td>
<td><strong>Transferability</strong>: how far the findings/conclusions can be transferred to other contexts and how they help to derive useful theories</td>
</tr>
</tbody>
</table>

Table 6: Gasson's Comparison of Trustworthiness criteria between positivism and interpretivism.
Onwuegbuzie and Leech’ (2007) Qualitative Legitimation Model integrates a number of internal and external threats to credibility in qualitative research, some of which are addressed in the next section.

**3.4.1 Threats to Credibility**

Internal credibility in the Qualitative Legitimation models can be defined as the truth value or credibility to interpretations (Onwuegbuzie & Leech, 2007, p. 234) and the model identifies several threats to internal credibility, many of which center around the researcher’s ability to observe events and interpret collected data:

1. Voluptuous legitimation is the extent to which the researcher’s interpretation exceeds their knowledge of the data (p. 235).
2. Observational bias occurs when the data has an insufficient sampling of behaviors or words from the study participant. This is likely to occur in short field engagements (p. 235).
3. Researcher bias is the unacknowledged personal biases or a priori assumptions that he/she is unable bracket (p. 236).
4. Reactivity is the changes in personal response that result from the individual or group’s awareness of their participation in the study (p. 236).
5. Confirmation bias is the tendency for interpretations and conclusions based on new data to be overly congruent with a priori hypothesis (p. 236).
These threats to internal credibility cannot be quantified for this study; however, throughout this section I have identified how rigorously following grounded theory’s constant comparative method, concepts had to earn their way into the theory.

Researcher and observer bias is certainly a threat to validity in any research, and even more so in qualitative research because the results are the researcher’s interpretation of the data. This why I clearly stated in my background and bias in my Personal Reflections (sub-section 3.1.3) to make both myself and others aware of my biases and how they may influence my ability to observer and interpret the data. Working with my committee members decreased the likelihood of my biases operating unchecked.

I also made the results auditable by maintaining logs of my interviews and observations which included date, time, and location. I created a large bank of both theory and process memos and have offered thick descriptions of concepts and categories. Auditability and thick description supports credibility, and reduces the likelihood of observational bias, as does data triangulation. I gathered data using different mechanisms (interviews, participant observation, and some document analysis) and different types of sites. Data were collected from sites with different project types and different sizes of organizations. These activities reduced the potential effects of reactivity from study participants.
3.4.2 Member Checks and Feedback

To avoid confirmation bias I needed to engage study participants to validate that *Reconciling Perspectives* was part of their reality rather than confirming my a priori hypothesis. I made efforts to conduct individual member checks with interview subjects who were available and willing to participate in this process. It was somewhat surprising to me to find that individuals seemed either generally uninterested in the member check, or had a high degree of trust that I would not misrepresent their interview. Only six of the interview subjects were willing or available to follow through with the member check. The member checks were done informally, where I summarized the categories I gleaned from the interview and discussed emergent categories. I have a concern that I am seen as an authoritative person on the subject of software development, and that interview subjects may have been unwilling to challenge my interpretation of their interview.

This methodology section describes our process in terms of how we performed sampling and how the theory emerged. Large parts of the data were observational data and not interview data, so I was not able to approach one person and ask if my interpretation of the interview was a fair representation of their reality. During this study, I conducted a number of so-called “water cooler” conversations, where I both gathered data and also solicited feedback on emerging concepts. When the theory finally emerged, I invited study participants to seminars to explain the theory and solicit feedback. In all situations, there were no criticisms, only questions about specifics. One individual at Site 2 remarked “yeah that’s my life!” I am concerned about the level of engagement participants had with validation steps because, beyond a few questions for clarification, there were no disagreements with my interpretations of the data.
Finally, a substantive theory may be a valid explanation of a phenomena but, in an applied science, if it is not a useful explanation then it is of little value. I have taken the theory “on the road” with me and presented it in small informal settings, such as at “Agile Open North West” and at after-hours sessions at the annual Agile Alliance Conference, where I received useful and constructive feedback. The feedback I received in these situations led to some renaming of concepts, and to some tightening of the explanation of others. We have published the theory in a journal and in conference papers, and I have used these publications in commercial engagements with my clients to explain “what is going on” in their organization and to develop organizational policy.

3.5 Limitations of this Study

While I assert that Reconciling Perspectives is a good explanation for “what is going on” in the everyday lives of people participating in the process of software development, I cannot claim that it is definitive. It was not my intention to recruit only Agile teams or teams claiming to use Scrum into this study, these where the characteristics of my participants. There were significant variances between text book Scrum and its deployment at the study sites, and how Scrum was integrated with other software development practices.

The participants in this study are not intended to represent of all individuals managing the process of software development. Firstly, half of our interview data came from “larger than life” individuals who wielded significant influence over their teams, and the engineering organization.
Many of the stories of past project challenges told by these participants were of the form “yes it was challenging before but now that I am here and we are doing agile properly, things are better.” I do not believe that this was an attempt for these individuals to aggrandize themselves; rather, I believe that the interview participants were just telling their interpretation of events.

Secondly, many of our interview subjects spoke of their relationships with “others” and gave second hand stories of how “others” had succeeded or failed. Unfortunately, in most situations I was unable to recruit those others into this study to discover their interpretation of events.

Thirdly, while I have some observational data of individuals from the business side of the organization participating in status meetings, and was also able to interview individuals who operated as proxies for the business side, I do not have interview data from those on the business side of the organization.

Finally, several of the interview participants were responsible for the introduction of Agile into their organization, and were strong advocates for Agile. While the influence of these individuals is mitigated by the observational data, the observation data also comes from observing their teams in action. I have some interview and observational data suggesting that some of the Scrum teams were “golden” and insulated from customer interruptions, and I am personally aware that at two of the study sites there were significant re-organizations that downgraded the influence of the Scrum teams about a year after this study concluded.
Chapter 4

THE THEORY OF RECONCILING PERSPECTIVES

4.1 Introduction

Our analysis resulted in the construction of the theory of *Reconciling Perspectives*, a basic social structural process (BSSP) explaining how people manage the process of software development.

The presentation of *Reconciling Perspectives* is organized as follows:

- Sub-section 4.2 is a high-level synopsis of the theory and provides a context for understanding subsequent sub-sections.
- Sub-section 4.3.1 is a description of the organizational context in which the theory operates.
- Sub-section 4.4 presents the theory of *Reconciling Perspectives* in terms of the flows and relationships between concepts.

To assist the reader, I have included a concept glossary in Appendix J.

4.2 Overview

The process of software development takes place in the context of an organizational *Ecosystem* where *Acquirers* ask *Suppliers* to perform a *Job* for them. The main concern of individuals participating in the process of software development is *Getting the Job Done*. The diversity of the organizational *Ecosystem*, combined with the narrow and different specializations of the *Acquirers*
and Suppliers, can result in a Perspective Mismatch where the Acquirers and Suppliers have different Perspectives about the Job. Perspective Mismatch(es) between Acquirers and Suppliers impede Getting the Job Done.

People use a process of Reconciling Perspectives to remove the impediments to Getting the Job Done created by the Perspective Mismatch. Reconciling Perspectives is a two-stage process of Converging the Perspective Mismatch to create a Consensual Perspective and then Validating the Consensual Perspective by creating Work Products that can be objectively evaluated. Converging itself is a two stage process of Reaching Out and Negotiating Consensus. When either an Acquirer or Supplier becomes aware of a Perspective Mismatch, then the Acquirer or Supplier attempts to engage the others (Reaching Out). If the other(s) agree to participate, then they will negotiate a Consensual Perspective (Negotiating Consensus). Once a Consensual Perspective is reached, the process transitions to Validating which is another two-stage process of Bunkering and Evaluating. During Bunkering, the Supplier seeks quiet focused time to construct the Job Work Products. The participants can only really know that the Perspective Mismatch is reconciled when the Work Products are presented for acceptance (Evaluating) and are accepted by the Acquirer (Accepted Work Products).

A Consensual Perspective is subjective and results when both the Acquirer and Supplier agree they are not receiving Communications that conflict with their Perspective of the Job. It is possible during Bunkering for subsequent Communications to reveal a Perspective Mismatch and Invalidate the Consensual Perspective. Invalidation causes the process to return to Converging. Invalidation does not lead to the creation of Waste because Invalidation indicates Acquirers and Suppliers are
responding to new *Communications* that revealed a *Perspective Mismatch* when there is still time to effectively *Converge* the *Perspective Mismatch* and avoid the creation of *Waste*.

Figure 15 is a schematic of the process:

![Figure 15: Reconciling Perspectives](image)
Waste is created when the Job does not get done and the delivered Work Products do not meet the expectations of either the Acquirer or Supplier. Waste manifests itself in many forms, such as unusable Work Products, or Work Products that require rework before they can be deployed. Waste can also manifest itself in terms of dissatisfaction on the part of the Supplier. Surprise can occur when significant unexpected Waste is discovered during the Evaluating stage and Work Products dramatically fail to satisfy the expectations. There are four ways Waste is created in Reconciling Perspectives:

<table>
<thead>
<tr>
<th>Failure Path</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to Reach Out</td>
<td>The individual or organization noticing a Perspective Mismatch never Reaches Out, or other parties decline to engage.</td>
</tr>
<tr>
<td>Convergence Stalling</td>
<td>The negotiation never reaches an explicit Consensual Perspective and both Supplier and Acquirer continue to work with a Perspective Mismatch.</td>
</tr>
<tr>
<td>Isolation</td>
<td>During Bunkering, the Supplier rigorously filter all Communications thereby ignoring Communications that may either Invalidate the Consensual Perspective or indicate further Perspective Mismatches.</td>
</tr>
<tr>
<td>Saturation</td>
<td>During Bunkering, the Supplier does not filter any Communications such that they become distracted and unable to focus on creating the Work Products.</td>
</tr>
</tbody>
</table>

Table 7: Reconciling Perspectives Failure Path Flows.

Colloquially, we can diagnose and categorize these software process failure patterns as either:

1. Ignoring the “Elephant in the Room” and not engaging with colleagues when suspecting there is a Perspective Mismatch (Failure to Reach Out, Isolation)

2. Not reaching a collaborative decision and/or moving forward in a timely manner (Convergence Stalling, Saturation)
Reconciling Perspectives is characterized by four properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>The number of individuals in the organization affected by the Perspective Mismatch</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>The interval of time from when the Perspective Mismatch is detected until the reconciliation is evaluated</td>
</tr>
<tr>
<td>Communications</td>
<td>The frequency, volume, and diversity of information exchange between individuals in the organization</td>
</tr>
<tr>
<td>Social Dynamics</td>
<td>The capability of a team to manage their Communications</td>
</tr>
</tbody>
</table>

Table 8: Reconciling Perspectives Properties

Reconciling Perspectives requires individuals and teams to be concurrently open to Communications, so they can become aware of Perspective Mismatches, while also being sufficiently sheltered from Communications so they can focus on constructing Job Work Products. This need creates an inherent Communications tension in the process that must be managed. Reconciling Perspectives is moderated by Social Dynamics that enables individuals and teams to effectively manage their Communications. Positive Social Dynamics in terms of Personal Strength, Leadership, and Openness in individuals balances this tension. The process is scalable and applies to all project Scopes and durations (Cycle Time), from the short conversation between developers to the protracted conversations between development and stake holder organizations.
Mismanaging *Communications* was a root cause explanation for many of the stories of challenged projects encountered during this study. *Social Dynamics* is the capability of individuals and teams to manage their *Communications* and engage with others appropriately to resolve impediments to *Getting the Job Done* while not becoming overwhelmed by extraneous noise. *Social Dynamics* moderates *Reconciling Perspectives* and explains why different flows through the process may unfold.

*Reconciling Perspectives* is not a single thread process, and individuals may be participating in a large number of independent instances of *Reconciling Perspectives* in different roles, with a variety of *Scopes*, and *Cycle Times* (Figure 16). An individual *Bunkering* in one instance as a *Supplier* may be actively engaged in another instances as an *Acquirer* *Negotiating Consensus*. For example, a team lead may be *Negotiating Consensus* with their product owner, at an organizational *Scope* with a *Cycle Time* measured in weeks, while concurrently negotiating an interface for an API with local *Scope* and a *Cycle Time* measured in hours, if not minutes.
Figure 16: Multiple Overlapping Instances of Reconciling Perspectives

The multiple instances may be unrelated in that the areas of concern are not dependent, but the multiple instances can still interfere with one another. An individual or team *Bunkering* in one instance may not be sufficiently open to *Communications* to detect a new *Perspective Mismatch* or an *Invalidation* of a *Consensual Perspectives* in other instances.
4.3 The Organizational Context of Reconciling Perspectives

4.3.1 The Process of Software Development

The software development process is not a conceptual part of Reconciling Perspectives; yet to understand how people managed the process of software development, we need to understand how they understand the process of software development. An assumption built into our research question is that people know what the software development process they are managing is\(^1\). Interviews started with the question “In your own words, tell me how you develop software here.” Most of our more technically focused interview participants provided a broad and shallow description of the organizational software development process, punctuated by a more detailed description of the team level processes.

Most interview subjects did not describe technical practices or work products beyond describing a few requirements documents. Rather, most described their software development process in terms of social interactions, especially at the team level. Project managers and analysts tended to see software development more as a requirements management process than a software development process. The following interview excerpts offer a descriptive glimpse into how people understood their software development process at study sites:

---

\(^1\) This may not be a valid assumption because as one technical manager observed – “we are good at writing code, but we have trouble delivering software”
“Well, I think um software development starts with a need. Um a business needs to be able to do something and ah we build applications for them to be able to do it. So, that's really where it starts” – Site 1 Subject 1 35:36

“So, basically the client ah comes up with some requirements or I would say draft requirements and I do some brainstorming as to what they want or what they really want” – Site 1 Subject 2 54:55

“we spend a lot of time looking at the functionality, discussing it in detail, discussing the complexities of it and discussing all the bits and pieces and steps that we'll need to do to do the functionality. We don't write that down. We might write down chunks of it. You know, like um needs an automated test script to - needs a new QA environment. We won't um, but we won't write it down in great - in detail. It will be very high level. And ah, so once we do that, then we get back down with the business people and we ah prioritize those development units or those user stories. And then we'll ah break them out into iterations” – Site 1 Subject 1 72:83

“typically what we do um so that as a development team we’ll then – oh yeah – they’ll be a developer or a couple of developers or maybe the team and they’ll say this story, which we’ve said is you know maybe a five point, which is maybe around a week’s work. We’ll then break that down into smaller tasks than they’ll be a couple of write these costs or write this code or this sort of testing. So that’s the sort of finest – finest grain of tasks” – Site 2 Subject 1 42:47
There was significant variation in how teams performed their quality assurance (QA) tasks. Some teams had good libraries of automated regression tests and had integrated QA into their implementation process.

“So [use to be] a lot of throwing over the wall stuff. We—we’ve brought the QA’s into the teams and really integrated them so we QA as we go along. So it’s—you know a day or two days before developers get feedback on—the stuff they’ve committed. But quite often QA—sort of QA’s they literally as the developers are developing as well. So develop a bit—QA a bit—develop a bit. We do quite a bit of automated testing—unit testing and system testing and that sort of thing so um—all of that around the up-front planning that we try and do” – Site 2 Subject 4 28:33

Within the same organization, there could be different approaches to QA, some much more disjointed than others:

“In PS there seems to be more of a lag between—the time the developers develop a component and the time it gets actually QA'd. There can be a lag from—anywhere from a week-2 weeks-3 weeks—a few months. In PD they—they tend to operate in quick turn around on-on QA. They'll operate in little groups of a couple developers to a QA and they'll get quick QA feedback—as to the quality or if it's meeting the-the spec or if it's going to play nicely with other parts of the system. In PS there's a much longer lag because they'll just say okay everyone develop these pieces. You typically have less QA resources internally—if any And then you'll—you'll kind of be waiting for the feedback from the client's QA. So whenever—they'll have a different schedule or
different mechanism or methodology of approaching the QA work. So it's kind of more disjointed” – Site 2 Subject 5 222:232

Figure 17 describes a three-stage process of software development that emerged from interviews. What we normally think of as the “text book” stages of software development – requirements elicitation, requirements analysis, design, implementation, and test – cut across the boundaries of the stages in the process as described by the interview subjects.

Figure 17: The Software Development Process
Stage | Process
--- | ---
Negotiating the Scope and Priority of the Problem | Interview participants tended to describe the requirements elicitation and analysis process very much in terms of trying to understand how much work there was and what the priorities were.
Investigating the Solution(s) | If people were not negotiating then they were investigating…often they seem to move rapidly back and forth between negotiating and investigating. The term “discuss” was frequently used here, but the process described and observed was a negotiation of solution trade-offs, time, cost, and technical complexity.
Validate the Solution | Validating the solution was the exit point of the software development process and, in all cases, required the involvement of quality assurance regardless of whether QA were integrated with the team or were a separate team.

Table 9: Process of Software Development as Described by Interview Participants

From our observations, there was a high degree of overlap between the stages and, while interview subjects identified multiple stages in the software development process, few subjects clearly distinguished between the stages or definitive transitions between the stages because individuals tended to move back and forth freely between the stages. Even with the documented process, it seemed that the stage gates were treated more as “advice” rather than a prescription

The only time we could definitely say where we were in the process is during team status meetings or project status meetings, where it was explicitly stated where the project was in the

---

13 The best analogy for the utility of the documented process is when I was driving in Rome, where four lanes of traffic were driving in three marked lanes. My colleague explained that the white lines were advice and rather handy when it was foggy.
process. It was also hard to determine if this status was more a declaration than a statement of reality.

All participants said they were using Scrum at the team level, and then immediately acknowledged that they were either not following it consistently or that they had adapted it to their needs and our observations support this “metaphoric” interpretation of Scrum. Some teams worked with a product backlog, executed in fixed time boxes, and had an engaged Product Owner and designated Scrum Master. For some teams, the only connection with Scrum was their ritualized daily standup meeting. Furthermore, the use of Scrum was not an official organizational standard, and all teams worked within the context of an organization-wide software development methodology. For the majority of participants in this study, Scrum was only used for the implementation phase of a project. The following interview excerpts offer a descriptive glimpse into the application of Scrum at the study sites:

“Our team uses the Scrum process. Um, so but unfortunately we have to fit within a larger organization that's using ah a waterfall methodology” – Site 3 Subject 3 7:9

“Okay-okay. So—a lot of my efforts—as the manager and kind of as the product owner is-is mapping our current—water flow—waterfall process to our—our SCRUM development process. And—so—generally—the—only our development team really is-is practicing or knowledgeable in-in the SCRUM process. So for example a—someone who writes the software requirement—generally I will take aspects of that and-and-and actually cut and paste and turn it into an epic—and-and create our—and-and create our project. From that I'll create our—releases and sprints
and so on. And so I end up doing a fair amount of mappings—as-as I said—from an SRS kind of to an epic—what we call a P-feed to a theme and-and from that a use-case into an actual backlog and stuff like that. And working with those parties to kind of sometimes define acceptance crit—criteria as well. And of that is kind of—an ongoing process. I'm hosting backlog grooming meetings where we're—we're taking some of these use cases and trying to-to query in—get acceptance criteria and then also kind of manage the releases and stuff like that. And then—on a day to day basis it-it still is a lot of reporting on projects outside as well” – Site 3 Subject 5 39:50

“so I mean what we'll do is of course we'll do a standard sprint planning session at every two weeks. Um, sprint planning session consists of our demo, followed by our - ah - our - we do a retrospective to see if there's anything we can fix in our process. Um, then we'll do our actual planning session where we'll start picking things off the top and pulling them into the sprint. Um, at the point developers - different developers will take each of the backlogs and volunteer to break them down into tasks. And then once the tasks are broken down, we meet again for a quick 20-minute meeting just to ah - just to assign tasks to different developers and different people. And sometimes we will take on entire backlogs and sometimes we can parallelize a backlog out to multiple developers. And we'll split it like that” – Site 3 Subject 1 97:106

“Yeah. And each PS team operates differently from each other PS team. There—there's one I had experience with recently—that is project manager driven. Work's still assigned to individual devs—the devs don't talk overly—they don't collaborate—they don't finish what they're doing. They try and you know they get on at work on as many things as they possibly can at the same
time and it's—they're not very strong QA and it's not a continuous QA it's not a collaborative QA again. There's no real iterations. Iterations are kind of arbitrary as a—it's a nice name to say” – Site 2 Subject 4 364:381

There was little evidence of documented technical practices or working agreements for the team level practices. During interviews, when I asked how individuals learned the process, people answered “by asking other team members.” Experienced individuals seem to flow into teams quite well, whereas new grads found the lack of documented technical practices challenging. The team process in use was dependent on whoever was considered to be the team’s technical thought leader. This was clearly evident at Site 3, where some teams used Scrum and others did not based on the technical thought leader’s view of the methodology.

4.3.2 The Organizational Ecosystem

Software development occurs in the context of a greater organizational Ecosystem, where an Acquirer has a Job they wish done and a Supplier has at least some of the ability to perform that Job. The need for the Job emerges from the Ecosystem:

“...software development starts with a need. Um a business needs to be able to do something and ah we build applications for them to be able to do it. So, that’s really where it starts” – Site 1 Subject 1 35:37

We use the term Ecosystem to describe the organization because, from our observations, software development takes place in a diverse and chaotic mix of software methods, corporate policies,
competing interests, personal agendas, personality types, and among all variety of formal and informal relationships, task forces, alliances, and in-groups. In ecological terms, there is a great bio-diversity, and organizations are anything but a mono-culture. All these diverse cultural groups must work together to create software. This diverse Ecosystem is also a great source of uncertainty for a software project, and the expression “Change Happens” captures the view of many developers because the Ecosystem is constantly changing which consequently changes the Job.

“Um change is always guaranteed, right? Release dates may move, priorities may move or the customers may demand something else” – Site 2 Subject 1 67:69

At each site there was at least one plainly evident cultural divide. Site 1 was a small fast-moving team embedded in the service of a large quasi government organization. At Site 2, the development organization was split between a highly energized and focused product development group and a highly pressured customer-facing services group reacting to seemingly chaotic customer demands. At Site 3, a formerly local company was in the midst of a merger with a large multi-national.

There were also divides by organizational division such as product management, development, and testing. At Site 3, which was considerably larger than Site 1 or 2, there were cultural divisions by major component group, and also by geographic boundaries. Each of these groups had their own norms, rituals and language, factors that can impede collaboration.
“it’s kind of difficult having one aspect of a company be agile and then other aspects not really fully comprehending what that means” – Site 2 Subject 3 89:90

Getting the Job Done requires the Supplier to remain connected to the organizational Ecosystem because it is the source of the information that the Supplier needs to Get the Job Done; however, it is also a significant source of interference, which can impede Getting the Job Done:

“it's obviously not a controlled environment cause there's a lot of external interference” – Site 2 Subject 3 98:99

4.3.3 A Central Concern: Getting the Job Done

The central concern of participants was Getting the Job Done, that is, delivering the best Work Products they knew how to create. A major source of frustration were impediments to Getting the Job Done.

“Um for myself particularly the concern is ah being blocked to something. I can't do, I can't achieve by myself. I know that supplying this would be sort of would hamper progress (unclear). Sometimes maybe to the point where the project is completely halted for a long period of time until it's resolved. And so I think that's my main concern is um is really just not being able to do the work that ah that needs to be done” – Site 1 Subject 3 lines 205:209

Participants defined Getting the Job Done as creating working software that:

- Appeases the Acquirer or makes the Acquirer happy,
- Satisfies the Supplier's need to see the end and achieve a feeling of accomplishment, and
• Satisfies the Supplier’s desires to minimize technical debt.

Despite marketing hyperbole of “delighting customers”, the customer satisfaction bar is set low at appeasement or, more forthrightly, at “anything that keeps the heat off”.

“They feel that they’re under pressure so they just have to find a way to get it to work” – Site 2 Subject 5 461:461

In situations where Supplier teams were located at an Acquirer’s site, Suppliers felt pressure to trade-off their needs in order to satisfy Acquirer demands. While the Work Product may appease the Acquirer, Suppliers felt that their needs for Getting the Job Done were not met.

“I don't really enjoy solving a problem as much when I'm given just an hour to deliver something or a few hours to deliver something that I know will—if I don't spend more time it won't be as good quality as it should be.” – Site 2 Subject 5 495:497

For Suppliers that were distant from a specific Acquirer, Getting the Job Done focused more on technical craftsmanship and greater satisfaction with the Job. Much of the frustration expressed by software developers during the interviews was because they believed that Getting the Job Done was not only about delivering software to the Acquirer but also about delivering software that met their standard of quality and technical craftsmanship.
4.3.4 Acquirers and Suppliers

Software development is a process where an Acquirer asks a Supplier to perform a Job for them. The Acquirer has a need they wish to have fulfilled and they engage a Supplier to create a set of Work Products that satisfies the Job requirements. We observed the Acquirer-Supplier relationship at all levels in the organization. The Acquirer may be an individual or group representing a customer organization requesting a new system feature, or a single individual requesting a change to a small technical implementation feature. The Supplier may also be an individual or group representing an engineering organization capable of delivering complete systems to a single individual. During the interviews, participants explicitly described organization level Acquirer-Supplier relationships where customer representatives negotiated system feature requirements with directly with representatives from the engineering organization.

“A business needs to be able to do something and ah we build applications for them to be able to do it. So, that’s really where it starts” – Site 1 Subject 1 35:37

“So here in—like we start with the—something called Market Requirement Documents (MRD). So that is delegated by looking at what are the trends in the market? What is needed? And where are the opportunities? So that starts at the product management level” – Site 3 Subject 2 32:34

While most interview participants spoke of Acquirer-Supplier relationships explicitly at the organizational level, we also observed the same Acquirer-Supplier relationship at the team and individual level where, within a team, one individual may request a local feature change to a sub-system that another individual owned. Most of the teams participating in this study were
component teams and owned a specific set of a system’s components. Adding a new feature to a system often required one team to request another team to provide a new Application Program Interface (API) or modify the behaviour of a component they owned. Often the teams were collocated, and these negotiations were informal; however, as geographic distance increased, these negotiations became much more formal.

An individual or group could be in multiple Acquirer-Supplier relationships and be both an Acquirer and a Supplier. For example, an individual could be a member of an engineering team engaged by a customer to develop a system feature and therefore be a Supplier to that customer. To get their part of the job done, that same individual could then need another individual or team to modify a component they owned, therefore also being an Acquirer to the other team.

### 4.3.5 Perspectives

A Perspective is an individual’s point of view, which they use to interpret the Job and establish what they believe “done” means. A Perspective establishes expectations for what Work Products should be delivered for a Job and informs the Job’s trade-off priorities.

### 4.3.6 Perspectives Mismatches

Throughout this study, we discovered that a frequent and difficult problem in software development is “getting everyone on the same page”: 
“And the biggest problem you have is—I can think of one case recently where we had the team lead here and—one of the managers here and a team lead at the (unclear) team kind of discussing this bug and the three of us were discussing it in lieu for a week only to realize that all of us had completely different interpretations of it….So we weren't even on the same page on it. So it leaves more room for miscommunication which is—not a good problem in software.” – Site 2 Subject 3 260:268

Everyone seems to have a different point of view or Perspective about what the Job is and the process for delivering the Job is. Much like how members of a choir must all sing from the same sheet of music, everyone participating in the software project must also see the Job the same way for the Job to get done. The inability to get everyone on the same page is a Perspective Mismatch and a significant impediment to Getting the Job Done. Perspective Mismatches often result because individuals are mostly disconnected from the organizational Ecosystem and only understand their localized area:

“I just find it creates a lot of friction because people don't - only understand their layer” – Site 3 Subject 6 470:470
There are four broad *Job* areas where *Perspective Mismatches* can occur:

<table>
<thead>
<tr>
<th>Job Area</th>
<th>Perspective Mismatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>The behavior, scope (what features are in, and what features are out), and quality of <em>Job</em> features</td>
</tr>
<tr>
<td>Priorities</td>
<td>The order and relative importance of <em>Job</em> features with respect to other <em>Job</em> features and other <em>Jobs</em></td>
</tr>
<tr>
<td>Method and Work Products</td>
<td>What <em>Work Products</em> are created, how those <em>Work Products</em> are created, and when in the software life cycle those <em>Work Products</em> are delivered</td>
</tr>
<tr>
<td>Schedule and Status</td>
<td>When a <em>Work Product</em> is ready and the current quality of <em>Work Products</em></td>
</tr>
</tbody>
</table>

**Table 10: Job Areas Where Perspective Mismatches may Occur**

Feature quality is a significant source of *Perspective Mismatches* because feature quality attributes, such as scalability, reliability, and supportability, are reflective of the *Supplier’s* need for technical excellence in *Getting the Job Done* whereas, for the *Acquirer*, attention to such quality attributes may be seen as “gold plating”. The result is that the *Acquirer* sees attention to these quality attributes as *Waste* and is *Surprised* when the *Job* is presented to them and it does not have the behaviour expected or needed, or that the costs have escalated because the *Job Work Products* were built far beyond what was needed. A common expression describing this scenario is “*we wanted a Volkswagen and they gave us a Cadillac*” and, during planning meetings, it was not unusual to hear a Product Owner dismiss a delivery team’s proposal for a quality enhancement because the Product Owner did not see the benefit. In some situations, the delivery team’s desire for quality enhancement was so strong that, despite the Product Owner’s dismissal, they proceeded with the enhancement anyway. Conflicting priorities were another source of
Perspective Mismatches because, in the organizational Ecosystem, there are many individuals with different agendas and therefore opinions about what should be done first.

“it might get confrontational from time to time. I would probably say usually gets - it's pretty - it's a pretty decent discussion but ah you know each person has their own ah priorities...You know you - we can ship something that you know it's great but the customer doesn't want it because it doesn't do what it needs, so obviously there's pressure and legitimate pressure from both sides. Like there's pressure from the you know what you want quality - we can't give it to you unless we cut down on some of the features, but on the other hand there's pressure from marketing saying, you know what if you're sure you want to make some money you might need to put some stuff in” – Site 3 Subject 7 157:176

All organizations participating in this study claimed to have a corporate-wide software development methodology that was intended to ensure everyone created software in the same manner, followed the same processes and created the same types of Work Products. However, what we observed is that few people had any interest in the corporate methodology, with the result that there were many different Perspectives on the method for creating Work Products:

“Um, I'll be honest - as somebody who's a lot further down the chain I very rarely pay attention to that process or the milestones associated with it. I hear vague um speeches of it. You know, we're hitting EMRA and whatever, or whatever the milestone name happens to be. And there are big meetings to prep for it. But as somebody a lot further down the chain, to me, it's just dates” – Site 3 Subject 1 19:24
4.4 Reconciling Perspectives

*Reconciling Perspectives* is a basic social structural process that explains how people socially interact to remove the impediments created by a *Perspective Mismatch* to *Getting the Job Done*. There are two outcomes of the process, *Accepted Work Products* or *Waste*. There are two clear stages that account for variations observed in the data, *Converging* and *Validating* and four properties that characterize the process, *Scope*, *Cycle Time*, *Communications*, and *Social Dynamics*. Figure 15 is a schematic of the process.

4.4.1 Process Flow

4.4.1.1 Converging: Reaching Out

The process begins when a *Perspective Mismatch* is suspected by either an *Acquirer* or *Supplier* and they begin *Converging* the mismatched *Perspectives* by *Reaching Out* to the other. *Reaching Out* is the first stage of *Converging* (Figure 18) and begins when either the *Acquirer* or the *Supplies* receives *Communications* that does not match their *Perspective* of the *Job*. 
Observing Scrum Sprint planning meetings provided rich opportunity for examples of Reaching Out such as when a developer at Site 2 asked the Product Owner who was explaining a feature during one sprint planning meeting “But I thought we were getting rid of wizards?” This inquiry revealed a Perspective Mismatch between the Product Owner (Acquirer) and the developer (Supplier) regarding the desired implementation of this feature. The Product Owner’s explanation of the feature during the sprint planning meeting called for the use of a wizard for configuring the feature, whereas the developer was under the impression that organizational policy was to move away from using wizards. What the developer heard the Product Owner Communicating did not match the Perspective the developer had of the Job.
4.4.1.2 Converging: Negotiating Consensus

When the other agrees to engage with the one Reaching Out, they begin Negotiating Consensus (Figure 19) to converge the Perspective Mismatch to a Consensual Perspective. Negotiating Consensus is a dialog between Acquirers and Suppliers intended to converge their Perspectives and hence their expectations for the Job and remove the impediment to Getting the Job Done created by the Perspective Mismatch.

![Figure 19: Converging: Negotiating Consensus](image)

The intent behind the format of Scrum’s Sprint planning meetings is to encourage Reaching Out and Negotiating Consensus. Acquirers and Suppliers come together to negotiate features and plan work for the next sprint. During the sprint planning meeting at Site 2 where the one developer had remarked “but I thought we were moving away from wizards”, the Product Owner explained why he thought a wizard was appropriate for configuring this feature. There was a “spirited’
discussion of alternative approaches and the effort required for their implementation. It was clear the developer was not happy with the proposal to use the wizard, but the developer also understood why from the point of view of the Product Owner, the wizard was the desired approach.

A senior developer at Site 3 explained the insights he gained during the Sprint Planning meeting:

“I got a little bit of a better insight as to what the - the requirements were and why we were taking things on. Um, and after talking to the product owner what we ended up doing was we started during our sprint planning sessions ah with the whole team. Now what we do is we actually go over each backlog and we explain why this backlog is here and why it’s a high priority. So the team has a good feel for why we’re taking on what we’re taking on” – Site 3 Subject 1 137:142

Here the Consensual Perspective is described in terms of having a “better feel”. During an interview at Site 2, one developer explained how the Consensual Perspective “gave them focus” and how that focus enabled them to work more productively:

“We got people talking. We tried to have a focus for each iteration so that 2 or 3 people could work on the same thing and make a more cohesive solution—sort of thing” – Site 2 Subject 4 460:463

4.4.1.3 Validating: Bunkering

Once a Consensual Perspective is reached, the process transitions to Validating. Consensual Perspective is subjective because the Acquirer and Supplier are not receiving Communications that
conflicts with their Perspectives of the Job. The Consensual Perspective is not validated until the
Job Work Products are created and accepted. After Converging, both the Supplier and the Acquirer
believe their expectations of the Job are aligned, but there is no objective evidence for this until
the Job is done. Only when the Work Products for the Job are delivered by the Supplier to the
Acquirer during Evaluating is there objective evidence that the Perspective Mismatch is reconciled.

The first stage in Validating is Bunkering where the Supplier seeks the opportunity to focus on
creating the Job Work Products. During Bunkering the Supplier prefers to avoid distraction and
therefore reduces Communications (Figure 20).

Figure 20: Validating: Bunkering
**Bunkering** is the opportunity for individuals and teams to “get into a flow” and work productively creating the *Work Products* that will *Get the Job Done*. One team lead explained the experience and benefits of uninterrupted flow during *Bunkering*:

“I guess cause you don’t have to sort of switch gears—like you’ve got more of a flow. You tend to get a little more efficient—basically when you plan that way everything is—that you’re doing in those 2 weeks is around the same area. You’re using the same—development environment. It’s the same project and everything right so you just like—this task—sit down do this—done” – Site 2 Subject 1 follow-up 62:66

A *Consensual Perspective* is subjective and results when both the *Acquirer* and *Supplier* agree they are not receiving *Communications* that conflict with their *Perspective* of the *Job*. It is possible for subsequent *Communications* to reveal a *Perspective Mismatch* and *invalidate* the *Consensual Perspective*. *Invalidation* causes the process to return to *Converging*. As one business analyst put it:

“you really don’t know what you are getting into until you get into it”:

“we don’t know about anything until we develop something and then we hit it and go oh there is—there’s a problem right? So it—in fact we had to turn around several times like that. But that was okay like you know. And we worked around those blocks” – Site 2 Subject 3 315:319

The *Ecosystem* is a “noisy” place with numerous distractions and interruptions. To maintain the *Bunker* individuals must have the *Personal Strength* to decline or defer interruptions or risk becoming distracted and not completing the *Job Work Products*. Individuals do not have to stand on their own and are often *Sheltered* by others assuming *Leadership* positions where they either
deflect interruptions or take on those interruptions themselves. One individual explained this Sheltering role by telling a story of his response when a team was asked to take on more work:

“And I kind of said, “Hold the – hold the phone, right now. We’re not – we’re not burning down very well. It’s looking a little bit scary, as for delivery on Thursday. And ah if you’re spending two days writing this stuff for the MDAS team, you’re not working on stuff that we asked for. And that’s not stuff that our product owner was looking for” – Site 3 Subject 1 230:234

A consequence of Bunkering and the desire to reduce Communications is this reduces the opportunity to become aware of Perspective Mismatches because individuals become aware of Perspective Mismatches when the Communications they are receiving conflicts with their Perspective of a Job. One senior developer suggested one root cause of their development challenges were the developers were far too busy writing software to Communicate with the project managers (PJMs):

“If you articulate that clearly to the stakeholders, then you or (unclear) whatever can make the call. And so okay in this case I think this. And there should be a dialogue there. And that doesn’t always happen. And that’s tough because people are busy and ah a lot of the developers I think don’t actually engage in that much dialogue. Or at least I observe, don’t engage in that much dialogue with the PJMs which I - I think if we had a higher ratio of PJMs developers that might be a better situation” – Site 3 Subject 6 989:985

The need to be both concurrently open and closed to Communications creates a tension in the process that individuals have difficulty with managing on their own. Most individuals tended to either Cut-off all Communications and risk Isolation, or engaged in all Communications and risk
Saturation. One senior developer described the desire to reduce Communications during Bunkering as “losing value”

“But I think it’s a very important part of ah - ah the development process is for – the developers are going to know the realm of the possible and the realm of the trade-offs once they start getting into it. And if that never gets communicated back to the stakeholder or stakeholder representative you lose a lot of value in the process” – Site 3 Subject 6 983:986

Other individuals simply could not say no, became Exposed and found themselves floundering as they fell into Saturation. One business analyst characterized Exposure as a problem of having “too many masters” and therefore not being able to focus on Getting the Job Done for any one Job:

“Oh you have it on different levels. There's always different levels of priority. So—kind of expecting having the team meet high expectations from several different clients and work on the existing project. So that kind of detracts the attention from the project that's ongoing” – Site 2 Subject 3 follow-up 102:105

One practice many teams participating in this study used to manage this tension were regular and frequent Communications check point ceremonies that promoted short Cycle Times. All the teams participating in this study claimed to use the Scrum methodology which uses short time boxes and collaborative planning meetings. Facilitation practices during Scrum planning meetings are intended to encourage Communications that reveal Perspective Mismatches and Converge Perspective Mismatches. Once a plan is agreed and committed to, the participants are afforded a

14 While all participants claimed to follow Scrum there was wide variation in their practices and in my personal opinion less than a third of the teams followed Scrum beyond holding regular stand-up meetings.
quiet and undisturbed interval during which they can focus on creating the Work Products without interruptions.

A common failure pattern in this practice was many teams and individuals did not appreciate the intention of these checkpoint ceremonies and only “went through the motions”. Many stories of challenged projects can be traced back to limited engagement and Communications during the checkpoint ceremonies. A lack of Diversity in Communications Isolated individuals and teams and they either were not aware of Perspective Mismatches or chose not to resolve them by Reaching Out. This situation was evident in one developer’s story of a near catastrophic project failure:

“So—there was no real—time boxing. You know it just—it was just—a flow. It just kept going and not a good flow in the sense of sort of (unclear) sort of style. It would just—it was just one great big lump. There was no—there were no markers for developers. They just came in Monday morning and they just carried on with what they were doing anyway. And the other aspect of that is that—work was assigned by team leads or project managers to specific developers. There was no theme for the iterations. So no-no two developers were ever working on the same thing. So there was no collaboration really available beyond the very—very technical questions about how you do a certain thing. There was no collaboration on the sort of—well does this make sense in this domain? Because no one else was in the same head space. So there was—it—it’s got to be one of the quietest projects I’ve ever worked on. It was—no developers were talking to anyone” – Site 2 Subject 4 155:166

Another practice for managing this Communications tension were individuals who took a Leadership responsibility to actively engage in Communications and then Alert others if they
became aware of a *Perspective Mismatch*. At some of the sites participating in this study, this *Alerting* role was considered as part of an individual’s job description:

“Ah, we’ve got ah a number of people called program managers that act as sort of the liaison between the product management group and development” – Site 3 Subject 3 68:70

Others took on the responsibility and jumped in when they became aware of a *Perspective Mismatch*:

“it’s ah – mostly it’s - it’s you know getting involved with the team when specific things come up you know? I tend to do ah much more evolutionary architecture kind of approach. It’s not ah you know a big design up front or top down architecture. I think all of the teams own the particular micro-architecture of what they do. But at the same time you can always see these frictions between the teams as communication comes up or when there’s decisions around ah should this particular function go in this layer or this layer or should it be done in this component or that component. That’s where I tend to get involved” – Site 3 Subject 6 419:425

At each of the sites participating in the study, more senior and *Experienced* individuals assumed the *Alerting* role. Some actively “roamed” the organization comparing the *Communications* they were hearing from one team with the *Communications* they were hearing from others. Others maintained “higher level” or broader view of the *Job* and *Communicated* with individuals and teams when they suspected a *Perspective Mismatch*. 
4.4.1.4 Validating: Evaluating

When the Work Products are ready, they are evaluated and accepted by the Acquirer and Supplier, which results in Accepted Work Products and validates the Consensual Perspective (Figure 21).

![Diagram showing the process of Converging, Validating, Reaching Out, Negotiating Consensus, and Bunkering, with an arrow leading to Evaluating.]

**Figure 21: Validation: Evaluating**

The teams in this study that were actually following Scrum used the Scrum time-box to establish the rhythm for when Work Products were ready for Evaluating. At Site 2 one of the BAs joked how she forced the Work Products to Evaluating on an almost daily basis:

“I've seen slight deviations. But the fact that we work with iterations—at least in the projects that I've worked on so far as a BA they've been pretty locked solid so you assign the work for 2 weeks—we agree on it—I'll see at the end of 2 weeks minimum—typically we tend to review the build actually every single morning. So that's a bit of a control freak aspect of me (laughing). So I like to see what they've done every night. I get to see in their daily reports what they've done. It
also allows me to give them a faster feedback on maybe some places where they're deviating from requirements” – Site 2 Subject 3 187:194

Every software methodology has some practice that forces Accepting however the Cycle Times can vary widely. Time boxed methods like Scrum encourage shorter Cycle Times, usually less than one month, whereas other software methodologies may defer Acceptance for an extended period of time.

“I really like working in iterations—I think that we could still be better at it. I would love to see a day where we could actually re-release at the end of an iteration and we're not there. But at the same time it's much better than having a project that lasts you know a year even two years. At the end of it it's like when is this thing going to end? And then you completely forget what the original goal was. So I've worked on a project like that as well and that's kind of—it's a bit demoralizing I think working on projects like that where you it seems nothing's been accomplished and so much time has gone by” – Site 2 Subject 3 385:391

4.4.2 Outcomes

Reconciling Perspectives has two outcomes, Accepted Work Products or Waste. Significant unexpected Waste may also result in Surprise.
4.4.2.1 Accepted Work Products

Getting the Job Done means creating Work Products that at least appease the Acquirer and provide the Supplier with some sense of accomplishment. An Accepted Work Product satisfies the Acquirer’s and Supplier’s expectations and is objective evidence the Consensual Perspective was valid. A Work Product may be only partially complete or deficient and still accepted if it was the expectation was the Work Product would be deficient. This situation arose frequently during Scouting when the purpose for creating the Work Products was to gain knowledge that could subsequently help resolve a Perspective Mismatch. There were many situations where an Acquirer really “didn’t know what they wanted but would know it when they saw it, and the expectation was “low fidelity” Work Products would help narrow the solution space.

4.4.2.2 Waste and Surprise

Waste is loss of value that results when the delivered Work Products do not meet expectations of the Acquirer or the Supplier. From the Acquirer’s point of view, sources of Waste are:

- Missing or deficient features,
- Unnecessary or over engineered features, or
- Opportunity cost and lost revenue resulting from late delivery work products.

From the Supplier’s point of view, sources of Waste are the accumulation of technical debt they believe will impede future growth of the system, the loss of satisfaction with the system, or the development of features that are not required. One interview subject characterized the Supplier
view of *Waste* in a story about unnecessary effort to create an internal tool that was inferior to what could be obtained commercially off the shelf:

“there was so much ah maintenance required – that ah – you see in - in ah <COMPANY> we have this nothing vetted here since – meaning that no tool is good enough. So, we’ve – it’s - it’s ridiculous the amount of in-house tools that we build. Like we built our own framework for PNR and we built our own driver for PNRs when there are – there are actually companies making money and knowing how to do this right” – Site 3 Subject 7 320:324

A characteristic of unexpected *Waste* is *Surprise*, when delivered *Work Products* or the response to the delivered *Work Products* is significantly different from what was expected. This tended to occur when there was reduced *Communications* between *Suppliers* and *Acquirers* and *Cycle Times*. This was evident in organizations that tended to use written specifications as the primary means of *Communication* between *Acquirers* and *Suppliers*. An extreme example of significant and unexpected *Waste* resulting in *Surprise* occurred at study Site 2:

“So the final thing I think that really did it was the—for want of a better word—the lying that occurred from development management to the executive. They were telling you—they were telling the executives it's all fine. It's all on plan. Everything's going okay. And so the exec believed them. And ah—yeah—then it all kind of went very wrong when-when it was realized that we were just nowhere near” – Site 2 Subject 4 171:182

Later in the same interview the subject observed:
“I don't think we communicate that as well as we should. I think we've surprised the exec a couple of times by getting rid of things or adding new different things—which haven't been agreed to” – Site 2 Subject 4 310:312

The result of significant Waste and Surprise can be far reaching and dramatic. In the story from Site 2, the resulting Waste was so significant that it almost left the company without its next generation system and the ability to move forward in its market. At Site 3, a Surprise cost several individuals their jobs and resulted in a re-organization.

“I mean this was - the person ended up getting fired - um the development manager - for that. So, going dark um yeah, I have and it wasn't pleasant. And we ended up getting a thing that - got crucified by (unclear) for this reason. Absolutely and they caused a number of people to get reassigned. And um I mean and some lost their job. And ah we got absolutely killed um it caught - we had to launch a special project to recover from this because this team went for so long and their managers at the time weren't willing to confront it. Um in our set of - so that's - that's just what happened” – Site 3 Subject 8 172:181

4.4.3 Multiple Instances

Reconciling Perspectives is not a single thread process, and individuals may be participating in a large number of independent instances of Reconciling Perspectives in different roles, with a variety of Scopes, and Cycle Times (Figure 16). An individual Bunkering in one instance as a Supplier may be actively engaged in another instances as an Acquirer Negotiating Consensus. For example, a team lead may be Negotiating Consensus with their product owner, at an organizational Scope
with a Cycle Time measured in weeks, while concurrently negotiating an interface for an API with local Scope and a Cycle Time measured in hours, if not minutes.

The multiple instances may be unrelated in that the areas of concern are not dependent, but the multiple instances can still interfere with one another. An individual or team Bunkering in one instance may not be sufficiently open to Communications to detect a new Perspective Mismatch or an Invalidation of a Consensual Perspectives in other instances.

The reconciliation of a Perspective Mismatch could raise another mismatch that must be resolved before the original mismatch can also be resolved, in a recursive or nested application of Reconciling Perspectives. Figure 22 shows how one instance of Reconciling Perspectives depends on another that must be resolved before the first instance continues:
Some of the interview participant stories suggest this type of flow occurs during the requirements elicitation phase when there is a high degree of uncertainty.

“So, let's - let's take an example, let's say we're developing um a different way to do exception highlighting, let's just say. Ah exception highlighting - So, in that case we cou - we know what our competitor products do. We will do some mock ups. We will talk with customers about it, do a basic workflow. Um and then we'll deliver that to developers. And then we'll talk about it
whether it's technically feasible to do. Usually we've been involved with developers up until that point. So, there's a lot of work before it goes to the developer” – Site 2 Subject 8 45:51

In these situations, the Negotiating Consensus strategy is called Scouting because neither the Acquirer nor the Supplier has sufficient knowledge to reach a Consensual Perspective. Scouting is a Negotiating Consensus strategy aimed at acquiring the necessary information to reach a Consensual Perspective. Finding and creating the knowledge required to help resolve the Perspective Mismatch may, in itself, reveal another Perspective Mismatch that must be resolved.

“I shouldn't say developed but kind of adopted a few approach—for for a technical issue we will generally try and do a spike solution. Try and try and narrow down on what is the—problem or the—what we think is challenging technical and we'll try to do spike solution. And at that point it-it's kind of giving it to the developer and making sure that the—their scope is clear” – Site 3 Subject 5 97:101

4.4.4 Tension in the Process

The variation in Communications needs between the Converging and Validating stages creates a Communications tension. Converging Perspective Mismatches and reaching a Consensual Perspective requires open frequent Communications, yet such frequent open Communications can saturate individuals and interfere with the construction of Work Products during Bunkering. During Bunkering, the Supplier prefers to minimize Communications so they can focus on creating and delivering the Work Products. Yet they also must remain sufficiently open to Communications to
become aware of information that may invalidate the Consensual Agreement or indicate other Perspective Mismatches.

This tension was evident when comparing customer facing teams, that is teams that interacted directly with customers, and internal product development teams that were buffered from the end customers by a product manager and did not have direct customer contact. The customer facing teams always seemed to be completely Exposed – what is sometime referred to as “operating in interrupt mode” – trying to cope with customer demands.

“And I think for ah certainly observing a lot of the teams around here there's no buffering at all. For sure. Especially direct from customer” – Site 3 Subject 6 640:643

For Reconciling Perspectives to Get the Job Done, an individual or team must appropriately balance their Openness between being Cut-off and being Exposed. The individuals and teams that managed to balance their Communications had a Social Dynamic that moderated the process and enabled them to find a sheltered state that allowed them to remain open to Communications without excessive distractions draining away their energy. Leaders were able to shelter individuals and teams by serving as a buffer between the Acquirers and Suppliers.

“I think - I'm generally more a part of the buffering firewall effort. I think ah you know between ah [DEV MANAGER], myself PGMs15 we tend to do a lot of that interfacing with the other teams and the rest of the company and the processes of the company, so the team can mostly you know focus on doing iterations” – Site 3 Subject 6 93:96

15 Program Managers
4.4.5 Converging

Converging is the first stage of Reconciling Perspectives, and it is entered when individuals recognize a Perspective Mismatch that potentially impedes Getting the Job Done. Converging is a significant part of managing the process of software development as one manager observed:

“Working with people as much as possible so that they kinda—you can understand what they need. You can understand what their exp—expectation are. Manage those expectations if you have to. Kind of discuss with them various options as to how you can achieve what they want” – Site 3 Subject 5 39:57

Individuals reach out to engage others and Converge their Perspective Mismatches into a Consensual Perspective. Converging has two sub-stages: Reaching Out and Negotiating Consensus. Converging transitions to Validating when a Consensual Perspective is agreed to or when the transition is “forced” because it is time to present the Job Work Products. A “forced” transition is a symptom of either a Failure to Reach Out, or Convergence Stalling.

4.4.6 Validating

The second stage of Reconciling Perspectives is Validating, where the Consensual Perspective is validated by the Suppliers creating the Work Products and delivering them to the Acquirer. Variations in the frequency and diversity of Communications clearly distinguish the Converging and Validating stages. During Converging, there is significant Communications occurring as individuals reach out to engage each other and negotiate a Consensual Perspective. During
Validating, Suppliers validate the Consensual Perspective by creating the Work Products, and there is a distinct drop in Communications as Suppliers focus on Getting the Job Done.

Validating has two sub-stages: Bunkering and Evaluating. During Bunkering, Suppliers create Work Products with minimal outside distraction or interference and are therefore less likely to want to engage in Communications. Bunkering transitions to Evaluating when the Job Work Products are made available to the Acquirer for evaluation. This transition occurs:

1. At a time consensually agreed to by the Acquirer and Supplier; for example, at a scheduled milestone or iteration boundary.

2. When the Supplier presents the Work Products to the Acquirer; for example, a delivery team member presents a completed story to the Product Owner.

3. When the Acquirer demands the Supplier present the Work Products for evaluation.

Acceptance of the Work Products by the Acquirer during Evaluating validates the Consensual Perspective and results in Accepted Work Products. Validating can also revert back to Converging as an individual may hear or observe Communications that Invalidate how they understood the Consensual Perspective. If the Acquirer is less than satisfied or declines the Work Products then the output of the Validating stage is Waste and we have objective proof that the Consensual Perspective is invalid.
Reaching Out is the first stage of Converging, and it begins when individuals reflect on the Communications they are receiving about the Job and recognize Getting the Job Done is impeded by a Perspective Mismatch. Perspective Mismatches are discovered when individuals critically reflect on what they believe they are hearing in their Communications with others (verbal or written) and realize their understanding or expectation of the Job does not match what is described by the content of those Communications. Reaching Out can occur in a number of ways and be either formal or ad hoc. Reaching Out in an ad hoc manner occurs when an individual takes the initiative to question something that, for them, does not sound right.

“Mostly because I find any issue especially let’s say a bug fix which doesn’t look like it’s super huge—when you need you know more that you know 2 days’ worth of email discussion about it—something’s off. Either the thing is way bigger and actually requires refactoring or you’re not understanding one another” – Site 2 Subject 3 270:273

Creating and encouraging the opportunity for Reaching Out is part of the intention of formally scheduled meetings such as requirements reviews and Sprint planning meetings. One manager explained the utility of a regular weekly management meeting that provided the opportunity to detect Perspective Mismatches and Reach Out:

“we have a management meeting ah every week in which we discuss – okay how are we doing to the - ah - plan?” Site 3 Subject 7 137:137

Perspective Mismatches often arise when an Acquirer asks a Supplier how long a Job will take or cost, and the Supplier’s answer does not match the Acquirer’s expectations. Perspective Mismatches
also arise when a Supplier explains their approach for implementing a feature and the Acquirer responds with “Why are you doing that?” The conversation around “why” reflects different interpretations of the development methodology and what Work Products must be created for a Job – “no we don’t have to create those documents”. It could also reflect a difference in understanding of the priority of Work Products – “no we don’t need to have those right now, we have other higher priority work”.

Once an individual recognizes the possibility of a Perspective Mismatch, he or she needs to reach out and invite the others to engage in negotiations to resolve the Perspective Mismatch. The Social Dynamic property of Personal Strength is the propensity of the individual to act when they believe a Perspective Mismatch is impeding Getting the Job Done; an individual with low Personal Strength may not Reach Out. Many of the interview subjects described other software developers as quiet or introverted individuals who are reluctant to Reach Out.

“And maybe that's another thing. Is the—is this sort of—developers aren't the most social people at times right? So they-they don't go and speak to people about their problems necessarily. Being able to—so to some extent maybe it's a lack of confidence in the—if you're seen to ask—or people seem to think that asking for help is a sign of weakness or something” – Site 2 Subject 4 99:103

Leadership can create a supportive Social Dynamic that contributes to Personal Strength which gives individuals the confidence their backs are covered and they will not be sanctioned for raising issues that may have serious consequences for the Job. They can also keep
Communications open so ideas are expressed freely and new ways of thinking are developed. Leaders are in a position to either support or block individuals who want to Reach Out.

“Communication and not being afraid to express ideas and have a discussion you know? I'll have one way of viewing something or designing something but I don't feel if—I'm afraid to say it because I'll get trampled on by somebody who's more experienced or more senior. But everyone is able to offer their view and kind of like learn—everyone can be involved in the kind of—the learning experience. And then contribute to you know—some good quality code being delivered”

– Site 2 Subject 5 558:56

Leadership also has the potential to create a Social Dynamic that stifles and shuts down individuals who express concerns about Perspective Mismatches. There were several stories of leaders “wallow their team off” from the rest of the organization and therefore cutting the team off from Communications. There were also leaders who generated a culture where honest and open Communications was either ignored or not appreciated. In one story of a troubled project, two team leads reach out to the organization’s executive with their concern that they were trying to accomplish too much in the time available. Rather than addressing the team leads’ concern, the outcome was additional work.

“myself and someone else—finally figured out that this wasn't going to plan and we actually went and spoke to the exec and said look—you know we're going to need some more time—but we're going to aim for a specific date. Even then—new scope was being added. So it was just—yeah. Everything you didn't want in a project” – Site 2 Subject 4 208:213
While it is necessary for an individual to recognize a Perspective Mismatch and have positive Social Dynamics to reach out and voice their concern to others, this is not sufficient for the Converging stage to continue because the targets of Reaching Out may choose not to engage. The Converging stage of Reconciling Perspectives cannot continue if the individuals to whom the person is reaching out will not engage in the process.

“it was very typical of projects that kind of go astray. Where people live in denial for too long. And—people who voice their concerns are kind of ignored—until it just bubbles up. I—don’t think that anybody was really quiet about it. I don’t think that anybody—who was being realistic or worked on it—like everybody could say well hey this thing is really not going very well. I just don’t think that that was percolating all the way up” – Site 2 Subject 3 follow-up 255:260

The consequence of not Reaching Out or not being able to engage others is Failure to Reach Out, and the loss of an opportunity to reach an explicit Consensual Perspective. Individuals and teams that continue to work with their mismatched Perspectives will later find that it results in Waste. One example of how Waste is created by Failure to Reach Out is in terms of the opportunity cost that is the result when Suppliers construct Work Products that are not valuable to the Acquirer.

“Or that they'll have spent like—75% of their time cleaning up an edge case when you're like—you know what—it turns out that that edge case is never to happen. Or you know they won't have enough—they'll base a lot of their actions around things-things that they imagine how the system will work but their domain knowledge is maybe not as-not as correct as it should be.” – Site 2 Subject 5
Figure 23 shows the flow through *Reconciling Perspectives* that results when individuals participating in the process *Fail to Reach Out*. Individuals are aware of the *Perspective Mismatch*, they do not get agreement to move to *Negotiating Consensus* and therefore do not reach a *Consensual Perspective* and remain stuck in *Reaching Out* until it is time to deliver the *Work Products*.

A popular metaphor for this flow is “ignoring the elephant in the room” where no one is willing to call out an obvious problem for fear of offence or violating a cultural norm.
“I could see that there were people who were not going to get something done but no one will say it—in the meeting. No—so there's a certain culture of I don't know I guess you'd call it fear or something like that” – Site 3 Subject 4 537:540

A common theme for *Failing to Reach Out* was a perception that it was culturally unacceptable to say something could not be done. This philosophy led to a self-imposed *Communications Cut-Off*, sometimes justified by a belief that “we’ll find a way”.

“It just—it was always important to say we're screwed but don't worry. We're working on it” – Site 3 Subject 4 541:542

*Failing to Reach Out* generates *Waste* because *Suppliers* are putting effort into activities that may not deliver value for the *Acquirers*.

“Certain developers are far more prone to it. And it—they seem to do it—I don't know—seem to do it more often for sure. Other developers will point out that something's gone wrong and that helps sort things out but sometimes you give someone the (unclear) or you know they find a pair and they both go dark as well. That's-that's even worse cause that means—quite-quite often what we find—it-it's technically more complex and developers will go off and actually start a massive amount of refactoring. And it's like you've got to come and tell us that you're going and doing that crazy stuff because we might just—if it's—it's blown the estimate so it's not what we said it was. Maybe it's not worth the business value anymore—for the implementation value so” – Site 2 Subject 4 70:79

*Reaching Out* transitions to *Negotiating Consensus* when the other parties agree to negotiate. This
may be as simple as agreeing to the question “Can I talk to you about this?” or as formal as placing an agenda item for a change control meeting. If the other parties do not agree to participate in the dialog, then the process will follow the *Failure to Reach Out* flow and *Reaching Out* will transition to *Evaluating* when the *Work Products* are either scheduled for, or declared ready for, evaluation.

### 4.4.8 Negotiating Consensus

Negotiating is the activity that stands out in software development because it seems everyone is always negotiating: stakeholders and developers negotiate features and budget, development teams negotiate resources, team members negotiate task assignments, and developers negotiate Application Programming Interfaces (APIs) and allocation of behaviors to modules. One interesting observation made during this study was that no-one definitively distinguished between the activities of planning, requirements gathering, analysis, and design during the interviews. All were collectively described as negotiation, so the process of software development seemed to be one of ongoing negotiation towards various forms of consensus.

*Negotiating Consensus* is a dialog between *Acquirers* and *Suppliers* intended to converge their expectations for the *Job* and create a *Consensual Perspective* that will enable them to remove the impediments to *Getting the Job Done* caused by a *Perspective Mismatch*. The *Consensual Perspective* sets the *Acquirer’s* expectations for the *Work Products* they can expect to receive, and sets the *Supplier’s* expectations for the *Work Products* they are committing to deliver. Many colloquially view negotiation as a process for making trade-offs and, while this is part of
negotiation, it can also be a learning process for both Acquirer and Supplier to share and create knowledge, thereby enabling them to discover better informed trade-offs.

Negotiation occurs at all variations of Scope, from Chief Technical Officers (CTO), senior analysts, and product managers negotiating product features with large external customers, to individual developers negotiating an interface.

The Reconciling Perspectives properties of Scope and Cycle Time, while properties of the overall process, also strongly characterize Negotiating Consensus. Many negotiations were short, measurable in minutes if not seconds. Such negotiations were frequently observed at the individual levels between developers where the Scope of these negotiations is narrow and specific; an API or the behavior of a component. The negotiations are conducted between individuals representing only their own interests, without much formality or structure. Other negotiations may be between informal groups or formally defined groups such as a Scrum team, or even between organizational divisions.

As the Scope increased, the formality of the process increased to accommodate the more complex coordination needs of the participants with established meeting times, meeting facilitation rules, agendas, action items, and follow-ups. Furthermore, as the Scope increased, individuals represented their teams or organizations in the negotiations rather than only representing themselves.

There are three broad strategies for Negotiating Consensus:
<table>
<thead>
<tr>
<th>Negotiating Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>Negotiation strategy when the <em>Perspective Mismatch</em> results from a difference in terminology.</td>
</tr>
<tr>
<td>Broadening</td>
<td>Negotiation strategy for when one of the parties needs to include the other’s <em>Perspective</em> into their own so they can understand the problem at hand from the other’s point of view.</td>
</tr>
<tr>
<td>Scouting</td>
<td>Negotiation strategy when it is agreed the gap between the <em>Perspectives</em> cannot be reconciled without more information.</td>
</tr>
</tbody>
</table>

**Table 11: Strategies for Negotiating Consensus**

*Translation* is a straightforward negotiating strategy. Participants use it when *Perspectives* are well aligned but the *Acquirer* and *Supplier* are using different terminology to express themselves. It is easy to overhear these conversation conclude successfully with “...*oh now I see what you mean!*” *Translation* is a strategy most often observed in informal short cycle negotiations; however, examples of *Translation* could also occur during more formal negotiations.

“When we talk story points to our customers, they don’t like it because it’s um abstract and amorphous to them. They don’t ah – what they want is predictability. So, for them you know they don’t care whether we’re saying, “Well, this is a 200 story point project.” What they care is um how much does the story point cost and how long does it take to build it? And how much stuff can I get for it? So, that’s the way we present it to the customer” – Site 1 Subject 1 92:100

The second strategy for *Negotiating Consensus* is *Broadening*, where an individual attempts to broaden their understanding of the *Job* by trying to understand the other’s point of view.
Common sources of impediment were software developers viewing the Job from a technical perspective and not fully appreciating the business impact of their decisions, and Acquirers not understanding the technical consequences of their demands – for example understanding how a request may increase technical debt or limit opportunities to grow the product. The ability or desire to Broaden their Perspectives certainly appeared to be a strategy commonly employed by more experienced individuals. One developer spoke of Broadening in terms of seeing the context or “bigger picture”:

“I think kind of—one important thing is the context. So if a stakeholder is only aware of their—small item—is the most important thing to them in-the world. But if they're able to see the big picture and have an understanding of what the whole big picture is then I think that allows it.” – Site 3 Subject 5 241:243

In both Translation and Broadening, one or both parties had sufficient knowledge to understand the Job, but were unable express it in a way the other understood or cared about until they used one of these two negotiating strategies to resolve the Perspective Mismatch.

Scouting was used as a strategy in Negotiating Consensus when neither the Acquirer nor the Supplier had sufficient knowledge to explain their view to the other party. The negotiation became blocked because neither side had sufficient information to answer the questions posed by others. The purpose of Scouting is to discover the information necessary to help the Acquirer and Supplier converge their Perspectives and make a decision that removes the impediment to Getting the Job Done. One interview participant describes how, by Scouting, they arrived at a Consensual Perspective:
“The way I try to focus on managing it was to—we had to kind of solve within a certain period of time that we didn't exactly know how to solve—just to generate as many kinds of options as possible. And then discuss them with the people involved—the developers—the business analysts—as many people as were relevant. And then go with the option we felt would end up working well for the client” – Site 2 Subject 5 182:186

Scouting as a negotiation strategy opened the path for a recursive flow (see Figure 22). In the process of obtaining the needed additional information, it is possible that another Perspective Mismatch will arise. Many of these recursive flows were of the form where the Acquirer and Supplier had a Perspective Mismatch about what a third party was requesting of the software development organization. This occurred frequently when a software development organization provided products to multiple clients with contradictory requirements.

Convergence Stalling is a common failure flow that occurs when the Negotiating Consensus stage stalls and does not resulting in an explicit Consensual Perspective in a timely manner. Several challenged projects that were observed or reported during this study could be explained as never reaching an explicit Consensual Perspective. Figure 24 shows the flow for Convergence Stalling, where the process reaches Negotiating Consensus but never progresses beyond this stage until the Work Products are requested for evaluation.
This was a common pattern in many stories of challenged projects where timely decisions were not being made and people continued to create Work Products without reconciling their Perspective Mismatch. During this study, I observed several planning meetings that were not Converging to a Consensual Perspective; rather, they seemed to be “going around in circles” hashing and re-hashing the same problems. The more technically-minded participants were frequently “diving into the details” and leaving other participants behind – effectively cutting the others off from the conversation. At Site 1, for example, a “tiger team” assembled to resolve a severe performance issues was teetering into Convergence Stalling debating technical minutia until someone took Leadership and began driving the agenda forward to a Consensual Perspective. Several interview participants told stories of stalled decision making on projects and, combined
with my observations of these rudderless meetings, it is easy to see how Convergence Stalling results in Waste.

At Site 2, there were many historic stories of a large challenged project where many of the issues could be diagnosed as Convergence Stalling because decisions were not made in a timely manner, or participants did not share the same Perspective of the decision. One new technical lead, who was dropped into a struggling team, recounted his experience:

“I got everyone in a room and said—okay who understands multi-catalogue and multi-store? And no one did. No one put their hand up. No one had a clue. So I just white-boarded it out. And that's my understanding of not having worked on the project for 4 months like the majority of them. So no one had the big picture. No one. No one knew where we were headed. No one had the—conceptual view that—oh this is where my bit that I'm working on fits in. And no one spoke to anyone else. So how would they know? And it was just very—very strange” – Site 2 Subject 4 470:475

One reason negotiations may stall is there is a Social Dynamic where no-one is willing to make a decision. During a retrospective of a near failed project, we observed exemplars of these situations where a project appeared to have “wandered through the desert” for six months without creating a true Consensual Perspective and there seemed to be a reluctance to face up to the situation. According to the project post-mortem, the project began to get back on track when various individuals “filled the leadership vacuum by stepping up to the plate” and began making decisions.
Negotiating Consensus may also stall because individuals explicitly choose not to engage because they do not want to compromise on a Consensual Perspective. There were stories of individuals or teams who deliberately stalled negotiations to protect their Perspectives, by limiting or even cutting off Communications. While there are numerous anecdotal stories of project managers, stakeholders, and other Acquirers bullying developers into accepting unreachable and unattainable features and schedules, we clearly observed and heard numerous stories of developers bullying managers and stakeholders. They protected their agenda by cutting off Communications.

“They completely disconnected themselves and what they did was they um they gave very low ball estimates - got them - got ah a major re-architecture approved by developed, based on these estimates and then by the time - and then they came and said, “Oh, by the way, this is going to take way longer.” And um it was a black hole” – Site 3 Subject 8 165:169

Scouting Blowout is a type of Convergence Stalling that occurs when Scouting is not constrained and does not produce the information necessary to attain an explicit Consensual Perspective in a timely manner. The Supplier either continues to make perceived progress to Getting the Job Done by working with their Perspective of the Job, or remains stuck in Scouting and fails to deliver the Job.

“So yes we’ve definitely had the time boxes blown out—by people just digging into things and then not making the progress we expect and sometimes we don’t spot that in time.” – Site 2 Subject 4 56:58

In some situations, individuals with greater Personal Strength averted Convergence Stalling by stepping in to fill the Leadership vacuum and obtain an explicit Consensual Perspective. There
were many instances of leaders forcefully driving an agenda where the intent was not to force acceptance of the leader’s choice but to make a decision in a timely manner, even if that decision was to acquire more data and meet at a later time. This is an example of Leadership maintaining a rhythm that leads to a decision in a timely manner. This is exemplified by Site 2 Subject 4’s story of individuals stepping up to recover a near catastrophic project caught in what could best be called “leaderless drift”:

“I don't think—without that step up in leadership—and the other team lead—once he figure it out as well stepped up as well. And without that—I don't think I be sitting here talking to you—not in this company anyway. I think it would have been—if I'd carried on—it-it just never would have been released. So it—stepping up—getting everyone to talk—getting everyone focused.” – Site 2 Subject 4 452:458

Negotiating Consensus transitions to Validating and Bunkering when a Consensual Perspective is reached. The Consensual Perspective is based on a shared assumption by all participants they now have a shared Perspective of the Job and that the impediment created by the Perspective Mismatch has been removed. If a Consensual Perspective is not reached in a timely manner then the process will follow the Convergence Stalling flow and Negotiating Consensus will transition to Evaluating when the Work Products are either scheduled for, or declared ready for, evaluation.

4.4.9 Bunkering

Bunkering represents the quiet “heads down” focused working stage of Validating. Using the knowledge gained from the Consensual Perspective Suppliers validate the Consensual Perspective by creating the Work Products they believe will satisfy the Acquirer’s Job requirements. Unlike the
Reaching Out and Negotiating Consensus stages, where there is free and open Communications, there is a distinct drop in Communications and interactions between Acquirers and Suppliers during Bunkering. This “quiet time” is built into software methods like Scrum, which explicitly cycles between periods of high Communications and periods where the team is shielded from interruptions. One interview subject viewed this as a measure of agility:

“I think it's partially—it's obviously not a controlled environment cause there's a lot of external interference. So I think that's the part where there's still differences in terms of not being fully agile” – Site 2 Subject 3 follow-up 98:100

Whether quiet time is baked into the methodology or not, the opportunity for quiet time was seen as essential to Getting the Job Done. One of the most common complaints that came up during informal conversations was of the form “I could get this done if it weren’t for all these <expletive> meetings”. Once an individual or team believes a Consensual Perspective has been reached, construction of a Work Product should proceed relatively undisturbed until it is done. Individuals and teams went through considerable efforts to create this quiet time, from trying to establish “quiet hours” when no one was supposed to interrupt team members, to physically separating themselves from the Ecosystem by hiding in a conference room or working at home.

The quiet, undisturbed environment created during the Bunkering stage is what most developers believed boosted their productively:

“we were in a lot of ways kind of flying under the radar..... Therefore we didn't have to answer to anybody—so directly. So we got a good chunk of time running like that on our own. We got a lot of stuff done” – Site 3 Subject 4 76:80
During Bunkering, new information may become available that Invalidates the Consensual Perspective. The Consensual Perspective is subjective, and is tentative because it was reached when all parties originally Negotiating Consensus agreed that the Communications they were hearing did not conflict with their Perspectives of the Job. New information could change that agreement and Invalidate the Consensual Perspective. In this situation, the process shifts back to Converging, and the process of Reconciling Perspectives begins again following an iterative flow (Figure 25). Perspective Mismatches regarding Work Product quality were frequent problems during Bunkering where the consequences of reducing quality to get a Job out on time were not understood and frequently resulted in an Invalidation of the Consensual Perspective.

Figure 25: Reconciling Perspectives: Iterative flow leading to Accepted Work Products
**Invalidation** is not a failure; it is an indicator that individuals or teams are remaining sufficiently connected to the *Ecosystem* during *Bunkering* they can be aware of new information that changes how they perceived the *Job*. One interview participant was well aware of this, and actively looked for information that may **invalidate** her assumptions:

“I also realize that my requirements are not perfect—so that even though I provide use cases and different UI that there are different scenarios that I don't account for necessarily and I try not necessarily to wait for QA to get there. But by doing kind of daily—you know small kind of regression testing in areas that they’ve done it allows me to see maybe some holes in my requirements and how I can fill those before the iteration's done” – Site 2 Subject 3 194:203

An individual or team that is not well connected to the *Ecosystem* will be oblivious to events occurring around them and will risk creating *Waste*. This conversation may have unfolded in a substantially different manner had it occurred later in the process and this is what differentiates the *Iterative Flow* from failure flows such as *Failure to Reach Out* and *Convergence Stalling*.

If individuals and teams take their need for quiet uninterrupted time to the extreme, and *Communications* are **Cut-Off**, then they risk missing *Communications* that **invalidate** the *Consensual Perspective*, causing the process to follow the *Isolation* flow (Figure 26) and generate **Waste**. One interview subject explained this in a story of two co-located developers who cut themselves off from each other after reaching a *Consensual Perspective*:

“Like two people who work together and they maybe need to integrate some stuff with the rest of the team but they kind of pigeon-hole themselves and you know only talk to each other but not really interact so much with the rest of the team members and the end of it iteration approaches
and you realize that they've been working on something that's actually not—you can't integrate with the rest of the team—with their code. So—definitely kind of that isolated lack of talking to other people” – Site 2 Subject 3 282:288

![Diagram of flow path](image)

**Figure 26: Waste Resulting from Isolation Flow**

The similarity in the flow path between the Simple Linear Flow that leads to Accepted Work Products and the Isolation flow that leads to Waste reinforces the assertion the Consensual Perspective is tentative until it is validated with Accepted Work Products. One interview participant described how individuals isolated themselves by “disappearing” and the consequences of that isolation:

“So—and quite often some of the things we've seen where developers do just disappear—they come back and their solution's just horrible and causes rippling effects everywhere. Because the scope expanded so much that it actually did expand to include the whole system effectively in the
way that the changes they wanted to put in rippled out. Cause they can't do that level of change and do all the automated tests and do everything else to make it a safe change. So the system's obviously not ready for it.” – Site 2 Subject 4 108:114

One source of Isolation that was commonly cited was caused when teams organized by function (e.g. analysts, developers, testers) threw partially completed Work Products “over the wall” but otherwise had no Communications with each other.

“in a lot of cases there was maybe the main knowledge that needed to be there to determine what the outcome should have been and there was not. There really wasn't support to figure that out. The conversations were not happening also as well. Like the QA team was sitting—completely separately from the developer team. So—the developers would sit in one room—the QA team sat in another room. It was a complete like throw it over the wall and then it falls over the wall and—it's not good and it gets thrown back kind of mentality” – Site 2 Subject 3 follow-up 278:282

A variation on Isolation is Hitting the Wall. Hitting the Wall is triggered when the Supplier either realizes they will not be able to deliver the Work Products when they are expected or with the expected level of functionality or quality. The Supplier engaged in a Heat Calculation where they decide which of the undesirable options remaining will cause the least “heat”. They needed to decide whether to move back to Converging and Reaching Out and re-engage the Acquirer, or stay in Validating and push through to Evaluating hoping for minimal Surprise despite the Waste.

During project status meetings, I observed teams debating whether they would catch less heat
from pushing out a deficient product or from releasing the product late. Some groups were up
front and decided to take the “heat” now rather than later.

“delivering bad news all the time, but it’s pay me now or pay me later. So, you either do it now
and take lesser heat or you do it later and take a bigger hit. Because then you’ve lost your
credibility as well as not deliver on the feature” – Site 3 Subject 8 188:191

In this situation, the flow becomes like an Iterative Flow and loops back to Converging and
Reaching Out in an effort to engage the Acquirer. Others believed they were in time-critical
situations, and decided pushing forward was a more appropriate course of action, and that the
opportunity to remediate the issue would come later, if at all.

“Well then it would literally just go in as something that was more or less broken or missing
functionality. Then it would go into maybe the testing phase and if it—if it did get caught later on
there or enough people were unhappy then they'd—they'd just put extra pressure on—throw it
back to the development team” – Site 2 Subject 5 199:207

In this situation, the process will progress linearly towards Evaluating, where it is likely that the
Work Products will be evaluated as Waste.

The opposite extreme of Cut-Off is Exposed, where individuals or teams indiscriminately engage
in all forms of Communications, which impedes their ability to focus on Getting the Job Done.
Being Exposed exacerbates Perspective Mismatches related to priority and schedule when Suppliers
have low Personal Strength and are reluctant to say “no”. The many layers of decision makers in
the Ecosystem means not everyone shares the same Perspective on Job priorities, and we observed
people directly approaching developers to get what they perceived as a higher priority Job done. The result is Suppliers who have far too much work in progress, reactively change priorities, and are constantly interrupted by escalations. Individuals and teams that could not say “no” became, as one project manager put it, “their own worst enemies”:

“But you're your own worst enemy when you do that because the next thing you know, the BA's asking this developer for an estimate on this and can you do that? And oh I found this new scenario and last week” – Site 1 Subject 1 305:308

Every meeting invite, every conversation, every e-mail, and every IM results in an interruption requiring rework, or an issue that must be explored, or even a diversion to working on an unrelated Job. Individuals who are Exposed and unable to focus on Getting the Job Done because they say “yes” to everything and cannot focus on Getting the Job Done.

“They were just working together, the team was very new at that point in time and they ah didn't really ah understand the impacts of taking on those types of changes. And so we came to the - ah - release date and I still had at least three weeks' worth of work left” – Site 1 Subject 1 267:271

The above conversation is an example of what is often referred to as requirements creep, where the Acquirer is asking the Supplier for more than what was originally agreed to. This flow is essentially an uncontrolled Iterative Flow, because the Consensual Perspective is repeatedly being Invalidated. This repeated Validation is often referred to as “churn”; the inability to filter Communications leads to situations where everyone is negotiating and no one is making progress.
The consequences of being Exposed during Bunkering are evident at the end of projects when individuals and teams are panicking to remove defects. It is common to see priorities rapidly changing in reaction to discovery of new defects, and partially finished Work Products being put aside to work on higher priority defects. The result is defect opening rates exceed defect close rates and the Job simply does not get done. Software developers often refer to this situation as “bug fix hell”:

“...most of the developers were not happy about the way things were (unclear). It was - it got to this what we call a bug-fixing hell of a state. There was so many bugs and we were trying to fix the bugs a couple of months. There was long days when we were trying to get this working” – Site 2 Subject 1 205:212

Social Dynamics moderates Reconciling Perspectives by managing Communications to appropriately balance Openness between Cut-Off and Exposed. Individuals and teams must remain sufficiently open to Communications to be able to detect new Perspective Mismatches while not becoming Exposed to rapid changes and vagaries of the organizational Ecosystem that can derail focusing on the Job at hand and interfere with Getting the Job Done. This need to find balance between being Cut-off and Exposed highlights the inherent tension of the Reconciling Perspectives process:

“Um ah and I think we still haven't got to a good place where we get that balance between you know short term, heads down focus, but also a sense of where the vision of things are going” – Site 3 Subject 6 227:229

Positive Social Dynamics facilitates balance in Communications by Sheltering individuals or teams from the turmoil of the organizational Ecosystem. In this study, we frequently observed leaders
protecting their teams (sometimes aggressively) from exposure by standing between them and the Ecosystem while trying not to cut the team off completely from the organizational Ecosystem. One participant referred to the leader taking on a protective or buffering role like a “firewall”:

“Sometimes it's a firewall kind of role. And our (unclear) plays that role quite a bit as well - a ton. He insulates all of us from a lot of that stuff” – Site 3 Subject 7 52:54

A leader in this role is able to stay open to Communications and yet offer some measure of stability to those who are trying to Get the Job Done. The ability of leaders to stay open to Communications was influenced by the leaders’ beliefs about their responsibilities in the organizational hierarchy:

“the higher up you move in the chain because you're expected to-to take on more—more responsibility right? So you'll only get—you only really get shielded so far in as much responsibility as somebody has given you. If you have a reasonable amount of responsibility you won't be needed—you won't need to be shielded that much” – Site 2 Subject 5 521:525

Bunkering transitions to Evaluating when Suppliers present the Work Products to the Acquirers for their acceptance. The presentation of the product is triggered either by a previously agreed scheduled milestone, a demand from the Acquirer, or by the Supplier deeming the Work Products to be ready. Bunkering may also transition back to Converging, when the Consensual Perspective is Invalidated, which can result in an iterative flow. A severe form of Invalidation of the Consensual Perspective is Hitting the Wall, which happens when, during Bunkering, the Supplier receives Communications causing them to realize that the Consensual Perspective is not valid and that there
is insufficient time or limited options available to Get the Job Done. The team engages in the Heat Calculation to decide whether to continue to Evaluating or return to Converging:

“we're - we're close enough that we can see the end and we can start to panic. .... And those essentials are the things that are now left and if one of those is say missed, it's literally, we can't ship that part - this part of the product. So, I mean, there have been different strategies looked at” – Site 3 Subject 1 469:478.

### 4.4.10 Evaluating

During Evaluating, the Supplier presents the Work Products to the Acquirer and solicits their approval. Evaluating may be represented by the Acquirer informally stating, “Ok it’s good” or there may be a formal ceremony for transitioning and Evaluating a Work Product. After the Work Products are accepted, the process is complete and any Perspective Mismatches have been reconciled. Only when the Work Products are accepted is there objective evidence that all involved were operating from a converged Perspective. If the Work Products are declined by the Acquirer, or if the Supplier is dissatisfied with the Work Products, then Waste is created.

If significant unexpected Waste is created, then the Acquirer and Supplier may be Surprised. The intensity of a Surprise is generally proportional to the perception of Waste in the delivered Work Products. We observed many situations in which lengthy Bunkering stages directly influenced Surprise. Reconciling Perspectives processes with long Cycle Times have a greater potential to result in Surprise or in larger Surprises than processes with shorter Cycle Times. A near catastrophic Surprise occurred at one site where Supplier and Acquirer teams effectively cut each other off for over six months.
The perception of *Waste* and *Surprise* is not just an *Acquirer* phenomenon, and the delivery of low quality *Work Products* may result in the *Supplier* seeing the *Work Products* as *Waste* because their need for achieving technical excellence is not satisfied. In such situations, the *Supplier* may unilaterally go back to the *Bunkering* stage and engaging in “stealth” refactoring to bring the *Work Products* up to a desired level of technical excellence, despite the acceptance or even the enthusiastic endorsement of the *Work Products* by the *Acquirer*.

### 4.4.11 Cycle Time

*Cycle Time* is an observable property of *Reconciling Perspectives* and is the interval of time from when *Reaching Out* begins until the *Work Products* are presented during *Evaluating*. *Cycle Time* tends to vary with *Scope*, with larger *Scopes* tending to have longer *Cycle Times*. For small ad hoc conversations between individuals, the *Cycle Time* could be expressed in minutes; whereas, with much larger *Scope*, the cycle time may be expressed in hours, days, or even months. Shorter *Cycle Times* tend to reduce *Waste* and *Surprise*. Close proximity of participants also tends to reduce *Cycle Time*, whereas geographic distribution tends to increase *Cycle Time*:

“solving those in-house is usually the turnaround is much faster. It's much easier to go over and speak to somebody” — Site 2 Subject 3 254:259

One participant used the “rule of 3” to describe the increase in cycle times resulting from geographic distribution. A local issue that could be held in 3 minutes between co-located individuals may take 3 hours if the individuals were on different floors, 3 days if in the same region, and 3 weeks if they are in different countries. While greater geographic distribution
introduces *Communications* impediments that reduce the frequency of *Communications*, geographic distribution is not the only factor influencing *Cycle Time*.

“But it's amazing how much that increases it right there for a minimum - not 30 seconds but 30 minutes to three hours. Just that little bit of a barrier, you know” – Site 3 Subject 6 309:313

*Social Dynamics* influence *Cycle Time*; positive *Social Dynamics* tends to result in individuals and teams who respond quickly to *Communications* and reduce *Cycle Times*, whereas negative *Social Dynamics* tends to result in deferring *Communications* which subsequently increase *Cycle Time*.

“I think first off—in PS there seems to be more of a lag between—the time the developers develop a component and the time it gets actually QA'd. There can be a lag from—anywhere from a week-2 weeks-3 weeks—a few months. In PD they-they tend to operate in quick turn around on-on QA. They’ll operate in little groups of a couple developers to a QA and they’ll get quick QA feedback—as to the quality or if it's meeting the-the spec or if it's going to play nicely with other parts of the system. In PS there's a much longer lag because they'll just say okay everyone develop these pieces” – Site 2 Subject 5 222:232

### 4.4.12 Scope

*Scope* is an observable property of *Reconciling Perspectives* and is the number of individuals who have a direct interest in the outcome of the *Perspective Mismatch*. While an individual may initiate *Reconciling Perspectives*, the outcome of the process may have large numbers of stakeholders who must either directly or indirectly participate in the process. *Scope* can be classified into three broad categories:
<table>
<thead>
<tr>
<th>Scope Category</th>
<th>Covers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>Two individuals who have an interest in the outcome of a Perspective Mismatch.</td>
</tr>
<tr>
<td>Team</td>
<td>Three or more individuals from the same identifiable team have an interest in the Perspective Mismatch.</td>
</tr>
<tr>
<td>Organizational</td>
<td>When individuals from multiple teams have an interest in the Perspective Mismatch.</td>
</tr>
</tbody>
</table>

Table 12: Scope Categories

At the organizational Scope, representative individuals or teams act on behalf the organization rather than the entire organization directly engaging in Reconciling Perspectives. For example, a single Business Analyst (BA) may act on behalf of an Acquirer organization and a small group of senior team leads may act on behalf of the development organization. Generally, Cycle Time increases with Scope for two reasons. First, the size of the Perspective Mismatch in terms of complexity of the issues involved requires more time and effort to negotiate. Second, there is a coordination overhead associated with getting a larger numbers of individuals to come to consensus.

There is more formality with larger Scope and larger Scope may slow Communications because of greater Communication distance. Larger Scope also requires greater Personal Strength on the part of individuals to Reach Out when they discover a Perspective Mismatch.
4.4.13 Communications

Communication is an observable property of Reconciling Perspectives and is the exchange of information between two or more participants. Communications is how individuals and teams stay connected with each other and with the organizational Ecosystem. Communications is the foundation of software development, and it is impossible to imagine a useful software development effort that Gets the Job Done without Communications. Acquirers need to express to Suppliers their expectations of a Job, and Suppliers need to express their expectations to the Acquirer. Without Communications there is simply no way for two or more individuals to engage in the process of software development. One project manager summed this up succinctly:

“just communicate, communicate, communicate” – Site 1 Subject 1 follow-up 32:32

Communications manifest in many forms, from informal “water cooler” conversations between individuals, and ad hoc team meetings, to formally scheduled meetings that may be electronically mediated teleconferences and span the globe. Communications may be delivered verbally, such as in face to face conversations, or may be written and exchanged using electronically mediated tools such as e-mail or instant messaging.

Communications permeates all stages of Reconciling Perspectives. The process starts when an individual reflects on Communications and realizes their interpretation of the Communications content conflicts with their mental model or Perspective of the Job. The less Communications there are, then the less opportunities there are to recognize a Perspective Mismatch and engage others in Reconciling Perspectives. As one subject summed up the root cause for a failed project:

“There just weren’t enough conversations taking place” – Site 2 Subject 3
Volume and Diversity are two Communications properties. Volume is the frequency and the quantity of relevant information exchanged. Low Volume Communications were situations where people simply did not talk to one another, or talked frequently about social trivia that was irrelevant to the Job. Communications Volume is influenced by the channel used; face-to-face Communications enable a very high volume of information to be exchanged, while electronically mediated Communications limit the Communications Volume.

“...and you would find that definitely if you were relying on communication over the phone and ov—via email—you—it—you have to infer a lot of information about what the person means. You know the old whole aspect of ton—tonality and emails and you know knowing wha—how a person feels about something over the phone is difficult—more difficult to read if you don't know them.” – Site 2 Subject 5 86:102

Diversity is the variation in the sources of Communications. A low level of Diversity is when individuals or teams Communicate only with their nearest neighbors, often like-minded individuals. An individual may enjoy a high volume of Communications within their immediate team, but may be reluctant to talk to those beyond their team. We frequently observed situations where team members talked freely among themselves yet used slow low Volume Communications media such as e-mail to communicate with someone down the hall. A high Diversity of Communications results when an individual pursues or engages in a dialogue with others in different groups, in different functional areas such as marketing or sales, and in different geographic areas. Lack of Diversity in Communications can be a form of Cut Off and often led to situations where an engineering Perspective drove a Job and resulted in Waste and Surprise for
Acquirers. Many Experienced individuals made considerable efforts to create personal social networks within the organization to Diversify their Communications:

“You have to go out and find them. Um, I have a – and I’ve been doing this here for about ten years now, so I have um a set of contacts I trust in the field. So, you go out and find them, you qualify them. And some of them you ignore. Some of them you pay – you pay attention to so – nobody really finds them for you” – Site 3 Subject 8 101:104

Communications is inversely related to the distance between individuals and teams; the greater the distance between the communicating parties, then the lower the frequency of Communications. This decrease in frequency is counter-balanced by increasing Diversity. While distance is often considered measured in geographic units such as kilometers, once communicating parties were not co-located then the relevant measure of distance is time-zones:

“Ah, it’s the usual kind of thing when there’s any distance and as the distance increases, it’s amplified. Is - is – it is – you don’t hear back you know. You run into a problem. You raise the problem. It maybe goes through email or through you know trouble ticket kind of bug tracking type system. Then a reply back the next day maybe if you’re lucky or two days and then it’s just a little bit misunderstood, and then it goes back. They finally come to a resolution. It gets fixed. Then you realize the fix – okay there was something totally everybody misunderstood” – Site 3 Subject 6 285:292
Close proximity encourages frequent Communications; however, it also increases the opportunity for becoming Exposed. In many situations where an individual Supplier or Supplier team worked in close proximity to their Acquirers, there was greater opportunity for distracting interruptions. This type of distraction was clearly visible and a significant source of frustration for customer facing teams, especially teams which were co-located with their Acquirer. Similar pressure was not observed on internal engineering teams (non customer facing teams) during this study. In the absence of a Leadership that could Shelter the team, responding to Acquirer requests could quickly drain much of the team energy. One interview participant captures this when describing the trade-offs of being co-located with their customer:

“And that's one of the things - that's one of the risks you take on when your customer is right there, interacting with your developers every single day. I don't want to be there hovering over all of their interactions. I want them to have direct interactions. And it's a risk that I think is endemic to have the role because customers always want more than what you can give. And either um so there's - there's always that little bit of tension there, I think. Sometimes they're very good and they don't. Sometimes they're not so good and they try to go around you. (laughing) It happens. But you want them to have the open relationship - you know the open um communication” – Site 1 Subject 1 361:368

During Bunkering, Suppliers prefer fewer interruptions and distractions so they can focus on Getting the Job Done. There is a significant and observable difference between the level of conversation during the Converging and Validating stages of the process. Reconciling Perspectives therefore depends on the ability of an individual or team to appropriately manage their
Communications for their stage of work. Positive Social Dynamics enable individuals and teams to manage their Communications, reducing the likelihood of either reducing their Openness to the point of being Cut-off, or of becoming overwhelmed by being Exposed.

“Well I've tried to—as far as possible—kind of shield the team from—from the pressure. I—cause I know it's the worst situation to be in if you're trying to figure out a problem and somebody's telling you—I need this done yesterday. It's the least enjoyable situation to be in” – Site 2 Subject 5 491:494

4.4.14 Social Dynamics

A recurring theme from the interview data is the belief software development is about working with people, and that it is a social process.

“Well working with people as much as possible so that they kinda—you can understand what they need. You can understand what their exp—expectation are. Manage those expectations if you have to. Kind of discuss with them various options as to how you can achieve what they want” – Site 2 Subject 5 39:41

Social Dynamics is the capability of an individual or team to manage their Communications and balance the tension between the Converging and Validating stages to Get the Job Done. Social Dynamics are a set of properties that moderate Reconciling Perspectives such that individuals or teams do not become Exposed to excessive distractions or Cut-Off Communications and isolate themselves from the Ecosystem. There are four properties of Social Dynamics:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal Strength</td>
<td>An individual’s ability to reach out, and also to know when to say no.</td>
</tr>
<tr>
<td>Experience</td>
<td>The knowledge gained over time working within a specific Ecosystem, and</td>
</tr>
<tr>
<td></td>
<td>the familiarity an individual develops with others.</td>
</tr>
<tr>
<td>Openness</td>
<td>The receptiveness of an individual or team to Communications and</td>
</tr>
<tr>
<td></td>
<td>negotiation.</td>
</tr>
<tr>
<td>Leadership</td>
<td>The ability of individuals to establish a culture that helps individuals and</td>
</tr>
<tr>
<td></td>
<td>teams manage Communications and moves the process forward.</td>
</tr>
</tbody>
</table>

**Table 13: Properties of Social Dynamics**

*Personal Strength* and *Experience* are personal traits of individuals for managing Communications while *Leadership* is the ability of individuals to facilitate a dynamic that enables others to more effectively manage their Communications. During interviews people spoke of leaders supporting them and “covering their back” or how leaders sheltered the team by taking on interruptions while the rest of the team enjoyed quiet focused time to Get the Job Done. *Leadership* can also facilitate a dynamic that either cuts individuals and teams off from the Ecosystem or leaves them Exposed.

The four properties of Social Dynamics are not independent of each other, and variations in one property influence the others. For example, *Experience* influences *Personal Strength*; the more *Experience* an individual has, the greater their *Personal Strength*, and the greater will be their tendency to Reach Out. However, a sudden catastrophic *Experience* can dramatically reduce *Personal Strength* and reduce the tendency to reach out. This phenomenon was clearly observed
after the release of a severely challenged project, when the formerly gregarious project manager now tended to hole up in his office rather than engage with the team.

*Social Dynamics* are either positive or negative in terms of the moderating influence on *Reconciling Perspectives*. Positive *Social Dynamics* is an interplay of *Personal Strength*, *Experience*, *Leadership*, and *Openness* that leads to flows that result in *Accepted Work Products* because individuals and teams use *Reconciling Perspectives* to quickly remove impediments to software development that result from *Perspective Mismatches*. Positive *Leadership* and *Openness* means individuals and teams are open and connected to the *Ecosystem*, and are aware of *Communications* that may indicate a *Perspective Mismatch*. Greater *Personal Strength* means individuals *Reach Out* when they suspect a *Perspective Mismatch* and are open to engaging to *Negotiating Consensus*. *Leadership* helps move the process forward and prevents *Failure to Reach Out* or *Convergence Stalling*. Resulting *Cycle Times* are shorter, and *Suppliers* are able to present *Work Products* for acceptance before the differences between what is delivered and what the *Acquirers* expect becomes too great.

Negative *Social Dynamics* result in flows that lead to more *Waste* and *Surprise*. Individuals with low *Personal Strength* do not reach out when they suspect a *Perspective Mismatch*, do not have the *Support* of their *Leadership* which can result in a *Failure to Reach Out* flow (Figure 23). The process can stall during negotiations when no one in *Leadership* steps up to move the process forward which can result in *Convergence Stalling* (Figure 24). Individuals and teams may cut themselves off from the *Ecosystem* and be unaware of *Perspective Mismatches* and can fall into an
Isolation flow (Figure 26). An individual may lack the Personal Strength to say “no” or a team may be left Exposed and unable to focus on Getting the Job Done and follow a Saturation flow.

4.4.14.1 Personal Strength

Personal Strength is a personal trait that is elusive to define beyond characterizing it as an individual’s ability to Reach Out and engage others when they suspect something is amiss. It is also the ability to avoid becoming Exposed and knowing when to say “no”. During this study, there were many references to developers lacking social skills and the ability to engage others.

“You know there's certain logical conclusions that they can make but I think in general with lots of devs they're kind of more um—I'm totally missing the word um—kind of in terms of social skills they're mor—they keep more to themselves” – Site 2 Subject 3 332:334

Personal Strength enables positive Social Dynamics because stronger individuals are more likely than weaker individuals to Reach Out when they suspect a Perspective Mismatch and avoid the Waste created by a Failure to Reach Out. Individuals with higher Personal Strength are more likely to avoid becoming Exposed and therefore avoid creating Waste and Surprise through Saturation. Personal Strength is partly an innate trait of the individual and partly influenced by other Social Dynamic properties such as Experience and Leadership.

The lack of Personal Strength enables negative Social Dynamic because an individual is either unwilling to reach out when they suspect a Perspective Mismatch, or they cannot focus on Getting
the Job Done because they are Exposed. Personal Strength has limits, however, and excessively strong individuals may also be seen as bullies. While they are personally strong, their level of Openness is near Cut-off and their approach to negotiation is “take it or leave it”. During this study, it was clear that some of the higher strength individuals were very much regarded as near bullies.

Personal Strength is not static, and is influenced by Leadership and individual Experience. Leadership can increase Personal Strength by creating a supportive environment for individuals that gives them the confidence to know it is safe to Reach Out. Experience can increase Personal Strength because it informs individuals where potential Perspective Mismatches may exist, and with how well Reaching Out was received in past. Overtly negative Experiences can also reduce Personal Strength for individuals who have been involved with the creation of significant Waste and Surprise.

4.4.14.2 Experience

Experience is knowledge gained over time throughout one’s career. It includes knowledge gained working in specific Ecosystems and the familiarity that an individual develops with other individuals within and outside of their immediate team. The more Experience an individual has, the more likely it will be that they have developed a set of personal patterns that help guide them through the process of software development, and the more likely it will be that they are “plugged into” their organizational Ecosystem. Experience enables positive Social Dynamics because individuals can understand current problems as instances of a pattern of problems they have seen before. Experience creates an incentive for individuals to become personally stronger
and Reach Out when they suspect a Perspective Mismatch because, over time, they have become aware of the consequences of not Reaching Out.

“you know if you develop a bit longer and you get a bit more experience then you actually realize that asking for help is probably the first thing you should be doing. Because it'll (unclear)—just unjar you and-and get you going again. And you shouldn't be afraid to admit the things that—you know. And that stops you going dark and wasting time basically. Because it'll (unclear)—just unjar you and-and get you going again. And you shouldn't be afraid to admit the things that—you know. And that stops you going dark and wasting time basically” – Site 2 Subject 4 104:108.

Experience creates a set of personal patterns that gives an individual a set of triggers or cues they use to quickly recognize the existence of a Perspective Mismatch. One interview subject summed it up succinctly by claiming they “could smell” when things were going off track and they knew they needed to begin Reaching Out. Throughout the observations, and in the interview data, there is a pattern of more Experienced individuals pro-actively probing others to discover Perspective Matches when something didn’t “smell right”.

Experience also helps reduce the probability of becoming Exposed because it informs an individual about what they need to be concerned for and, more importantly, what not to worry about – something that one interview subject described as “messing around”.

“You have to be organized and you have to know what's in the product. You have to be in touch with what's currently being developed. Um, and you have to make quick judgment calls. Really.
You can't mess around too long. There's too many requirements. You have to have enough experience to make a quick call” – Site 3 Subject 8 128:135

Another dimension of Experience is the personal network individuals develop that Diversifies their Communications and creates a greater opportunity for them to detect a Perspective Mismatch.

“I have a - and I've been doing this here for about ten years now, so I have um a set of contacts I trust in the field. So, you go out and find them, you qualify them. And some of them you ignore. Some of them you pay - you pay attention to so - nobody really finds them for you” – Site 3 Subject 8 98:104

Personal networks were often used by a Supplier as a filter to detect and qualify a Perspective Mismatch, often by first Reaching Out to the personal network and asking “What do you think of this? Is it worth worrying about?” The personal network was often exploited during the requirements analysis either as proxy for an absent Acquirer, or to determine if it was worth escalating the suspected Perspective Mismatch to the Acquirer. The personal network may also be used as a proxy for multiple Acquirers during Negotiating Consensus, to resolve the Perspective Mismatches resulting from contradictory requirements.

While the lack of Experience contributes to negative Social Dynamics, Experience in the form of catastrophic events (e.g. Hitting the Wall) can also enable a negative Social Dynamic. A catastrophic event can result in a previously strong individual going into hiding and cutting themselves off from the Ecosystem and even from their team. For example, after the near
catastrophic failure of an important project at Site 2, the original high energy project manager was put to pasture within the organization. For the next six months, the former project manager kept to himself, until he left the organization. While this was a dramatic situation, this pattern of a previously strong individual cutting themselves off after a failure was observed on more than one occasion and at more than one site during this study.

4.4.14.3 Openness

The Social Dynamic property of Openness characterizes how permeable to Communications, and how open to compromise, individuals and teams are, and it ranges from Cut-Off to Exposed. Individuals and teams may completely cut themselves off from all Communications and therefore have little or no opportunity to become aware of feedback that challenges their Perspective of the Job. Cut-off may manifest itself in the form of the team simply not engaging in conversations, returning e-mails, or responding to invites. Cut-off reduces the opportunity for the process of Reconciling Perspectives to begin because the opportunity to receive and reflect on Communications is reduced. In contrast to Cut-off, individuals and teams may be Exposed and indiscriminately engage in any and all Communications. Being Exposed may manifest itself as simply agreeing to all requests made of the individual or team and not engaging in negotiations. Simply being agreeable does not create a Consensual Perspective, and does not reduce Waste and Surprise.

Openness enables a positive Social Dynamic when there is a willingness to respond to others who are Reaching Out, a willingness to engage in negotiations, and a willingness to make compromises and respond flexibly. Openness enables a negative Social Dynamic at its extreme points of Cut-Off and Exposed. When an individual or team Cut-Off Communications or simply
refuses to compromise, then there is no Reconciling Perspectives. When an individual or team becomes Exposed, unable to filter any Communications and responding to every inquiry and e-mail, or being distracted by other Jobs, then there is little or no opportunity to focus on Getting the Job Done.

“Oh you have it on different levels. There’s always different levels of priority. So—kind of expecting having the team meet high expectations from several different clients and work on the existing project. So that kind of detracts the attention from the project that’s ongoing” – Site 2 Subject 3 follow up 102:105

Exposed usually manifests itself as a Supplier either unquestioningly engaging in all Acquirer requests or being distracted by other Jobs. The likelihood of becoming Exposed increases if the Supplier and Acquirer are in close proximity to one another. While co-location and onsite customers may be the gold standard for collaborative software development between Suppliers and Acquirers, during this study we observed that close proximity between the Acquirer and the Supplier also increased the opportunity for the Supplier to become Exposed. Both Site 1 and Site 2 worked directly with external Acquirers. At Site 1, the Supplier team was in a field office located on the Acquirer’s premise, and Site 2 frequently sent Supplier teams to Acquirer sites. At both Site 1 and Site 2, it was a fairly common sight to see a Supplier returning to their desk only to see a yellow sticky note pasted to their monitor demanding yet another change to the Job.

The most striking example of this relationship between Acquirer/Supplier proximity and exposure was at Site 2, where the Supplier organization was split into two groups: a customer facing
services group, and an internal product development group. The product development group was *Sheltered* by the CTO and operated with a purposeful steady flow, whereas the services group always appeared to be operating in “interrupt mode” with little opportunity to push back and a tendency to work much longer hours. These situations are examples of where individuals and teams were not, or could not be, *Sheltered* by their *Leadership*.

The opposite extreme of *Openness* is *Cut-off*, where an individual or team simply does not *Communicate* at all or is not open to compromise. *Cut-Off* is not a situation where a shy *Supplier* is not engaging, because such individuals are more likely to become *Exposed*. Rather, *Cut Off* often occurs because an individual or team intentionally isolates themselves from the *Ecosystem* in an effort to protect themselves from outside interference and get something done. Often, *Cut-Off* is a method for avoiding compromise and protecting an agenda. While there are many anecdotal stories of *Acquirers* bullying *Suppliers* to agree to unrealistic schedules, *Cut-Off* was also used by *Suppliers* to bully *Acquirers*. This was a common pattern behind the late delivery or high cost of over-engineered systems.

“They completely disconnected themselves and what they did was they um they gave very low ball estimates - got them - got ah a major re-architecture approved by developed, based on these estimates and then by the time - and then they came and said, “Oh, by the way, this is going to take way longer.” And um it was a black hole” – Site 3 Subject 8 165:169 (cut-off)

*Communications Cut-Off* is a major cause for *Surprise* during *Evaluating* and, with long *Cycle Time* projects, the intensity of the *Surprise* can be catastrophic:
“There were so many massive things going on in the system all at once they were just trampling each other. And the fact that we had no demos and we had no working builds for so long—I think were just huge screaming problems that were ignored for a long time” – Site 2 Subject 3 301:307

Openness highlights the Communications tension in Reconciling Perspectives. While there is a significant advantage for the Suppliers being in close proximity to the Acquirers, this also increases the risk of becoming Exposed, and Reconciling Perspectives requires individuals and teams to balance their Openness between Cut-Off and Exposed so that individuals and teams stay sufficiently open to Communications to “catch wind” of Perspective Mismatches yet sufficiently focused to move forward and Get the Job Done.

“that’s one of the risks you take on when your customer is right there, interacting with your developers every single day. I don’t want to be there hovering over all of their interactions. I want them to have direct interactions. And it’s a risk that I think is endemic to have the role because customers always want more than what you can give. And either um so there’s – there’s always that little bit of tension there, I think. Sometimes they’re very good and they don’t. Sometimes they’re not so good and they try to go around you. (laughing) It happens. But you want them to have the open relationship – you know the open um communication. I would rather take the risk and you know get up and go, No, no, no. And, but keep talking to each other (laughing) than not because I think the outcome is better to have them talking to each other” – Site 1 Subject 1 358:368
4.4.14.4 Leadership

Leadership is a service an individual provides for others that facilitates the Social Dynamic and influences how they engage in Reconciling Perspectives. While most individuals who could be classified as “leaders” were more senior or experienced individuals only a few were designated managers. Leadership, from the point of view of this study, is a role performed by those individuals who actively facilitated a Social Dynamic. There are four Leadership roles that facilitate Social Dynamics and influence how individuals and teams engage in Reconciling Perspectives:

<table>
<thead>
<tr>
<th>Role</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting</td>
<td>Creating an environment that encourages Reaching Out.</td>
</tr>
<tr>
<td>Sheltering</td>
<td>Keeping individuals connected to the Ecosystem while protecting them from becoming Exposed (sometimes called buffering, or shielding).</td>
</tr>
<tr>
<td>Alerting</td>
<td>Preventing individuals from becoming Cut-Off by staying connected to the Ecosystem and bringing information that may indicate Perspective Mismatches to the attention of individuals and teams.</td>
</tr>
<tr>
<td>Drum Beating</td>
<td>Avoiding Convergence Stalling by moving the process forward and keeping negotiations focused.</td>
</tr>
</tbody>
</table>

Table 14: Reconciling Perspectives Leadership Roles

Leadership enables a positive Social Dynamic when a leader supports individuals by both giving them the confidence their concern is valid, and also by “covering their backs” when they Reach Out. Leadership also enables a positive Social Dynamic when it reduces the potential for becoming Exposed by Sheltering individuals and teams from distracting interruptions originating in the Ecosystem that could impede their Getting the Job Done. Conversely, the leader also reduces the
potential for individuals and teams to become Cut-Off by Alerting individuals and teams when Communications indicates a Perspective Mismatch is present that could impede Getting the Job Done. Leaders also contributed to positive Social Dynamics by keeping the process moving forward through Drum Beating and facilitating the conditions to make timely decisions.

Leadership can also enable a negative Social Dynamic when those assumed to be in a Leadership position simply abdicate their responsibility by leaving the individual or team Exposed, or when the leader is unable to balance the need for Openness with the need for focus on Getting the Job Done. There were many cases of this observed during this study, and many stories where a team effectively became a “black hole” disconnected from Ecosystem by the leader. There may not be any leaders on the team, or those assumed to be in positions of leadership (e.g. managers) may abdicate their responsibility for Leadership. This was the situation in a large near-catastrophic project failure, where those in assumed Leadership positions did not act as leaders and it was not until very late in the project that others “stepped up to the plate”.

Leadership, in the form of Support from an individual’s manager or from a respected thought leader within the organization, boosts Personal Strength and gives individuals confidence “their back is covered”. When individuals or groups believe their backs are covered, or believe that they will not be sanctioned for raising issues, they are more likely to take risks and Reach Out. Leadership also plays a role in moving the process forward by actively encouraging people to raise issues, confront Perspective Mismatches, and make decisions to resolve them.
“as long as you've been open with <leader> and treat him with respect and integrity um he will back you up. So, you're not going to get - you're not going to get backstabbed by him. And that's a very important thing. It relaxes me and I think it goes down to you - you - he gives us support. He expects - he expects you to be professional. He expects you to be professional. So, I would say that it - it allows me to take some - some risks if I know what I'm doing is right and I am open enough about it” – Site 3 Subject 8 241:246

Effective leaders Shelter (also called buffering or shielding) people from interruptions and distractions during the Bunkering stage of Reconciling Perspectives, and relieve others of some of their responsibility for maintaining the frequency and diversity of Communications.

“In a sense it's a buffer role so it allows the team to be agile without - with as much as possible without being disrupted by all of the non-agile stuff and ah - not even just the non-agile stuff but just other - I don't know other kind of horizontal initiatives that are necessary or - or exists in a big company. So, it kind of buffers the team from all that, so the team can focus on monthly iterations and it can plan a backlog and you know not get interrupted ah heavily by all of these things. So, that's the most typical kind of role I play ah with - yeah like I say the jumping into ah - ah a more direct ah in scrum role ah for certain projects as we kind of get them launched off the - off the ground. But, yeah, most of the time it's bridging between - between PIL - PLC and you know company-wide initiatives and the internal team” – Site 3 Subject 6 41:50
While *Sheltering* individuals from wasteful interruptions, effective leaders are also aware of issues that are sufficiently important to warrant *Alerting* individuals and teams because they indicated a *Perspective Mismatch* that affects the sheltered individuals.

“oh yeah—I try to shield the team from any—if there was something important—an important deadline—I'd let them know. I'd say—listen—we've got to achieve—we've got to achieve this. But I don't—I don't want you work the weekend if at all. Focus as-hard as you can—over the next 2 or 3 days. But we got to be able to you know—we'll deal with the problems next week or whatever” – Site 2 Subject 5 513:519

*Reconciling Perspectives* is a fragile process that is easily stalled by indecision and disengagement. It seemed that the most common recurring failure flows were *Failure to Reach Out* and *Convergence Stalling*. *Leadership Drum Beating* creates a rhythm that keeps the process from stalling, or restarts a stalled process. Teams that were observed making progress had effective Scrum Masters who diligently maintained the rhythm of the Scrum process and worked to remove impediments that could stall the team. For many teams, the Scrum Master was a subject matter expert and therefore the team’s (and other teams) “go to” person when an individual was stuck. It was not unusual to see a line-up forming around some of these individuals. There were also numerous situations where the process had stalled and other individual team members took it upon themselves to restart the process. Interview subjects used phrases like “drum beating” and “energizing” to describe efforts to by leaders to encourage people to *Communicate* and *Reach Out*. 
“certain developers will—they'll take on some work and they will be not as communicative as they should be. They—they will—they'll be happy to kind of go into their own zone and if you don't kind of—ping them—and take—it your responsibility to ping them every half a day or day or day or two they—they could just decide that something should be implemented a certain way or they'll have a certain vision that is different from the business analyst or the lead developer for a few days. They could end up coming back to you with something that's not—not what you need at all” – Site 2 Subject 5 361:369

The utility of Leadership Drum Beating was most visible during meetings. Many of the observed meetings tended to be unfocused, wandering off the agenda and often down a technical rabbit hole. A typical pattern was where a lack of information prevented the team from coming to a consensus on a course of action. Unless someone was willing to guide this meeting, the meeting could rapidly deteriorate into a time wasting argument over an individual’s favourite cause or technology. This is where we observed effective leaders driving the agenda to create a set of actions that could remove the impediment to reaching consensus.

Leadership helps mitigate the negative Social Dynamic enabled by lack of Experience. The Sheltering roles minimize the probability that an inexperienced individual or team will become Exposed. The Alerting role minimizes the probability that an inexperienced individual or team will miss the cues of a possible Perspective Mismatch, and Drum Beating prevents them from going down wasteful “rabbit holes” and reduces the potential for Cut-Off. Finally, Supporting giving an inexperienced individual the knowledge their back is covered, that they will not be persecuted for raising an issue, nor will they be persecuted for simply saying “no”.
Just as Leadership can establish a supportive safe environment for Reaching Out, it can also just as easily create situations where people are unwilling to Communicate their Perspective of a situation for fear of being sanctioned. A common pattern in these situations is that there is an awareness of a Perspective Mismatch but no one will reach out for fear of sanction.

“it's culturally unacceptable to say that you're not going to get something done. That was very clear after I joined this from the outside. I could see that there were people who were not going to get something done but no one will say it—in the meeting. No—so there's a certain culture of I don't know I guess you'd call it fear or something like that” – Site 3 Subject 4 537:543

Leadership can also subvert the Reconciling Perspectives process when the leader realizes there is a Perspective Mismatch and, rather than negotiate a Consensual Perspective, they strive to preserve their own Perspective by cutting their team off from the Ecosystem. This was a common bullying strategy used by both Suppliers and Acquirers. For Suppliers, this pattern of behaviour appeared when a team desired to preserve a favoured technology or was simply disinclined to create Work Products that appeased the Acquirer. A common Acquirer bullying pattern was to impose questionable budgets and schedules for a Job on an individual or team and then simply Cut-Off any Communication on the topic. Such bullying patterns of behaviour lead to Waste and Surprise.
Chapter 5

LITERATURE REVIEW PART 2: SITUATING RECONCILING PERSPECTIVES

This section is the second part of the literature review; it locates *Reconciling Perspectives* within the research landscape by comparing it with extant theory and other empirical studies. Such a comparison may seem, at first, to diminish the contribution of *Reconciling Perspectives* because seemingly little new is revealed when it is presented to study participants or compared to extant theory. Using *Reconciling Perspectives* to explain that individuals must *Communicate* more to *Converge* their *Perspective Mismatches* or otherwise risk generating *Waste* is not revealing anything new. Explaining that software developers need quiet focused time to create *Work Products* to *Get the Job Done* is not revealing anything dramatically new either. Neither does *Reconciling Perspectives* reveal much that is not already well explained by extant theory or that has not been already well studied by others. For example, Social Cognition theory (Bandura, 1986; Fiske & Taylor 1984) can be used to explain how *Perspective Mismatches* arise, how they are reconciled and the consequences of failing to reconcile a *Perspective Mismatch*. Csikszentmihalyi’s theory of flow (1997) and numerous studies (van Solingen, Berghout, & van Latum, 1998; Perlow, 1999; Sykes, 2011; Patrashkova-Volzdoska, et al., 2003) also demonstrate the need of software developers for quiet focused time to *Get the Job Done*. Finally, there is no shortage of research relating personality, experience, and leadership to the ability to *Get the Job Done* (Barrick & Mount, 1991; Homan et al., 2008; Faraj & Sproull, 2000; Burke et al., 2006; Hackman & Wageman, 2005).
This is to be expected because grounded theory is about explaining phenomena from the perspective of those experiencing the phenomena and is not about extending or proving extant theory. Glaser objects to comparisons intending to prove or disprove a substantive theory stating that “Contrasting the generated theory with extant other theories to prove, improve or disprove one or the other neglects or ignores constantly comparing the theories for category and property generation” (Glaser & Holton, 2004, p. 4).

The significance of Reconciling Perspectives is that it is an explanation of how the individuals managing the process of software development understand it from the point of view of the individuals participating in the process. Explaining software development from the point of view of the participants revealed an inherent Communications tension in the process that individuals and teams must manage. What is revealing for researchers is how Reconciling Perspectives weaves or integrates extant theory into an explanation that is relevant to software practitioners. Social Cognition theory explains why Perspectives Mismatches occur and how they may be Converged. However Social Cognition theory does not explain how the individuals and teams Get the Job Done. Csikszentmihalyi’s theory of flow (1997) explains why individuals and teams need quiet focused time to Get the Job Done, but it does not explain how the individuals and teams agree on what the Job is. From an engineering perspective, this enables us as engineering researchers to leverage extant theory and lessons learned in other fields to improve the software development process.
The purpose of this follow-up literature review therefore is to:

1. Compare and contrast *Reconciling Perspectives* with extant theory to position it within existing theoretical frameworks,
2. Use *Reconciling Perspectives* and extant theory to explain software engineering phenomena,
3. Compare results observed in this study with results reported in other empirical studies.

In this section, *Reconciling Perspectives* is first compared to social cognitive theories, specifically Schema theory (Bartlett, 1932; Minsky, 1975; Piaget, 1936; Schank & Abelson, 1977) and Mental Models (Cannon-Bowers, Salas, & Converse, 1993; P.N. Johnson-Laird, 1983; McComb, 2007). Next, the concept of a diverse organizational *Ecosystem* and the emergence of *Perspective Mismatches* is compared with “communities of knowing” and “thought collectives” (Fish, 1980; Fleck, Trenn, Merton, Bradley, & Kuhn, 1981; Wenger, 1999), and then to how these different communities create barriers to *Getting the Job Done* (Boland, 1978; Dougherty, 1992). *Converging* is then compared with Schema processing and Mental Model Convergence (McComb, 2007). This comparison also draws from Weick (1993) to explain why people may not *Reach Out*. Observations on *Bunkering* made in this study are compared with the results from studies on the effects of interruptions and multi-tasking.

### 5.1 Reconciling Perspectives and Social Cognition Theory

*Reconciling Perspectives* explains a process of how an *Acquirer* and a *Supplier* *Get the Job Done* by making sense of a *Job* through social interaction and *Converging* their mismatched *Perspectives* to
avoid generating Waste. *Reconciling Perspectives* is built around the concept of a Perspective that represents an individual’s interpretation or point of view of a Job. Social Cognition is about how people, through social interaction, make sense of other people and themselves, and it draws heavily from theories of cognitive psychology and places them into a social setting (Fiske & Taylor, 1984, p. 21). Schema theory is useful for interpreting and understanding the concepts of Perspectives and the Consensual Perspective, while Mental Model theory provides clarity on the process of converging Perspectives.

Cognitive psychology theorizes that individuals organize memories using cognitive structures that serve as mental templates, and that individuals impose these templates on an information environment to give it form and meaning (Walsh, 1995, p. 281). These cognitive structures are referred to as Schemas (Bartlett, 1932; Piaget, 1936), Scripts (Schank & Abelson, 1977), or Frames (Minsky, 1975), as well as a variety of other terms. Mental Models (Cannon-Bowers, et al., 1993; Johnson-Laird, 1983) are another cognitive structure that is often used to explain coordination in work groups (McComb, 2007; Rouse, Cannonbowers, & Salas, 1992).

### 5.1.1 Perspectives and Schema Theory

A Schema is a cognitive structure which represents knowledge as prototypical or generic templates that are activated when a “stimulus configuration” is encountered in the environment. The stimulus configuration is matched against the schema, and the ordering and relationships among the elements of the schema are imposed on the elements of the stimulus configuration.

---

16 I will use both the term Schema, which is common in the social psychology literature, and also the term Frame, which is commonly used in the computer science and information processing literature.
If information is missing, then defaults may be supplied from the schema. Multiple conflicting schemas may match the same information; which schema is activated depends on its accessibility, which is determined by what other schemas are activated and by personal experience and expertise.

Schemas develop in response to experience, and new information elaborates the schemas making them more complex, abstract, and organized (Fiske & Taylor, 1984). New information may be assimilated into a schema without altering the structure of the schema, or the schema structure may be tuned to accommodate the new information. Schemas may also be restructured to accommodate inconsistencies between the new information and the old schema.

### 5.1.2 Technology Frames of Reference

Technology Frames of Reference (TFR) (Orlikowski & Gash, 1994) were derived from Schema theory as a project-specific tool for understanding technology projects from a social cognitive perspective. Congruence was defined as the alignment of TFRs along key elements that are similar in structure (common categories) and content (common values). When there is incongruence in the TFRs of key organizational stakeholder groups, problems such as misaligned expectations, contradictory actions, resistance, skepticism, and poor appropriation of information technology (IT) may result (Orlikowski & Gash, 1994, p. 180).
5.1.3 Mental Models

A Mental Model is an individual’s mental representation of the world. “A model represents a state of affairs and accordingly its structure is not arbitrary like that of a propositional representation, but plays a direct representational or analogical role. Its structure mirrors the relevant aspects of the corresponding state of affairs in the world” (Johnson-Laird, 1980, p. 98). Mental models “enable individuals to make inferences or predictions, to understand phenomena, to decide what action to take and to control its execution” (Johnson-Laird, 1983, p. 397). Schemas and Mental Models are complimentary models of cognition where mental models are constructed from schema (Johnson-Laird, 1983; Wilson & Rutherford, 1989). In software terms, this is analogous to the relationship between a class and an instantiation of that class (an object).

5.2 The Organization as an Ecosystem of Communities

Reconciling Perspectives occurs within the context of an organizational Ecosystem, where individuals from a number of different communities participate in software projects. Reconciling Perspectives implies that membership in a community influences an individual’s Perspective, but does not conceptualize this relationship. Other researchers and theorists are more explicit about how membership in a community shapes the way an individual interprets the world. Knowledge intensive firms such as software product development companies are composed of multiple communities with specialized technology and knowledge domains (Purser, Pasmore, & Tenkasi, 1992). Such communities have been referred to as “communities of knowing” (Boland & Tenkasi, 1995), “thought collectives” (Fleck, et al., 1981), “interpretive communities” (Fish, 1980), and “communities of practice” (Wenger, 1999). With increasing specialization,
communities with different funds of knowledge and systems of meaning cannot easily share ideas, and may view one another's central issues as esoteric, if not meaningless (Boland & Tenkasi, 1995, p. 351).

In our study, we clearly observed this community of knowing membership. Community membership was not straightforward because individuals did not seem to belong to a single organizational community, such as their team, or a single functional area, such as “programmer”. Rather, people appeared to belong to multiple communities both within the organization and outside of the organization. An individual was not just a developer; they were an Agile developer and shared the views of the wider community of Agile developers. There were also indications of a status hierarchy between occupational communities (Van Maanen & Barley, 1984), the most obvious being a perception that developers who did programming held higher status than developers who did testing.

5.3 Are Software Projects Impeded by Perspective Mismatches?

The primary concern of individuals in this study was Getting the Job Done, which was impeded by Perspective Mismatches that emerged from the different communities in the Ecosystem. While different communities may give rise to different Perspectives, do Perspective Mismatches in general impede Getting the Job Done and create Waste and Surprise? Several studies suggest the answer is yes. Bostrom and Heinen (1977) theorize that problems with an information system implementation often result from the system designers’ technically-biased “frame of reference” filtering social and organizational needs. Boland (1978) argues that a lack of mutual
understanding between system users and analysts originates from the lack of a shared frame between both parties. Here, a frame was deemed to be the set of assumptions, understandings, and expectations that individuals have about a new information system that is used by those individuals to facilitate and articulate their requirements for that new system.

In a heavily cited study, Dougherty (1992) demonstrates how different stake holder “thought worlds” are barriers to communication and coordination and therefore impede new product development. A dramatic example of the differences between the marketing and technology thought worlds is Dougherty’s case of a manufacturing engineer demonstrating the durability of their product’s keyboard by proudly tossing it against the wall. In his “thought world”, the engineer prized reliability and quality. Unfortunately, the product was a commercial failure in part due to the difficulty in using the keyboard, and how often the keyboard could be thrown against a wall was not a factor for commercial success.

Orlikowski and Gash (1994) use their concept of Technology Frames to explain how differences in expectations and actions between technologists and users caused a groupware deployment project to deviate from plan. Davidson (2002) also uses technology frames to study “requirements churn” in the software requirements process and concludes that frequent “frame shifting”, interpreting the world through a newly activated frame (or schema), hinders the participant’s ability to maintain agreement about requirements. Reconciling Perspectives does not explicitly address impediments created by shifting Perspectives; such a phenomenon would impede Negotiating Consensus, or result in Invalidation of the Consensual Agreement if it is realized after the creation of the Consensual Perspective. More recent studies suggest that perspective
gaps between users and developers are a source of requirements uncertainty resulting in schedule slippages and cost overruns (Jiang, Klein, Wu, & Liang, 2009). The challenges experienced during the deployment of an information system at a small construction company are explained in terms of incongruence between stakeholder technology frames (Douglas, Wainwright, & Greenwood, 2010).

That there are Perspective Mismatches between users and developers that can impede Getting the Job Done is not surprising (Glass, 1999; Linberg, 1999), and it is not surprising that these Perspective Mismatches lead to Waste and Surprise. However, during our study we also observed Perspective Mismatches between developers – mismatches between individually held Perspectives that guided design trade-offs, technology implementation choices, work practices, and implementation priority. On a near daily basis it was easy to hear conversations between developers starting with “But I thought we were doing...” or “But I thought you wanted it like this...”

5.4 Converging

During the Converging stage, individuals become aware of a Perspective Mismatch, reach out, and negotiate a Consensual Perspective. There are two sub-stages in Converging: Reaching Out and Negotiating Consensus. In this section, I compare these two sub-stages to Social Cognitive theory.
5.4.1 Reaching Out

Reconciling Perspectives begins when an individual participating in the process of software development receives Communications that conflict with their Perspective of the Job. Viewing this process through a Schema theory lens enables us to elaborate the processing that occurs prior to Reaching Out.

A recurring event during this study was an Acquirer questioning the effort estimate made by a Supplier. During sprint planning meetings, Acquirers questioned estimates made by the Supplier team for a Job. The larger-than-expected estimates made by the Supplier do not match the Acquirer’s Schema for the Job, which informs him that the Job should be inexpensive. If this information cannot be matched to any schema, then the most likely outcome is to discard this information as an outlier or recast it to match a Developers-Think-Everything-Is- Expensive\textsuperscript{17} schema. While it may appear ridiculous to the Supplier that the Acquirer is not able to comprehend an expensive Job, the inability to comprehend observations has been used to explain many product development failures. Such “cognitive inertia” can blind individuals to information that threatens the validity of their schema (Harris, 1994; Krefting & Frost, 1985). Cognitive inertia has been used to explain high profile business failures such as the quality problems that plagued Toyota; financial analyst Alex Taylor suggested that Toyota believed so strongly in its Kaizen culture that it could not comprehend the possibility that its cars may have serious defects “Toyota’s operational skills obscured the need for any change in this [centralized management] structure” (Taylor, 2010).

\textsuperscript{17} Or, as some participants in this study implied, their managers activate and recast this conversation into an “I-will-ignore-everything-anyone-tells-me-that-disagrees-with-my-world-view” schema.
While ignoring or recasting information may help individuals make sense of ambiguous situations, it can also impede an individual from becoming conscious of a Perspective Mismatch and challenging the situation by Reaching Out. Karl Weick’s maxim that “believing is seeing” (Weick, 1995, p. 183) helps explain why the Acquirer may not comprehend an expensive Job because, when receiving ambiguous information, individuals are likely to perceive that which they are predisposed to see (Palich & Bagby, 1995). If the Acquirer’s experience is that Suppliers inflate estimates, then the Acquirer may be predisposed to simply recast the large estimates as being typical of Suppliers inflating their estimates.

In our study, there were many stories which could be explained as examples of cognitive inertia. In one situation, an interview subject claimed another project manager had lied about their project status. We cannot say if they ignored clearly visible warning signs because they could not comprehend their project being in trouble, or if they were truly being capricious. Adolph (2001) is an example of a team that failed because they could not comprehend the early warning signs that their project was heading for failure. Weick’s analysis of the Mann Gulch tragedy explains how thirteen smoke jumpers lost their lives because they could not comprehend their situation (Weick, 1993).

5.4.2 Negotiating Consensus

We referred to the social interaction between individuals who are Converging a Perspective Mismatch as a negotiation because this was how individuals repeatedly describe the process of

18 I do know the individual in question personally and I am doubtful he would deliberately spread false information.
managing software development during interviews. Offer and counter-offer was part of the dialog style we observed when groups gathered around white boards and workstations to resolve issues. Agreement was expressed in the form of the Consensual Perspective. Reconciling Perspectives only describes the Consensual Perspective by its usage to facilitate Communications about the Job and does not describe the structure or content of a Consensual Perspective.

Viewed through a social cognitive lens, a Consensual Perspective is the equivalent of a Shared Schema, or what Sims and Gioa (1986) referred to as a “cognitive consensuality”, that is created by individuals interacting as they do in a negotiation. Negotiation is not a conflict resolution process; rather, it is a social process of building mutual understanding and shared cognition (Van den Bossche, Gijselaers, Segers, & Kirschner, 2006, p. 495). Negotiation is a process designed to achieve agreement “between agents whether the initial starting point is one of conflict or whether it is simply one of ‘absence of agreement’” (Baker, 1995, p. 42). Congruence of technological frame is also similar to a Consensual Perspective where “there is a reasonable amount of implicit agreement among the organization members as to the appropriate meaning of events” (Orlikowski & Gash, 1994, p. 180).

Viewed through a mental model lens, a Consensual Perspective is the equivalent of a Shared Mental Model; Shared Mental Models are “knowledge structures held by members of a team that enable them to form accurate explanations and expectations for the task” (Cannon-Bowers, et al., 1993). The Converging stage is also congruent with what Sara McComb (2007) describes as “mental model convergence,” where team members’ mental models evolve into a shared mental
model through their interactions and observations of each other. McComb used a three stage process to describe mental model convergence (McComb, 2007, p. 103):

<table>
<thead>
<tr>
<th>Stage</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orientation</td>
<td>Orients team members to the team. An individual orients themselves to the team domain by collecting information that has captured their attention.</td>
</tr>
<tr>
<td>Differentiation</td>
<td>Differentiates the disparate perspectives of the team members. The collected information is encoded, organized, and merged with existing knowledge in a manner that provides individual team members with a view of the team that may (or may not) be similar to that of their fellow team members.</td>
</tr>
<tr>
<td>Integration</td>
<td>Integrates these views into the team perspective. The collected information is cognitively organized and merged with existing knowledge in such a way that the information becomes useful for making a response.</td>
</tr>
</tbody>
</table>

Comparing this model to Reconciling Perspectives, Orientation and Differentiation occur during the Reaching Out stage. Something captures an individual’s attention and they reflect on it by differentiating that information to understand how it may be different from information they have previously encountered. Integration is what we have described as negotiation, a transformative process where individuals must reconcile the diversity among their individual perspectives.

To converge mental models, an individual must appreciate that there is a difference between themselves and the individuals they are communicating with. Cognitive inertia impedes this process and, in stressful situations such as being under time pressure, decision makers tend to
rely on past issue perceptions (Dutton, 1993). In our study, we collected data that corroborates the effect pressure has on Reaching Out. We were told several stories of troubled projects where collaboration broke down, and individuals and teams Cut-Off communications in a desperate effort to Get the Job Done. Individuals and teams are less likely to reach out when under stress, which is precisely the time when they most need to reach out.

A significant difference between Reconciling Perspectives and Mental Model Convergence is the purpose of Reconciling Perspectives in converging the Mismatched Perspectives that individuals in an Acquirer-Supplier relationship have with respect to a Job. The concern of Mental Model Convergence is team formation and how individuals integrate themselves into a team. In shared mental model convergence, the process may rapidly iterate as individuals observe and interact with their colleagues, incrementally converging their mental models. In Reconciling Perspectives, this is more analogous to Invalidation, where new information invalidates the Consensual Perspective.

5.5 Validating and Bunkering

A Consensual Perspective is a necessary condition for Getting the Job Done, but it is not sufficient by itself to resolve the main concern of the study participants because Getting the Job Done requires the delivery of Work Products. During the Bunkering stage, study participants seek quiet focused time to create Work Products by filtering distractions. It is only after the Work Products are accepted that there is objective evidence the Perspective Mismatch has been reconciled. We conceptualized this stage from our observations of the quiet periods that frequently followed
protracted negotiations, by observing individuals brush others off by telling them they needed time to get some work done, and by interviewing subjects who realized that always being available and agreeable was not a good way to get a job done. Bunkering is the stage “when I can actually get some work done” as one individual quipped during an informal conversation regarding “quiet” hours at their site. Interruptions and distractions were frequently cited as highly frustrating impediments to Getting the Job Done – “I could get this done if they would just stop bugging me!” – and several team leaders saw it as their job to protect the team from time-draining interruptions.

Saturation results in Waste and occurs when individuals do not filter Communications and indiscriminately engage in all interrupting Communications. Every meeting invite, every conversation, every e-mail, every IM results in an interruption requiring rework, or an issue that must be explored, or even a diversion to working on an unrelated Job. Individuals cannot find or create the quiet focused time they need to Bunker and Get the Job Done. In a case study of interruptions at companies developing embedded systems, Demarco and Lister (1987) asserted that 15% of a software developer’s day is lost to wholly unproductive phone calls and e-mails (van Solingen, Berghout, & van Latum, 1998). Each interruption consumes, on average, about 15 to 20 minutes of the developer’s time, and approximately 1.5 hours per day of a developer’s time is taken up dealing with them. Perlow’s (1999) study of engineers at work suggests that the frequent coworker interruptions experienced by software engineers lead to “a time famine” wherein the engineers are plagued by the sense of having more job responsibilities than time in which to do them.
E-mails, telephone calls, a colleague dropping by to talk, and the occasional Nerf Dart war, were not the only source of interruptions that were complained about. Unplanned work was another source of interruptions, with **Job** owners trying to circumvent project management processes and directly interrupt individuals to get a job done. Multi-tasking was a frequent source of complaints. All teams in this study were project teams and were tasked with getting a specific **Job** done; however, these teams were frequently interrupted by support requests and ad hoc tasks such as estimating a bid. Many individuals were also split across multiple projects, or had consultative and support responsibilities for prior work.

Multi-tasking is generally viewed as inefficient because individuals must mentally task switch which, in itself, increases an individual’s cognitive load because of the mental awareness required to manage switching between multiple different tasks (Sykes, 2011). Experiments in task switching discovered participants lost time when they had to switch from one task to another (Rubinstein, Meyer, & Evans, 2001). As the tasks became more complicated, the participants took significantly longer to switch between tasks, and lost even more time. The cost of task switching can be substantial, with some citing Meyer’s (Meyer & Kieras, 1997a, 1997b) work as evidence this cost may be as high as 40% (APA, 2006).

Individuals and teams adopted numerous strategies for **Sheltering** themselves from interruptions during **Bunkering**. Individuals often made themselves unavailable for interruption by working “off-hours” or even “off-site”. Some teams used a “sacrifice-one-person” pattern (Coplien & Harrison, 2004), and put a select portion of the team on the “hot seat” to deal with interruptions while letting the rest of the team make progress towards **Getting the Job Done**. Team leaders
frequently worked as “gate-keepers” to deflect and shelter their team from interruptions. When sheltered from interruptions, individuals spoke of “being in the zone” and some even cited popular references to Csikszentmihalyi’s (1997) concept of “flow” which is frequently characterized as a high state of enjoyable focus or concentration on achieving a well-defined goal. Interview participants expressed frustration when the flow was disrupted, especially when they believed the interrupting activities were of low value.

5.6 Tension in the Process

Much of our data suggests insufficient Communications is a source of Waste and Surprise (Failure to Reach Out, Isolation) because if people do not communicate there is no opportunity to detect a Perspective Mismatch. Social cognition is a social activity because without social interaction there is no opportunity to create a social schema. One project manager mentioned that her three rules of software development were “Communicate, Communicate, Communicate”. Another individual cited lack of Communications as a cause for a significant project failure “…but there just weren’t enough conversations taking place”. From this, we could infer an inverse relationship between Communications and Waste and the associated Surprise. Simply put, the more conversations that occur, then the less Waste will result.

However, in my theory, the relationship between Communications and Waste is somewhat more complex because Bunkering conceptualizes the software developers’ need for quiet time to focus on Getting the Job Done. Some take this need to the extreme and completely cut themselves off from the organizational Ecosystem by ignoring and deflecting all interruptions. Cutting Off all
Communications is not a productive strategy in a software development organization because it leads to Isolation and inhibits Reconciling Perspectives. The root cause of many problems observed at subject sites could be traced back to Perspective Mismatches that were not reconciled because barriers to Communications were created by blocking all interruptions. Similar results have been reported in other studies, where blocking interruptions created barriers to shared perspectives (Boland & Tenkasi, 1995; Dougherty, 1992; Purser, et al., 1992). Yet other studies demonstrate that indiscriminately responding to interruptions impedes the creation of Work Products (Speier, Valacich, & Vessey, 1997; Sykes, 2011; van Solingen, et al., 1998). This tension can be expressed as a “U” shaped relationship between Communications and Waste.

![Figure 27: Relationship Between Communications and Waste](image)

A study of sixty cross functional teams discovered a similar curve-linear relationship between communications frequency and team performance (Patrashkova-Volzdoska, et al., 2003).
The study authors provide two explanations for why higher levels of communication will eventually lead to lower performance. The first is simply information overload; individuals were spending their time sifting through volumes of information to find what was relevant and “thereby using their limited time to process unnecessary information” (Patrashkova & McComb, 2004, p. 101). The second explanation is that high volumes of communication may indicate confusion or conflict in the team about the project goals – a Perspective Mismatch. Their assessment was that teams engaged in very high levels of face-to-face communications were in jeopardy. From this, we could infer that excessive Communications in Reconciling Perspectives may be an indication of Convergence Stalling, where the team is unable to achieve a Consensual Perspective.
During this study, we saw evidence of over-communications indicating goal conflict within teams that were co-located with clients. At Site 1, the team was co-located on their client’s site, and client representatives were frequently trying to bypass the “gate-keeper” and directly approach developers to add features to the system that were not in the agreed specification. At Site 2, teams were frequently co-located on the customer’s site and were subject to significant pressure to add features or make modifications that had not been agreed to. This is an example of what Patrashkova and McComb referred to as goal confusion or conflict.

Another result from Patrashkova and McComb’s study is the significantly higher level of performance achieved with face-to-face conversations and the low levels of performance related to email as a communications medium. This highlights a significant concern regarding performance in distributed teams because e-mail is the only medium for which usage increased with distance (Patrashkova-Volzdoska, et al., 2003, p. 267). We saw strong evidence of this at Site 3, which was a multi-national software product development company with development offices around the world. One subject in our study, when describing the increasing lag in decision time the further apart the decision makers were, cited what he called the Rule of Three. The Rule of Three suggests that as communication distance increases, the time required to reach a decision goes up by a multiple of three, from three minutes talking to a neighbouring colleague, to three weeks when talking to an overseas colleague.

_Bunkering_ highlights the individual and team need to establish good _Communications_ boundaries to _Get the Job Done_, but Social Cognition theory does not provide a good explanation for this boundary setting process. Much of the literature using social cognitive theory to analyze product
develop teams highlights the social cognitive consequence of barriers to communication such as distance and formal processes (Dougherty, 1992; Jiang, et al., 2009; Purser, et al., 1992). Getting the Job Done requires individuals and teams to have the ability to find the Communications equivalent of the Aristotelian Golden Mean. Individuals and teams need to socially interact to Converge their Perspectives, but they also need quiet uninterrupted focused time Bunkering to Get the Job Done.

5.7 The Social Dynamics of Managing Communications

Social Dynamics conceptualizes how individuals and teams interact socially to set their boundaries by managing their Communications. Social Dynamics is an aggregation of the four individual concepts of Personal Strength, Experience, Leadership, and Openness that determine how individuals and teams managed their Communications. There is no shortage of studies and theories relating social dynamics to team performance. A Google Scholar search on “team social dynamics performance” yields some 538,000 hits as of June 30th, 2012. We conceptualize Social Dynamics as the social interactions that enable an individual or team to manage their Communications such that they can Get the Job Done. We define performance as minimizing Waste and avoiding Surprise.

In this sub-section, I compare these properties of Social Dynamics with extant theory and other studies to highlight areas of overlap with our definition of these concepts and to further our understanding of how these properties moderate team performance. I then compare the
Reconciling Perspectives concept of Leadership with extant Leadership theory to further our understanding of how Leadership influences this process.

5.7.1 Personal Strength

Personal Strength was conceptualized to capture the willingness of an individual to Reach Out when they suspected a Perspective Mismatch or to remain focused on Getting the Job Done during Bunkering. Waste is created when people lack the Personal Strength to reach out (Failure to Reach Out) or when they cannot say “no” (Saturation) to interruptions. Several interview participants characterized software developers as introverted when explaining why they thought people did not reach out when they suspected a Perspective Mismatch, or say “no” when they were interrupted during Bunkering. Studies of software engineering personality types are, at best, mixed when it comes to supporting the assertion that “software types are introverts”. Lyons (1985) surveyed 1229 people from more than 100 companies and found that 67% of his subjects were introverted. Smith (1989) studied 37 systems analysts and reported that 57% were introverts. Later studies discovered a fairly even split between introversion (53%) and extroversion (47%) when a Myers Briggs (Myers, 1998) personality test was applied to an undergraduate software engineering class (Layman, Cornwell, & Williams, 2006).  

While team performance correlates positively with extraversion (Barrick & Mount, 1991; Kichuk & Wiesner, 1997; Neuman & Wright, 1999), assertiveness may play a bigger role in the result than other factors associated with extraversion (Barry & Stewart, 1997). Assertiveness is the capacity to effectively communicate in interpersonal encounters by sharing ideas clearly and  

19 Perhaps an indication that software development is becoming a more fashionable profession
directly (Wolpe & Lazarus, 1966), making available information that the team needs to effectively perform their tasks (Smith-Jentsch, Salas, & Baker, 1996). Pearsall and Ellis’s (2006) study of 268 students, in an introductory management course simulating battlefield decision-making, positively correlated critical individual assertiveness to team performance. Assertiveness captures much of our conceptualization for *Personal Strength*.

### 5.7.2 Experience

*Experience* was conceptualized as the knowledge gained over time throughout an individual’s career, and it includes knowledge gained through working in a specific *Ecosystem* and the familiarity an individual develops with other individuals within and outside of their immediate team. In other studies, individual experience – measured in terms of years of experience – positively correlates with productivity (Cheney, 1984; Shafer, Nembhard, & Uzumeri, 2001). The Constructive Cost Estimating Model (COCOMO) (Boehm, 1984) attempts to quantify experience as a software project cost driver and shows an experienced analyst is nearly 60% more productive than an inexperienced analyst. There are three types of experience that are relevant to this concept (Reagans, Argote, & Brooks, 2005, p. 870):
<table>
<thead>
<tr>
<th>Experience Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual Experience</td>
<td>The cumulative production history of any one individual that provides an opportunity to become more proficient at performing their tasks.</td>
</tr>
<tr>
<td>Experience Working Together</td>
<td>The cumulative production history of the individuals in a team. With more experience working together, individuals become more willing to share knowledge.</td>
</tr>
<tr>
<td>Organizational Experience</td>
<td>The cumulative production history of the organization. As the organization gains more experience, each individual has more opportunities to benefit from the knowledge accumulated by others; the knowledge of how to get things done.</td>
</tr>
</tbody>
</table>

Table 16: Types of Experience (Reagans, et al., 2005, p. 870)

The role of *Experience* with respect to managing *Communications* is that more experienced individuals know when something is wrong, and when it is appropriate to reach out. While the development of individual skills is important in a knowledge intensive industry, knowing how to get things done is an important aspect of *Experience*. *Reaching Out* depends on individuals knowing when things “don’t sound right”. One participant said she could “*smell*” when things weren’t right. Interview participants spoke of beginning to realize that it was “*OK to ask questions*” and knew who to ask questions of. People also spoke of learning the “*real way things were done*” in terms of what operating procedures could be ignored, and which ones had to be followed.

Organizational experience is a significant component of our conceptualization of *Experience*. For many of our study participants, a significant portion of organizational knowledge seemed to represent “*battle scars*” — harsh lessons learned from near failed projects. Experience of how to
work together contributes to team performance because individuals have a better sense of who
knows what, and therefore who to go to when they need help, and how to coordinate and use
their expertise (Faraj & Sproull, 2000).

5.7.3 Leadership

The concept of Leadership captures the influence individuals have on other individuals and on
team performance. This influence was dramatic because all the stories of challenged projects
and successful projects revolved around leaders who either led their team down a rat hole or
stood up and protected them. The disproportionate influence an individual can have on team
performance attracts significant research interest and a Google Scholar search on “Leadership”
generates no less than in 1,490,000 hits (June 30th 2012). The leader’s ability to influence the
outcome of the process is a common pattern in many of the stories that were related to us
regarding the success or failure of projects.

Our conceptualization of Leadership most closely matches Bass’s view of leadership as the
“focus of group processes” (Bass & Bass, 2008, p. 16). Building on Bass’s view, Northouse
defined leadership as a process “whereby an individual influences a group of individuals to
achieve a common goal” (Northouse, 2012, p. 5). In Reconciling Perspectives, Leadership
conceptualizes the roles played by individuals who influence others but are not necessarily in a
defined position of authority.20 The Reconciling Perspectives concept of Leadership captures the
four roles that leaders played in this study:

20 The term leader may not be the best term; Matthew J. Pearsall* and Aleksander P. J. Ellis used the term “critical
team member” in their study Pearsall, M. J., & Ellis, A. P. J. (2006). The Effects of Critical Team Member
<table>
<thead>
<tr>
<th>Role</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting</td>
<td>Creating an environment that encourages <em>Reaching Out</em>.</td>
</tr>
<tr>
<td>Sheltering</td>
<td>Keeping individuals connected to the <em>Ecosystem</em> while protecting them from becoming <em>Exposed</em> (sometimes called buffering, or shielding).</td>
</tr>
<tr>
<td>Alerting</td>
<td>Preventing individuals from becoming <em>Cut-Off</em> by staying connected to the <em>Ecosystem</em> and bringing information that may indicate <em>Perspective Mismatches</em> to the attention of individuals and teams.</td>
</tr>
<tr>
<td>Drum Beating</td>
<td>Avoiding <em>Convergence Stalling</em> by moving the process forward and keeping negotiations focused.</td>
</tr>
</tbody>
</table>

Table 17: Reconciling Perspectives Leadership Roles

### 5.7.3.1 Sheltering

In the *Sheltering* role, a leader protects individuals from interruptions that disrupt their flow and potentially impede them from *Getting the Job Done*. In the *Sheltering* role, leaders place themselves as gate keepers between the *Ecosystem* and the sheltered individual or team. They make it known that “outsiders” are to approach only the leader and leave the sheltered individuals alone. Coplien and Harrison (2004) called this role a “Firewall”, a managerial role for shielding team members from interactions with others from outside of the team. The intention of *Sheltering* was to filter *Communications* to reduce the number of interruptions so that the team could focus on *Getting the Job Done*. While it was common to see leaders in positional authority roles shelter a team, not all leaders playing the *Sheltering* role were in positions of authority over the individuals they were *Sheltering*. There were several instances of individuals who possessed significant domain or technical knowledge drawing away interruptions from other team members.
5.7.3.2 Alerting

The *Alerting* role is comparable to what much of the literature refers to as Boundary Spanning (Burke et al., 2006; Levina & Vaast, 2005) or Gate Keeping (Elkins & Keller, 2003; Katz & Tushman, 1981). In these roles, leaders acquired information from the *Ecosystem* and shared information with their team, and also shared information about their team with the organizational *Ecosystem*. In our theory, the *Alerting* role complements the *Sheltering* role because, without the *Alerting* role, the *Sheltering* role can deteriorate into a leader-enforced *Cut-Off*. Many team leaders saw their primary roles as being the gate through which information flowed between the team and the organizational *Ecosystem*.

Not all boundary spanning leaders sheltered teams. Some were like rangers, roaming about the organizational *Ecosystem* exchanging information and searching for *Perspective Mismatches* in an effort to keep alignment of the project vision. For example, an analyst at Site 2 moved from team to team sharing requirements information she acquired from other teams. Often, during her conversations with different teams, she encountered conflicting information, at which point she alerted the teams to a possible *Perspective Mismatch*. A different example was a technical thought leader who operated like an information clearing house. Members from other teams frequently went to him to get advice on technical issues, and it was not uncommon to see a line up at his cubicle as people waited to consult with him. Often people would ask him questions about technical approaches they were taking in their design that, in his mind, conflicted with either his vision of the system or with design approaches others were taking. Usually, he would immediately reach out to the parties involved to reconcile the *Perspective Mismatch*. 
Alerting is a component of the boundary spanning role, where leaders compared the information they were acquiring from the organizational Ecosystem with what they were hearing within their team, and discovered evidence of a Perspective Mismatch. The leader reached out to alert individuals to the potential Perspective Mismatch and started the Reconciling Perspectives process. In some of the stories of challenged projects, one common theme was of a leader who abdicated their role as a boundary spanner by either cutting the team off completely from the Ecosystem or leaving the team completely Exposed.

Similar to our findings, other studies have clearly demonstrated the importance of the boundary spanning or gatekeeper role. A meta-analysis of leadership behaviours found that boundary spanning and empowerment explained large variances in team effectiveness (Burke, et al., 2006). Edmondson (2003) found that boundary spanning was associated with successful technology deployment.

5.7.3.3 Drum Beating

We observed Communications breaking down during Negotiating Consensus, where either the Communications simply slowed until it Cut-Off altogether or it went “around in circles” with individuals incessantly arguing a variety of competing scenarios and debating minutia. This latter situation was an example of what Patrashkova and McComb (2003) were referring to as goal confusion or conflict. These situations frequently arose during Negotiating Consensus when insufficient information was available to arrive at a Consensual Perspective. When Drum Beating, a leader works to re-establish the negotiation rhythm and get all parties Converging again towards a Consensual Perspective.
Unlike our findings, Malone and Crowston defined Coordination as managing dependencies between activities (Malone & Crowston, 1994). We interpreted *Drum Beating* as a coordination activity because *Drum Beating* is about managing the dependencies that have blocked *Converging* towards a *Consensual Perspective*. A *Perspective Mismatch* emerges between individuals performing inter-dependent activities who need to coordinate to remove the impediment. When the process is blocked, leaders *Drum Beat* by either re-engaging individuals in the process or by facilitating the creation of a plan to obtain the needed information to unblock the process.

### 5.7.3.4 Supporting

Many individuals participating in this study described how their managers created a supportive environment for *Reaching Out*. As one interview participant said “*I know my back is covered*”. *Supporting* was not only encouraging and covering individuals when they reached out, it was also encouraging them to say “no” to *Communications* they found disruptive to *Getting the Job Done*. We interpreted this role as *Supporting* because leaders were helping individuals perform tasks (Hackman & Wageman, 2005). In their *Supporting* or coaching roles, leaders were helping individuals strengthen their personal contribution by motivating individuals to reach out when necessary, and teaching them when to say “no”.
5.8 The Influence of Social Processes on Software Development

Reconciling Perspectives explains a process for how people manage the process of software development, a process that is moderated by Social Dynamics. Social Dynamics explains how the social factors of Personal Strength, Experience, Openness and Leadership influence Reconciling Perspectives and the creation of Accepted Work Products or the creation of Waste. Other software engineering researchers have also studied social processes and the influence social process have on managing the process of software development and a comparison of their results can help elaborate Reconciling Perspectives’ concept of Social Dynamics.

Ferreira and colleagues (Ferreira, Sharp, & Robinson, 2012) study of how two different communities of developers; user interaction designers and agile software developers integrate their work describes a process similar to Reconciling Perspectives. Agile software developers and user experience designers are examples of two different “communities of knowing” (Boland & Tenkasi, 1995) who have specialized funds of knowledge and practice; cannot easily share ideas and yet, must work together to Get The Job Done. The four themes for achieving integration between these two different communities are described in table 18.
<table>
<thead>
<tr>
<th>Theme</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual awareness</td>
<td>The level of awareness the developers and designers have of each other’s work affects their ability to make informed judgments around coordinating their work.</td>
</tr>
<tr>
<td>Expectations about acceptable behaviour</td>
<td>Integration is shaped by expectations about how the other group should support them in their work.</td>
</tr>
<tr>
<td>Negotiating progress</td>
<td>Progress was negotiated among the developer and designers, who discussed what needed to be done</td>
</tr>
<tr>
<td>Engaging with each other</td>
<td>Integration relied on the recurring efforts of the Agile developers and UX designers engaging each other for their input.</td>
</tr>
</tbody>
</table>

Table 18: Four Themes for the Integration of Communities (Ferreira, et al, 2012)

Coming from different communities of knowing, UX designers and agile software developers may have different Perspectives of how the other community performs their work. These four themes align with Reconciling Perspectives Converging stage; to Reach Out and Negotiate Consensus, UX designers and agile software developers need to be mutually aware of each other’s work, have expectations of acceptable behaviour, and be engaging with each other. Impediments to these themes such as impediments to Reaching Out and Negotiating Consensus, for example, can lead to the creation of Waste. It was observed how Waste resulted when Leadership at one of their study sites discouraged individuals from the UX developer community and agile software developer community from becoming aware of the other’s work.
These themes for integration do not directly address the *Validating* stage of *Reconciling Perspectives* however it may be implied because it was observed “closeness” between designers and agile developers enabled these practices. In terms of *Reconciling Perspectives* this could also imply a fast *Cycle Time*. This supports the assertion shorter *Cycle Times* reduce *Waste* and the frequent interactions between the developers and designer ensured that expectations about how the developers and designer should support each other in their work was negotiated on a day-to-day, moment-by-moment basis.

Brown and colleagues (Brown, Lingaard, & Biddle, 2008) also studied collaboration between the same two communities of knowing: UX designers and agile software developers. Unlike Ferreira however, Brown and colleagues drew from cultural-historical activity theory (Vygotsky, 1986) as a framework for their study to understand how artefacts facilitated collaboration between different communities of knowing. One aspect of cultural-historical activity is the mediating role of cultural artefacts and the meaning individuals associated with those artefacts. Brown and colleagues identified twelve different types of artefacts that mediated collaborations and facilitated group cognition and included, design ideas, stories, exemplars, models, plans, and software under development (p. 91).

*Reconciling Perspectives* does not explicitly call out artefacts, treating them as another form of *Communications*. During our observations of participants, they made regular use of what Brown referred to as artefacts: sketching models, placing and moving sticky notes on a Scrum board, walking through code, manipulating wire frames all in an effort to facilitate their
Communications. Brown’s study highlights the role of artefacts in communications and in arriving at a Consensual Perspective.

Whitworth’s and Biddle’s study of XP teams investigated the social processes that contribute to the team’s success (Whitworth & Biddle, 2007). Their findings suggest agile practices were seen by the study participants to increase the ability for team members to work together with others in an environment of constant feedback and progress. Whitworth’s study described the experience of a positive social dynamic that was facilitated by the adoption of agile software development practices and uses social identity theory (Tajfel & Turner, 1986) to explain how XP practices mediated the experience of individuals and the team as a whole. Social identity theory posits individuals have social identities corresponding to perceived membership in a social group (Whitworth & Biddle, 2007, p. 8) and agile practices were seen to heighten the salience of a project team identity over the individual identity.

Whitworth’s study highlights the influence of agile (specifically XP) practices on individual experience, which contrasts with Reconciling Perspectives finding of the influence of Personal Strength and Leadership on the process and therefore the individual experience. However, while all teams self described themselves as “agile” there were substantial differences in their experience of agile which was very much driven by those who assumed leadership roles. This contrast maybe explained by variances in implementation of agile practices and therefore personal identity being more salient that project team identity. I collected many stories of teams where strong willed individuals used agile practices to either move a project forward or support their own agenda. People truly trump process (Cockburn & Highsmith, 2001).
Hoda, Noble, and Marshall’s (2010) investigation of self-organizing teams highlighted the influence of individuals in the roles they play facilitating the organizing of self-organizing teams. Hoda and her colleagues identified six roles: mentor, coordinator, translator, champion, promoter and terminator. While the focus of their study was on team self-organization in companies adopting Agile software development methods, when their findings are compared with Reconciling Perspectives their findings further highlight the influence of Leadership on Social Dynamics.

Several of the roles are comparable with the Leadership role of Supporting to boost Personal Strength: Mentor, Champion, and Promoter may all be seen as roles that give team members confidence in the process, and confidence to Reach Out and attempt to engage others when they believe they have a Perspective Mismatch. The role of the Coordinator is similar to the Sheltering and Alerting roles where a leader protects a team from interference yet keeps the team sufficiently connected to the organizational Ecosystem such that they are not closed or Cut-off from Communications. In Hoda and colleagues model, both the Translator and the Promoter could also be seen as creating a rhythm, and being the Drum Beater both within the team and to the outside organization.
Chapter 6

RECONCILING PERSPECTIVES AND SOCIO-TECHNICAL APPROACHES TO SOFTWARE ENGINEERING

This is an engineering dissertation, and a good engineering dissertation should apply what was learned to the practical everyday issues experienced by practitioners. I undertook this study to understand how people manage the process of software development with the intention of informing policy for the design of software development methodologies. What I observed at all three sites during this study is what I have also experienced in my personal career, which is that people involved in the process of software development do not follow a “rational methodology”. Rather, they create a “mashup” of practices borrowed from various different methods to create a “method in use” (Edberg, Ivanova, & Kuechler, 2012). All sites participating in this study employed a mashup of various agile and waterfall based practices.

In my personal view, a “good” software development methodology is one that individuals managing the process of software development will embrace more tightly, rather than abandon when the inevitable project pressure comes on. This means that individuals must believe the software methodology addresses their issues and will help them resolve their problems. In this section, I show how Reconciling Perspectives can guide the design of software development methodologies. Sub-section 6.1 uses Reconciling Perspectives to argue that software development is a social process, and explains why people do not use software methodologies. While social processes are a significant aspect of software development, we cannot diminish the technical
practices and sub-section 6.2 introduces socio-technical processes (Trist, 1981) where technical and social practices synergistically support each other. I argue that the rise of Agile methods is due, in part, to Agile methods viewing software development as a socio-technical process. I compare the Agile Manifesto to *Reconciling Perspectives*, and compare *Reconciling Perspectives* to Scrum, an Agile methodology that highlights social relationships. In sub-section 6.3, I provide guidance for how *Reconciling Perspectives* can inform the design of scalable socio-technical software methodologies.

Finally, while a software methodology can provide direction, guidance, a common vocabulary, and a common vision for how things are done, it cannot legislate personality. *Reconciling Perspectives* clearly demonstrates that the process is dependent on individual abilities to reach out, to negotiate, and to appropriately manage communications. In sub-section 6.4, we draw from the experience gained in other industries to recommend a training and management agenda.

### 6.1 Software Development as Social Process

The Grounded Theory Method generates theories that explain the main concerns of the participants in the study instead of the interests of the researcher. *Reconciling Perspectives* explains software development as a social interaction between individuals and groups, and does not describe software development as a reductive transformative process. The issues that are the main concern of the individuals participating in this study are social issues.
Reconciling Perspectives supports the argument that software development is a social process (Cockburn & Highsmith, 2001; Fitzgerald, 1997; Lister & DeMarco, 1987; Sawyer & Guinan, 1998). Reconciling Perspectives informs us that software development is about on-going negotiation and the management of Communications. Over and over again, the word “negotiation” was used by interview participants when describing their software processes – not just between business Acquirers and the Supplier software team, but also between different groups and individuals in the engineering organization. They reached out; they negotiated; they sheltered themselves from interruptions; they managed the tension between their need to Get the Job Done and their need to remain open to Communications.

Ginsberg (1939) defined a social process as “…the various modes of interaction between individuals or groups including cooperation and conflict, social differentiation and integration, development, arrest and decay” (p. 437). Social processes can generally be categorized as either “associative” or “dissociative” (Simmel, 1964; von Wiese & Becker, 1932). Associative behaviours increase the likelihood of mutual understanding and cooperation, whereas dissociative behaviours contribute to misunderstanding and competition (Gudykunst, 2000, p. 143). As methodology designers, our goal becomes to develop policies that accentuate associative behaviors and mitigate dissociative behaviours.

The majority of software methodologies do not address social roles in software development beyond simple platitudes such as “get the support of senior management” or “involve the end user”. This does not offer much guidance for accentuating associative behaviours and mitigating dissociative behaviours. The inherent problem of the transformational bias of most software
methodologies is that while they do not, in general, explicitly impede the opportunity to engage socially, they do not provide a forum for these activities. This masks the social nature of software development such that software developers do not understand, let alone fully appreciate, the social nature of systems development (Hirschheim & Newman, 1991, p. 30).

Curtis, Krasner, and Iscoe (1988) also suggested that software development is a process of learning, technical communications, requirements negotiations, and customer interaction. Software development was characterized as a learning process that is not well served by methods that only view software development processes as a series of development tasks. In response to this model mismatch, they proposed a layered behavioural model of software development.

6.1.1 Software Methodologies Are Not Used Because They Are Not Useful

I used various technology adoption models (TRA, TAM) to argue that people do not adopt and use software methodologies because they do not find them useful. Reconciling Perspectives explains the process of software development as a social process where people interactively engage one another to reduce Waste and Surprise resulting from Perspective Mismatches. In contrast, many software methodologies try to reduce Waste and Surprise using rigorous and unambiguous specification of work products, inputs, processes, outputs, and roles. For example, software requirements ambiguity is a significant source of Waste (Curtis, et al., 1988; Jiang, et al., 2009; Lyytinen, 1988) and the IEEE SRS guide specifies that a set of good requirements has the following characteristics (IEEE, 1998):

1. Correct;
2. Unambiguous;
3. Complete;
4. Consistent;
5. Ranked for importance and/or stability;
6. Verifiable;
7. Modifiable;
8. Traceable.

The guide then proceeds to define and elaborate the intent of each of these characteristics. For example, an unambiguous requirement is defined as “if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term” (IEEE, 1998, p. 4). The intention is to avoid, or at least reduce, the Waste resulting from Perspective Mismatches by removing ambiguity as a potential source of a Perspective Mismatch. If all inputs, processes, outputs, and roles of the software development process and Job have a clear single interpretation that is understood by everyone, then everyone will share the same Perspective of the Job and there will be no Perspective Mismatches as Job Work Products are handed off between software process stages.

To satisfy these recommendations for requirements rigor, many software methodologies make intense use of a variety of requirements models such as use cases, activity diagrams, state transition diagrams, etc. Many methodologies include detailed work product templates with numerous fields that must be filled in for each modeling element. Roles and responsibilities are defined, as are the tasks each individual playing a role must perform – all of which is often enforced by a tool. However, as more practices and the more rigor are added to a methodology
to reduce *Perspective Mismatches*, the greater the “weight” of the methodology. Software methodology weight is a subjective reference to a software methodology’s usage cost, and it is broken down into two components (Cockburn, 2003, p. 41):

1. The number of elements in the methodology (its “size”);
2. The rigor with which the elements must be carried out (its “density”).

One of the most common complaints about software methodologies is that they are too heavy and the cost of their practice exceeds what individuals consider reasonable for the benefit gained. It is not hard to see how a software method can become “heavy” from the practices put in place to prevent *Perspective Mismatches* which, in many situations, could perhaps be inexpensively solved by simple face-to-face conversations at a whiteboard.

When the process described by rational software methodologies and the process described by *Reconciling Perspectives* are compared, the gaps between how rational methodologies specify that software should be created and how people actually create software is revealed. People do not find rational software methodologies useful because rational software methodologies are:
Most rational methodologies are limited to using Input-Process-Output models for specifying linear technical processes and do not encourage, or provide support for, the way people actually create software. This study discovered that people use a social process of Reconciling Perspectives to create software, something which is not supported much beyond platitudes by rational methodologies (Newman & Robey, 1992).

Rational methodologies may even impede social processes by creating barriers to these social processes with rigidly defined job roles, work products, process, and an emphasis on rational formality (Dougherty, 1992; Purser, et al., 1992).

The practices that rational software methodologies specify to address the main concerns of people are heavy because rational software methods attempt to use technical practices to remove or prevent the impediments created by Perspective Mismatches. (Cockburn, 2000).

Table 19: Reason Why Rational Software Methodologies are not Useful

In the late 1990s, so called “lightweight” software development methods such as Scrum (Schwaber & Beedle, 2002) and Extreme Programming (XP) (Beck, 2000) began to emerge. These methodologies were a radical departure from conventional rational methodologies because they reduced methodological weight by giving prominence to social practices and collaboration. There has been considerable research into the use of these lightweight methods; early studies (Laanti, Salo, & Abrahamsson, 2011; Williams, Kessler, Cunningham, & Jeffries, 2000) reported that lightweight methods could improve software team performance, and also implied that these improvements resulted from improved team social practices (Abrahamsson, 2003).
6.1.2 The Agile Manifesto and Social Processes

Taking a social perspective on software methodologies is to view software development methodologies as descriptions of the “...various modes of interaction between individuals and groups...” (Ginsberg, 1939, p. 437) rather than as a series of transformational processes that individuals animate. A turning point in software methodology design occurred in 2001 when a group of software methodologists advocating lightweight software development methodologies gathered to express their desire for a more socially oriented focus on software development, and signed the Agile Manifesto (AgileAlliance, 2001a). The Manifesto captured four articles for guiding the creation of software methodologies by valuing:

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

The intent of the Manifesto was not to discard technical practices but rather to find a better balance between social and technical practices in software development (Fowler, 2001). An often overlooked line in the Manifesto explicates this desire to seek a different socio-technical balance, stating “while there is value in the items on the right, we value the items on the left more”.

Specifying social practices for supporting social processes, and giving those practices prominence in software methodologies, was seen as a more efficient and effective way to manage the process of software development than rationalizing social processes with technical
practices. Agile methods place a higher value on individuals and interactions over tools and processes because methodology weight, in the form of technical practices (processes and tools), can be traded off against individuals and interaction. “With better communication, we can shed bureaucracy. Conversely, to the extent that the group cannot get frequent and rich personal communication, it must add compensation elements to the methodology” (Cockburn, 2003, p. 43).

6.1.3 Scrum as Social Process

While there are numerous methodologies under the Agile umbrella, Scrum (Schwaber & Beedle, 2002) is the methodology that all teams participating in this study claimed to use; it is also arguably the most widely used Agile method (Begel & Nagappan, 2007). Scrum is a methodology for organizing people to deliver a product or service. Unlike other software methodologies, Scrum only describes how people interact to deliver a product or service and does not describe the technical practices for creating software. Scrum is an iterative process that delivers a product or service regularly using a fixed time box called a Sprint.

![Scrum Workflow](screenshot)

Figure 29: The Scrum Workflow (Sutherland & Schwaber, 2007, p. 15)
The workflow in Scrum is elementary; work is pulled from a “To Do” list called a Product Backlog, and is transformed into Deliverables that represent valuable products or services. Scrum defines three roles:

**Product Owner**
The individual who owns the Product Backlog. The Product Owner prioritizes the items in the Product Backlog and supports the Delivery Team as a Subject Matter Expert.

**Delivery Team**
A cross-functional self-organizing team of 7 +/- 2 individuals who collectively have all the skills necessary to transform Product Backlog items into products or services.

**Scrum Master**
A facilitator who coaches the Product Owner and Delivery Team members. The Scrum Master is also responsible for removing impediments to getting the job done.

Scrum defines four meetings where all roles come together to reach a specific objective:
Scrum Planning Meeting

Delivery Team negotiates with the Product Owner to plan and commit to the work they will pull from the Product Backlog. Scrum provides rules for the negotiation, such as the Product Owner having final say on the priority of items in the Product Backlog, and the Delivery Team having final say on how much work they are willing to pull from the Product Backlog.

Daily Standup

Short daily meeting where the Delivery Team checks in with each other to assess progress towards meeting goals. Scrum provides a three question agenda for the meeting.

Sprint Review

A celebratory meeting at the end of the Sprint to show off the work that has been accepted by the Product Owner.

Sprint Retrospective

A process introspection intended to apply lessons learned from the previous Sprint to the planning of the next Sprint. Unlike a project post-mortem the intent of the retrospective is to continually apply lessons learned in the current project to the current project.

There is a distinct difference between how Scrum defines roles and how rational methodologies define roles. Scrum defines roles by social interaction – Scrum Master, Product Owner, Delivery Team. Scrum specifically does not define any of the engineering roles usually associated with as software methodology, such as analyst, designer, programmer, or tester. Other methods, like Waterfall and RUP, tend to define roles by how an individual playing that role animates the process for transforming the inputs into outputs and not by how the individual interacts socially with other members of the team. In Scrum, therefore, individuals must have prior knowledge of software technical practices and must be able to negotiate the technical practices for creating software.
6.1.4 Agile Manifesto and Reconciling Perspectives

Reconciling Perspectives is a lens through which we can interpret the Agile Manifesto (2001a) and Agile Principles (AgileAlliance, 2001b). It is straightforward to connect the Converging stage to the Manifesto’s articles of “Individuals and interactions over processes and tools”, “Customer collaboration over contract negotiations”, and “Responding to change over following a plan”. In Reconciling Perspectives, individuals interact to reach out and negotiate a Consensual Perspective. Customers are one type of Acquirer and if we do not collaborate with them but, instead, use a contract to Cut-Off ourselves from our customers, there is minimal opportunity for early detection of Perspective Mismatches. The same is true if we strictly follow a plan that, again, cuts the team off from the organizational Ecosystem. Converging a Perspective Mismatch means that we are responding to change.

Connecting the Validating stage to the Agile Manifesto slightly blurs the interpretation because, while it is straightforward to connect the Manifesto article “Working software over comprehensive documentation” to Accepting, it is not straightforward to connect the activity of Bunkering to the Manifesto. Reconciling Perspectives suggests that individuals and teams need quiet focused time to Get the Job Done, which contrasts with the emphasis that the Manifesto places on interaction and collaboration. The need for open Communications is quite clear in our data because the root cause of many project challenges observed during this study was insufficient communication, both in volume and in diversity. However, many individuals were also challenged by too much communication and were unable to focus on Getting the Job Done. Too much of a good thing can also be a problem.
The need for focused time is perhaps best captured in the Agile Principle “Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done”, which could be interpreted as trust the team and give them the focused time they need to Get the Job Done. Agile software methods such as Scrum more explicitly capture the need for “focus” in the five Scrum values; moreover, the intention behind the interval extending from the time a Sprint is planned until the time the Sprint Demo is presented is that the team has the opportunity to work without interruption and make changes to their Sprint Backlog. During this time, the Scrum Master is tasked with protecting the team from outside interference.

6.1.5 Scrum and Reconciling Perspectives

Scrum’s prescribed meetings – sprint planning, daily stand-ups, sprint review, and sprint retrospective – are all opportunities for those participating in a project to check in and, if necessary, start Reaching Out. The sprint planning meeting is an opportunity to detect a Perspective Mismatch, and to Negotiate Consensus as the Scrum Product Owner (Acquirer) explains backlog items to the Delivery Team (Supplier). Once the Delivery Team commits to the sprint, then the Delivery Team focuses on Getting the Job Done, without being Exposed to interfering change (Bunkering). Team members are encouraged to regularly check-in with the team to prevent the Bunkering stage from disconnecting them from others (Cut-Off). When the Work Products for a story (Job) are complete, they are presented to the Product Owner for acceptance (Accepting). Scrum’s short Cycle Time of two to four weeks mitigates Surprise during Validating. The Scrum Master is a trained facilitator who owns the process and Shelters the team from outside interference.
During this study, there were many stories of failed Scrum projects, and Reconciling Perspectives may be used to diagnose those failures. The common thread running through the stories was of teams cutting themselves off and either not detecting the Perspective Mismatches, or not starting the Reaching Out process (Failure to Reach Out). They went “through the Scrum motions” but did not take advantage of the intent of Scrum. A condition cited for an effective Scrum team is a “strong” and available Product Owner, and our observations support this condition. We observed situations where other strong team members stepped in to fill the vacuum created by an unavailable or disengaged Product Owner.

One common failure was ineffective Scrum Masters who limited the effectiveness of the Scrum process by not taking ownership of the process, and by their inability to facilitate the process. Some merely “chaired” meetings rather than providing leadership. In many situations, ineffective Scrum Masters either let the team become exposed, or completely locked it down, cutting it off from the rest of the organization. While many of these Scrum Masters had Scrum Master training, the training often lacked emphasis on facilitation skills.

Another common failure was long Cycle Times. While most teams claimed to develop using short sprints, there were many stories and examples of sprint commitments not being reached, and incomplete work being rolled over into the next sprint. In other situations, the Work Products demonstrated during the sprint review were incomplete but gave a misleading impression to the Product Owner that they were done. The true state of the Work Products was not revealed until much later, usually at a significant product release milestone. The result was everyone acting as
if they were operating on short cycle times and assuming the benefits of a short cycle time while, in fact, they had very long cycle times.

### 6.1.6 Are Social Processes Sufficient?

Scrum is an example of a methodology that emphasizes social practices and does not provide technical guidance. Scrum does not explain how to elicit and analyze requirements beyond assuming a conversation with a quasi-omniscient Product Owner. Scrum does not specify what a requirements *Work Product* should look like beyond describing it as a backlog item, and does not specify how design and implementation are performed, or how validation and verification are achieved. Scrum assumes that members of the Delivery Team have knowledge of technical practices and are able to negotiate their technical practice norms.

For a software team to operate effectively, there must be a *Consensual Perspective* on the technical practices used to create *Work Products*, otherwise *Waste* is created. Individuals will follow their preferred idiosyncratic work practices and create their own idiosyncratic *Work Products*. This is precisely the problem rational software methodologies were created to solve in the first place (Parnas & Clements, 1985). The technical practices that are the common language of software projects can, without a technical methodology, deteriorate into a “Tower of Babel” of incompatible technical practices.
6.2 Software Development as a Socio-Technical Process

A number of software methodologists have advocated a “socio technical” approach to the design of software methodology (Robey, Welke, & Turk, 2001; Scacchi, 1984; Sommerville & Rodden, 1995). The reality is that all organizations are socio-technical systems because the individuals within them interact socially with each other and with the technology. Even in highly technically disciplined methodologies such as Clean Room (Mills & Linger, 2002), people still interact with each other and animate the process to Get the Job Done.

The intention behind socio-technical design is clarified in a fascinating account of the dramatic productivity improvement achieved at the Haigh Moor coal mine, where Eric Trist (Trist, 1963; 1981) credits not the introduction of new technology but how the workers socially re-organized themselves to change their highly structured work patterns to a more collaborative work style. Trist explained the improvements as an example of a socio-technical system where an organization’s objectives are best met not by the optimization of the technical system and the adaptation of the social system to it, but by the joint optimization of the technical and social aspects, thus exploiting the adaptability and innovativeness of people in achieving goals instead of over determining the manner in which these goals should be achieved (Cherns, 1976, p. 784).21

Socio-technical work systems are “...made up of two jointly independent, but correlative interacting systems – the social and the technical. The technical system is concerned with the

21 Chern’s analysis also addressed what every union leader knows: working to rule, that is rigidly following the documented work procedures, is a way to bring a firm to its knees. Is there any wonder then why well intentioned software developers do not follow rational software development methodologies?
processes, tasks, and technology needed to transform inputs to outputs. The social system is concerned with the attributes of people (e.g., attitudes, skills, values), the relationships among people, reward systems, and authority structures. It is assumed that the outputs of the work system are the result of joint interactions between these two systems. Thus, any design or redesign of a work system must deal with both systems in an integrated form” (Bostrom & Heinen, 1977, p. 17).

There are a number of principles for socio-technical design (Cherns, 1976, 1987; Clegg, 2000) and, while they are all relevant to the practice of software development, one principle stands out – the “Principle of Minimal Critical Specification” – which states that no more should be specified than is absolutely essential, and that we should identify what is essential (Cherns, 1976, p. 786). Cherns observed that, while it is necessary to be precise about what must be done, it is rarely necessary to be precise about how it is to be done, and that most organizations were far too precise in specifying how things should be done. This principle, with respect to software methodology, is frequently expressed in the Agile software development community as “…barely sufficient process” (Highsmith, 2002).

### 6.3 Refocusing Software Methodology Design

In their study of software development process performance, Sawyer and Guinan (1998) concluded that the benefit of a software methodology was to bring production aspects of

---

22 Another frequently cited similar principle in the agile community is “YAGNI” or “You Ain’t Going to Need It” which is typically used as advice for guiding the design of software systems by minimizing features and complexity of features.
software development into a social context, and that the formal coordination aspects of a methodology, such as meetings and documents, were valuable because they provided an opportunity to socialize (p. 562). From this conclusion they argued that simply ‘allowing’ discussion of people factors in software methodologies may be emphasizing the wrong aspect: putting the cart before the horse. Software methodologies should explicitly encourage socialization among developers and, as such, the design focus of software methodologies should be away from production-centered practices and toward socially-centered methodologies (Sawyer & Guinan, 1998, p. 563). Three policy guidelines for the design of a socially-centered software development methodology can be derived from Reconciling Perspectives:

- Define social roles and supporting practices,
- Favour short Cycle Times, and
- Support social scalability.

6.3.1 Define Social Roles and Supporting Practices

Technically-focused rational software development methodologies describe individual roles in terms of how they participate in the transformation of inputs into output work products. Individuals in the role of analyst take requirements as inputs and turn them into analysis models, individuals in the role of designer take analysis models as inputs and transform them into design models, and individuals in the role of implementer take design models as inputs and transform them into implementation work products. Furthermore, any social relationships between these roles are usually described in terms of the work products that are handed off in a linear manner from one role to another and not in terms of an interactive social process.
Reconciling Perspectives describes a social process between an Acquirer and a Supplier that is moderated by individuals in some form of Leadership role. While Scrum describes social interactions between the Product Owner, Delivery Team, and Scrum Master, it is necessary to explicitly and separately describe the leadership roles independently because in this study it was observed that it was not necessarily the individuals who were in the Scrum Master or Product Owner roles who were playing these social roles. Reconciling Perspectives identified four Leadership roles: Sheltering, Alerting, Drum Beating, and Coaching. A key Leadership role in this study that appeared to significantly influence the process was the presence of someone playing the Alerting role – often referred to as a “Boundary Spanner” or “Gate Keeper” in the literature.

The challenge with these social roles is that they are not something that can be inserted into an organization chart; simply nominating individuals into these roles without regard for their personal traits and experience may not yield the expected benefits. Nochur and Allen’s (1992) study of so called “nominated” boundary spanners discovered that they were not effective in transferring technology to other parts of the organization because the flow of information was one way and it effectively stopped with the nominated boundary spanners. This result is not surprising because, years before, Tushman and Scanlan (1981) observed in their study that successful boundary spanning needed individuals who were strongly linked both internally and externally and who were primarily “bench engineers” and not managers.

Five common attributes of those who effectively played the boundary spanning role were observed in this study:
• They were relatively experienced individuals, usually with at least 5 years of industry experience and at least 2 years within the organization. They either understood the system (“technical knowledge”) or understood how things were done (“organizational knowledge”). They tended to have the respect of their peers and were considered thought leaders.

• They were strong communicators, able to communicate issues to others in a manner they could understand.

• They had a broad picture of their work – not necessarily the complete big picture, but they understood how they and others fitted into a broader scope as opposed to only understanding their job role.

• They had the Personal Strength to reach out when they believed they discovered a Perspective Mismatch.

• They strongly believed it was part of their job to play this role and the organization gave them the flexibility to play it by not overburdening them with production work.

These Alerting roles are roles the individuals in our study assumed for themselves if there were no barriers to their taking on these roles. The most common barriers to these individuals were organizational barriers where such a role was not seen as part of the individual’s job description. Individuals were prevented from rising up to take a Leadership role because of a formal structure or because the responsibilities of their job description did not allow time to take on a Leadership role. This was a common complaint of individuals in the Product Owner role who were expected to produce technical work products while still being technically available to the team to help resolve Perspective Mismatches.
The *Sheltering* role, unlike the *Alerting* roles, was mostly a positional role with either a Project Manager or Scrum Master “officially” charged with *Sheltering* the team. However, there were cases where an individual who was not in an authoritative position also sheltered a group by drawing away interruptions from the other members of the group. This was an example of Coplien and Harrison’s “Sacrifice One Person” pattern (Coplien & Harrison, 2004, p. 85), where an individual sacrifices their productivity to enable the team to make forward progress toward *Getting the Job Done*.

### 6.3.2 Favour Shorter Cycle Time

Shorter *Cycle Time* reduces *Waste* and *Surprise* because it provides feedback that uncovers *Perspective Mismatches* while they are inexpensive to resolve. The common patterns across challenged projects in this study were long cycle times and low visibility. All things being equal, a shorter *Cycle Time* increases visibility because it engages the *Acquirer* and *Supplier* earlier, provides objective results to evaluate against the *Consensual Perspective*, and can encourage *Reaching Out*. Agile methods such as Scrum and XP use short time boxes usually limited to one to two weeks to *Get the Job Done* and, for large programs, releases are made every two to three months\(^\text{23}\).

---

\(^{23}\) This is in contrast to traditional practice where years could elapse between releases in a large program.
6.3.3 Support Social Scalability

Reconciling Perspectives is a scalable social process that’s occurs from the level of a pair of individuals negotiating a local technical implementation to the level of business units negotiating requirements and budget. In the previous section, Scrum was presented as one example of a methodology that is centrally focused on social processes. Unfortunately, Scrum as defined in textbooks is only a small team methodology, and most textbook descriptions provide little in the way of scaling advice. Yet, at a large scale, individuals must still have the Personal Strength to Reach Out, they must still negotiate a Consensual Perspective, and they must still have the quiet focused time necessary to create Work Products. Leadership still must provide the Coaching, Sheltering, Alerting, and Drum Beating whether the group is two people, twenty people, or over two hundred people.

Small projects require a lighter methodology, and larger projects require a heavier methodology. The concept of a scalable software methodology is not new; for example, RUP was an excellent attempt to create a scalable process framework. RUP was designed to accommodate the management of software development for a large national air traffic control system and it was assumed by its designers that RUP was scalable because it provided guidelines on how to tailor it down by removing processes, roles, and work products. In my personal experience, however, the problem I discovered with RUP is that it required a skilled software process engineer with a strong will to tailor down the process, because they had to know RUP in detail and know what practices could be discarded. The software process engineer required a strong will to argue in front of risk adverse stakeholders that the discarded practices were not needed. The result for one of my clients was the use of a methodology suitable for a national air traffic control system
for managing a ten person customer care voice response project. It is no wonder why many individuals find RUP limited, impeding, and heavy. This view is unfortunate because RUP provides excellent technical guidance.

Taking this lesson learned from RUP, and adhering to the socio-technical guideline of specifying only what is absolutely essential, we can take a different approach to creating a scalable methodology by scaling up rather than tailoring down. We want to specify the lightest methodology that will work and grow, – scale up – by adding both social and technical practices as the project grows such that the social and technical practices are supportive of each other.

Cockburn developed a grid for selecting a project’s methodology based on the size of the project, in terms of the number of people, and the criticality of the project, determined by the potential for loss should the system fail when in service (Cockburn, 2000). We propose a similar model that qualitatively guides scaling based on a project’s context. Kruchten identified eight social and technical factors that define the context of a software project and serve as drivers for software project methodology selection (Kruchten, 2011):
<table>
<thead>
<tr>
<th>Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>How big is the overall size of the system under development?</td>
</tr>
<tr>
<td>Stable architecture</td>
<td>Is there an implicit, obvious, de facto architecture already in place at the start of the project?</td>
</tr>
<tr>
<td>Business model</td>
<td>What is the money flow? Is what is being developed an internal system, a commercial product, a bespoke system on contract for a customer, or a component of a large system involving many different parties?</td>
</tr>
<tr>
<td>Team distribution</td>
<td>Often linked to the size of the project, how many teams are involved and how many are not collocated? Not being collocated increases the need for more explicit communication and coordination of decisions, as well as for more stable interfaces between teams.</td>
</tr>
<tr>
<td>Rate of change</td>
<td>How stable is the business environment, and how many risks (and unknowns) are being faced?</td>
</tr>
<tr>
<td>Age of system</td>
<td>Is this the evolution of a large legacy system (bringing, in turn, many hidden assumptions regarding the architecture), or is it the creation of a new system with fewer constraints?</td>
</tr>
<tr>
<td>Criticality</td>
<td>How many people will die or be hurt if the system fails? Documentation needs to increase dramatically to satisfy external agencies who will want to make sure that the safety of the public is assured.</td>
</tr>
<tr>
<td>Governance</td>
<td>How are projects started, or terminated? Who decides what happens when things go wrong? How is success or failure defined? Who manages the software Project Managers?</td>
</tr>
</tbody>
</table>

Table 20: Eight Project Contextual Factors (Krachten, 2011)

These contextual factors can then be plotted as social or technical factors that can guide project methodology decisions (Figure 30).
Figure 30: Scaling Model

In this scaling model, the Kernel Process uses *Reconciling Perspectives* to describe the social roles (e.g. *Acquirer*, *Supplier*, *Leader*) and the social practices for how these roles work together (*Reaching Out*, *Negotiating Consensus*, *Bunkering*, *Sheltering*, *Alerting*, *Drum Beating*, *Supporting*) to *Get the Job Done*. *Reconciling Perspectives* as a theory provides a framework for understanding the roles and social practices (Schon, 1983). The Kernel Process describes how a group will work together to deliver *Work Products* but does not specify the *Work Products* or the technical practices for creating the *Work Products*. Scrum is one example of a methodology that could serve as a Kernel Process. Social and technical practices are added as the social and technical scale of the project increases. Cockburn provides some guidance for scaling a software methodology in his four principles of methodology selection (Cockburn, 2000, p. 65):

1. A larger group needs a larger methodology.
2. A more critical system—one whose undetected defects will produce more damage—needs more publicly visible correctness (greater density) in its construction.

3. A relatively small increase in methodology size or density adds a relatively large amount to the project cost.

4. The most effective form of communication (for transmitting ideas) is interactive and face-to-face, as at a whiteboard.

Cockburn’s first principle recognizes larger systems need heavier methodologies, and is almost intuitively obvious. The coordination and communications needs of a project could, in theory, grow nearly exponentially with project size. The traditional approach in rational software methodologies is to limit coordination and communications complexity with technical practices that formalize coordination and communications. Sixty years ago, Trist observed the same phenomena occurring in the coal mines: “For several decades the prevailing direction had been to increase bureaucratization with each increase in scale and level of mechanization. The organizational model that fused Weber’s description of bureaucracy with Fredrick Taylor’s concept of scientific management had become pervasive” (Trist, 1981, p. 8)

Several authors have proposed scaffoldings of additional practices that can be constructed around Kernel Process methodologies like Scrum (Larman & Vodde, 2008; Leffingwell, 2007) to scale them from the team level to the organizational level. One frequently recommended scaffolding practice for managing inter-team coordination issues is the Scrum-of-Scrums, where the Scrum Master (or a designate) meets with their counterparts from the other Scrum teams. Another example of a scaffolding practice for managing larger scale projects is the creation of a Product
Council, which is a mechanism for the various team-level Product Owners to coordinate. The Scrum-of-Scrums and Product Council are two examples of formal structures for institutionalizing the Alerting or boundary spanning role. Such practices must be applied with care because they may simply replace one bureaucratic structure with another and, as cited previously, nominated boundary spanners may not be as effective as those who emerge into the role (Nochur & Allen, 1992; Tushman & Scanlan, 1981).

Cockburn’s last principle is a reminder that methodology weight can be traded off against personal communications, and it implies we can shed bureaucracy with better communication. If the group cannot engage in frequent and rich communication, then it must add compensating technical practices to the methodology. This is why a recommended software methodology design guideline is the definition of social roles. As the project scales up, the need to stay connected to the Ecosystem increases the need for the Leadership roles of Alerting and Sheltering because the sources of potential Communications increases. Results from this study, and observations from our professional experience, suggest experienced senior engineers are in a good position to assume Leadership roles if they are given the opportunity to do so. Unfortunately, the technical roles they often play within software methodologies, such as analyst or architect, impede the opportunity to take on the social role because their job description is strictly defined in terms of the Work Products they create rather than how they contribute socially to the team.
6.4 Is a Socio-Technical Methodology Sufficient? Enabling Positive Social Dynamics

One of the justifications for generating a theory is that it provides a rationale for individuals to explain what is happening around them (Schon, 1983, p. 274). *Reconciling Perspectives* provides all individuals participating in software development with a model for reasoning about their roles, practices, and processes from a social perspective. However, it is unlikely the creation and adoption of a socio-technical software development methodology informed by *Reconciling Perspectives* is enough because, despite high levels of agile knowledge and skills training, Agile transformations still experience a high failure rate (VersionOne, 2009). The problems I have observed are not traceable to a lack of engineering skill but are more associated with problems in *Social Dynamics*. One subject succinctly summed up the source for many of the difficulties experienced in software projects:

“But there just weren't enough conversations taking place” – Site 2, Subject 3

Simply getting people to talk to one another should benefit software team performance. Shared mental model theory predicts higher team performance when the members can have their mental models converge; people’s models converge when they observe and engage with each other. This suggests that enhancing the ability of those engaged in software development to reach out (*Reaching Out*) and negotiate (*Negotiating Consensus*) when they encounter impediments will enhance software team performance. Lawrence and Lorsche (1967) offer a solution in their recommendation for the use of multi-disciplinary teams, and for the use of personnel skilled in the art of coordination and conflict resolution. Scrum, and specifically Scrum’s Scrum Master
role, may be seen as a manifestation of this recommendation; however, the role of a facilitator is not a role that anyone can assume without appropriate training, and ineffective Scrum Masters were frequently cited as contributors to software team failure.

While individual traits such as extroversion and introversion will pre-dispose individual behavior, there is precedence from the aviation industry suggesting that training people how to reach out and negotiate can improve team performance. Cockpit Resource Management (or Crew Resource Management) (Wiener, Kanki, & Helmreich, 1995) is training that is given to all airline transport pilots, teaching those who have been trained as individuals how to work as a team. The program was instituted after investigating accidents blamed on “pilot error” in an effort to discover what was missing from pilot training. What was discovered was not a deficiency in so-called “stick and rudder skills” but an inability of the flight crew to function as a team.

The effectiveness of team training has led other professions where individuals must operate in teams, such as medicine, to adopt programs similar to Crew Resource Management training. This study demonstrates a strong need for managed communications in software development, and yet our profession provides little in the way of the necessary training. For example, the Certified Scrum Master (CSM) learning objectives (ScrumAlliance, 2011) state that individuals will learn the responsibilities of the Scrum Master role, including:

- serves the Product Owner and team,
- removes the impediments,
- coaches the Product Owner and team, and
protects the team.

While these are necessary learning objectives for individuals and teams using Scrum, and especially for those individuals stepping into the role of the Scrum Master, they are not sufficient because there is little opportunity within the CSM to teach or develop the skills necessary to fulfill these objectives. From this, we can make two recommendations for improving software team performance:

1. Team training should become part of the regular engineering and software professional education, and
2. For organizations practicing Scrum, greater attention should be paid to the selection of the Scrum Masters by either
   a. selecting individuals for the role who have demonstrated leadership and facilitation skills;
   b. providing facilitation training to stronger team members who have the respect of their peers;
   c. hiring or contracting individuals skilled in facilitation to serve as professional Scrum Masters.

Team training is an effective way to both instill skills and improve team processes, and it has effective outcomes: socialization, trust, confidence in team members’ abilities, and improved attitudes concerning the perceived effectiveness of team communication and coordination (Salas et al., 2008). Team training creates a facilitated opportunity for the team members to socially
interact, thereby creating the opportunity to develop shared schemas (Espevik, Johnsen, & Eid, 2011; Marks, Sabella, Burke, & Zaccaro, 2003).
Chapter 7

CONCLUSION

The motivation for this study was to understand why those involved in the process of software development do not embrace software development methodologies despite evidence that software development methodologies improve the software development process. I argued that part of the reason for this is that people do not find software methodologies useful because there is a gap between the processes described by the software methodologies, and the processes that people actually execute. If we understood how people really managed the process of software development, rather than trying to impose an artificial rational process on it, then we can create more useful software methodologies. My intention was, therefore, to generate a substantive theory explaining how people manage the process of software development that both serves as a framework for understanding the process of software development, and can inform the design of software methodologies.

In the final chapter of this thesis, I summarize Reconciling Perspectives. I discuss how Reconciling Perspectives answers the research questions, explains my original motivation, and how it may inform software engineering policy. I then summarize this study’s contribution and, finally, explain opportunities for further research.
7.1 Reconciling Perspectives: Summary

We discovered that the primary concern of people managing the process of software development is *Getting the Job Done* and that people use a process of *Reconciling Perspectives* when *Perspective Mismatches* impede *Getting the Job Done*. *Perspective Mismatches* can lead to the creation of *Waste*, which are *Work Products* that do not satisfy the expectations of *Acquirers* and *Suppliers*. *Perspective Mismatches* impede *Getting the Job Done* because an individual’s *Perspective* sets their expectation of the *Job*. As one project manager summarized the process:

“we all bring our own perspective, experiences and biases to the communications framework, events or however you want to call it um when we communicate. And so you know I may hear some - the same words as somebody else, but I’ll hear them completely differently. And it’s those – it’s - it’s getting everybody to understand things in a similar manner. Get everybody on the same page. I think that’s – that’s the biggest impediment is people walk away with misunderstandings and you haven’t clarified. And they just run with their assumptions or what the way they understand things. And then we either have to go and correct them or back out of it or redo um something. And I think that’s the thing that causes us the most pain. I spend the most time on in terms of preventing and mitigation” – Site 1 Subject 1 (follow-up) 21:30

*Reconciling Perspectives* is a two-stage process of *Converging* and *Validating*. When individuals recognize a *Perspective Mismatch*, they begin *Converging* their *Perspectives* by *Reaching Out* to engage others and if the others agree to engage then the parties begin *Negotiating Consensus* to create a shared *Perspective* or *Consensual Perspective* of the *Job*. While the *Consensual Perspective* removes the impediment to moving forward, it is only tentative because it is subjective; the
parties are assuming they have a better understanding of how the others see the problem and its solution because their Communications no longer indicate a Perspective Mismatch. Validating creates objective evidence that the Perspective Mismatch is reconciled when the Supplier constructs the Job Work Products during Bunkering and they are accepted by the Acquirer during Validating. During Bunkering, people prefer to work with quiet focus on creating the Job Work Products with minimal interruption and distraction.

![Diagram of Reconciling Perspectives](image)

Figure 31: Reconciling Perspectives
Waste is created when the Job does not get done and the delivered Work Products do not meet the expectations of either the Acquirer or Supplier, and Surprise occurs when there is significant unexpected Waste. There are four ways Waste is created in Reconciling Perspectives:

<table>
<thead>
<tr>
<th>Failure Path</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to Reach Out</td>
<td>The individual or organization noticing a Perspective Mismatch never reaches out, or other parties decline to engage.</td>
</tr>
<tr>
<td>Convergence Stalling</td>
<td>The negotiation never reaches an explicit Consensual Perspective and both Supplier and Acquirer continue to work with a Perspective Mismatch.</td>
</tr>
<tr>
<td>Isolation</td>
<td>During Bunkering, Suppliers rigorously filter all Communications thereby ignoring Communications that may either invalidate the Consensual Perspective or indicate further Perspective Mismatches.</td>
</tr>
<tr>
<td>Saturation</td>
<td>During Bunkering, the Supplier does not filter any Communications such that they become distracted and unable to focus on creating the Work Products.</td>
</tr>
</tbody>
</table>

Table 21: Reconciling Perspectives Failure Paths

The Reconciling Perspectives process is scalable and applies to all project Scopes and durations (Cycle Times), from the short conversation between developers to the protracted conversations between development and stake holder organizations.

The naïve conclusion is people managing the process of software development need to talk more to one another because the lack of Communications is a root cause of problems. Without Communications, there is no opportunity to detect Perspective Mismatches and there is no opportunity to negotiate a Consensual Perspective. There was no shortage of data in this study.
revealing how *Waste* is created because of the lack of *Communications*. As one analyst concluded about a severely challenged project:

“...*there just weren’t enough conversations taking place*” – Site 2 Subject 3

While *Communications* are necessary, more is not necessarily better because if the *Communications* are not focused on Converging the *Perspective Mismatches* to a *Consensual Perspective*, then the process risks generating *Waste* through *Convergence Stalling*. Many projects created *Waste* because no one was willing to make a timely decision.

“The customer had no project manager on their side to manage themselves. So when we sent our release plan over, I don't even think anybody read it. And um they just kept firing down change requests. And I went to them at one point and said um I'm very uncomfortable with this because we've now doubled the budget. And they're like, Well, don't - that's not your job to worry about. We'll worry about that. And I said, You know, I feel like I'm - you're like a blank cheque here, you know? And I think I was more worried about it than they were. And ah um but they had no project manager. So, they had no oversight, no governance, no discipline on their side” – Site 1 Subject 1 412:416

Excessive *Communications* are also disruptive because people cannot focus on *Getting the Job Done*. Software developers need quiet focused time to construct *Work Products* that *Get the Job Done* and interrupting *Communications* are disruptive.
“So, it just means time-wise ah you can't bug them about every little detail. They've got to do a lot of context switching because they have so many features they're working on” – Site 3 Subject 6

These disparate Communications needs create tension between the Converging and Validating stages of the process because people must concurrently be sufficiently open to Communications that may reveal a Perspective Mismatch while also remaining sufficiently focused to Get the Job Done.

The process is moderated by Social Dynamics, which is the capability of an individual or team to manage their Communications and balance the tension between the Converging and Validating stages. There are four properties of Social Dynamics:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal Strength</td>
<td>An individual’s ability to reach out, and also to know when to say no.</td>
</tr>
<tr>
<td>Experience</td>
<td>The knowledge gained over time working within a specific Ecosystem, and the familiarity an individual develops with others.</td>
</tr>
<tr>
<td>Leadership</td>
<td>The ability of individuals to establish a culture that helps individuals and teams manage Communications and moves the process forward.</td>
</tr>
<tr>
<td>Openness</td>
<td>The receptiveness of an individual or team to Communications and negotiation.</td>
</tr>
</tbody>
</table>

Table 22: Social Dynamics Properties

Leadership is a trait of an individual; it is a service an individual provides for others that influences how they engage in Reconciling Perspectives. There are four Leadership roles that influence how individuals and teams engage in Reconciling Perspectives:
<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supporting</td>
<td>Creating an environment that encourages <em>Reaching Out</em>.</td>
</tr>
<tr>
<td>Sheltering</td>
<td>Keeping individuals connected to the <em>Ecosystem</em> while protecting them from becoming <em>Exposed</em> (sometimes called buffering, or shielding).</td>
</tr>
<tr>
<td>Alerting</td>
<td>Preventing individuals from becoming <em>Cut-Off</em> by staying connected to the <em>Ecosystem</em> and bringing information that may indicate <em>Perspective Mismatches</em> to the attention of individuals and teams.</td>
</tr>
<tr>
<td>Drum Beating</td>
<td>Avoiding <em>Convergence Stalling</em> by moving the process forward and keeping negotiations focused.</td>
</tr>
</tbody>
</table>

**Table 23: Leadership Roles and Responsibilities**

### 7.2 Explaining Why People Don’t Use Software Methodologies

I cite a number of technology adoption model studies (Davis, 1985; Venkatesh & Davis, 2000) to argue that practitioners do not use software methodologies because they do not find them useful. In this study, my approach to discovering what makes a software methodology useful was to step back, go into the field, and ask practitioners “How do you manage the process of software development?” From their answers and my field observations, I constructed a substantive theory which explains software development as a social process. When *Reconciling Perspectives* is compared to the processes described by rational software methodologies, three explanations for why people do not find them useful emerge:
TABLE 24: Reasons Why Software Methodologies are Not Useful

To be clear, I am not arguing that rational software methodologies are useless because, without technical guidance, there is technical chaos. Sawyer and Guinan (1998) observed that the technical aspects of a methodology provided an agenda that informs software developers what they need to talk about and what issues they need to concern themselves with. In my personal opinion, every technical practice in a software methodology is a hard-won piece of experience that should not be casually dismissed by a practitioner simply because they believe it is inconvenient, slows them down, or restricts their creativity.
Rather, I am arguing that there is a social side to software development and that useful software methodologies will address it with more than just simplistic platitudes. As this study has illustrated, practitioners described software development processes in social terms, and rational methodologies have either ignored social processes altogether or substituted technical practices for social practices which, in turn, add unnecessary weight to the process (Cockburn, 2003). Therefore, we need to look at software development as a socio-technical process “...made up of two jointly independent, but correlative interacting systems – the social and the technical” (Bostrom & Heinen, 1977, p. 17). *Reconciling Perspectives* explains software development as a social process and therefore it can be used as a description of a social system that is correlative and interacts with the technical system.

### 7.3 Using Reconciling Perspectives to Inform the Design of Software Development Methodology as Socio-Technical Processes

The recommendation from this study for making software methodologies useful is to shift the focus from a strict rational and technical focus to a socio-technical focus where social factors are much more than just a platitude. *Reconciling Perspectives* provides a social process that explains a social system, and it can provide individuals with a common framework for understanding an important software development social process.

*Reconciling Perspectives* provides all individuals participating in software development with a model for reasoning about software development roles, practices, and processes from a social perspective; it provides a partial rational framework that answers the question “Why are we
doing this?” It is only a partial rational framework because *Reconciling Perspectives* does not explain roles beyond the abstraction of *Acquirer* and *Supplier* relationships and mediating leaders, and it does not provide an agenda for the negotiations. Three policy guidelines for the design of a socially-centered software development methodology can be derived from *Reconciling Perspectives*:

- Define social roles and supporting practices,
- Favour short *Cycle Times*, and
- Support social scalability.

### 7.3.1 Define Social Roles and Supporting Practices

Technically-focused rational software development methodologies describe individual roles in terms of how they participate in the transformation of input work products into output work products. *Reconciling Perspectives* describes a social process between an *Acquirer* and a *Supplier* that is moderated by individuals in some form of *Leadership* role. One role that, in our view, appeared to significantly influence the process was the presence of someone playing the *Leadership Alerting* role – often referred to as a “Boundary Spanner” or “Gate Keeper” in the literature. We observed some propensity for individuals with job titles such as Architect and Business Analyst (BA) to play these roles, though experience suggests the individuals in these roles do not take on the Boundary Spanning roles consistently.
7.3.2 Favour Short Cycle Time

All things being equal, a shorter Cycle Time increases visibility because it engages the Acquirer and Supplier earlier, provides objective results to evaluate against the Consensual Perspective, and it can encourage Reaching Out. Agile methods such as Scrum and XP use short time boxes, usually limited to one to two weeks, to Get the Job Done and, for large programs, releases are made every two to three months.

7.3.3 Support Social Scalability

Following the socio-technical principle of “Minimal Critical Specification” (Cherns, 1976, 1987; Clegg, 2000), a methodology should specify only what is absolutely essential. Knowing what is essential is challenging because Reconciling Perspectives is a scalable social process that occurs from the level of a pair of individuals negotiating a local technical implementation to the level of business units negotiating requirements and budget. We argue for using Scrum and Reconciling Perspectives as a minimal software process and scaling up by adding both social and technical practices. Cockburn (Cockburn, 2000) and Kruchten (2011) provide design guidelines for scaling software methodologies.

7.3.4 Team Training

While team training would not normally be regarded as part of software methodology design, it is highly likely that methodologies will be far better received and more successful if training is an integrated part of the software methodology and teams are trained together. The experience
of the airline industry demonstrates that “stick and rudder” skills are not sufficient for enabling effective social processes.

7.4 Using Grounded Theory for Software Engineering Research

One of the goals of this research is to offer greater insight into the use of the Grounded Theory method as a research method for software engineering research. The appeal of Grounded Theory is that it offers the opportunity to understand a problem from the point of view of the participants. I cannot over-emphasize the value of this because part of the motivation behind this study is the perception that there is a gap between the software engineering research agenda and how practitioners construct software. Had I pursued a more traditional logico-deductive approach to this study, to answer the question why do people not use software development methodologies, I would have likely developed a set of hypotheses around software methods being “too heavy” and then designed a survey instrument using Ajzen’s Theory of Planned Behaviour (TPB) (Ajzen, 2006) to gather data to either confirm or deny the hypothesis. The result would have been a nice study that suggested most methods are “too heavy” and have too many practices and do not support the way software is actually constructed. But I would not have developed a theory that describes how software is constructed from the perspective of participants, and I would not have been able to develop a theory that explains that process. I would not have developed insights into the tensions in the process, or understood the role Social Dynamics plays in mediating the process.
Constant comparison meant that concepts had to “earn their way into the theory” and many times this prevented me from pursuing my own “pet diagnosis” of situations. Many, many times I wanted to push the development of the theory towards what I believed were the key software development issues: the complexity of software, and the role of uncertainty. In this study, the participants controlled the research agenda, always taking me back to their central concern.

Six years ago, when I began the design of this study, the Grounded Theory method was often used in software engineering research to legitimize research that, in many cases, was just data categorization. Part of the motivation for this study was to demonstrate the utility of the Grounded Theory method as a software engineering research tool and to provide an exemplar of a software engineering grounded theory. Since that time, I am happy to say, a small number of other software engineering researchers have made effective use of Grounded Theory to inform software engineering practices through constructing useful theories (Hoda, Noble, & Marshall, 2010; Martin, Biddle, & Noble, 2009).

7.5 Contribution of this Study

From one point of view, this is yet another study demonstrating software development is a social process. The literature review shows this is a stance that has been argued and demonstrated since at least 1978. What this study contributes is a substantive theory that explains a social process that can be used to guide software engineering policy. This study is an explanation of the process of software development from the perspective of the practitioner. The primary contribution of this study is an explanation for how people involved in the process of software development
the Job Done by reconciling Perspective Mismatches. The process explains how Perspective Mismatches are reconciled to result in accepted Work Products and how un-reconciled Perspective Mismatches can result in Waste. The process is moderated by Social Dynamics which is the ability of individuals and teams to manage their Communications.

Several patterns of Waste creation result from Social Dynamics that limit Communications. It is not a dramatic contribution to suggest that people involved in the process of software development need to talk to one another because celebrity authors such as Tim Lister (Lister & DeMarco, 1987) and Gerald Weinberg (1986) have argued this for nearly thirty years. The new contribution this study highlights is not simply the need for more Communications but the need to manage Communications. Individuals creating software need quiet focused time to construct Job Work Products, yet they must also remain open to Communications that reveal impeding Perspective Mismatches about the Job while, at the same time, not becoming overwhelmed by distracting Communications that takes their focus away from constructing Job Work Products.

A significant contribution of this study is highlighting this Communications tension and the need of individuals and teams to manage this tension if they are to avoid creating Waste. I constructed four Social Dynamic properties that moderated the process: Personal Strength, Experience, Leadership, and Openness.

This study contributes to our understanding of why software methodologies are not used. When the process described by Reconciling Perspectives is compared to the process described by rational software methodologies, we quickly see the gaps between the technical focus of rational software
methodologies and the social focus of those managing the software development process. This study describes how *Reconciling Perspectives* may be used to inform the design of a socio-technical software development methodology.

A final contribution of this study is a demonstration of Grounded Theory as an effective and useful software engineering research methodology and the benefits of a diverse cross disciplinary research supervisory committee.

### 7.6 Future Research

The conclusion of this study reminds me of Pirsig’s alter ego ghost, Phaedrus (2011), in *Zen and the Art of Motorcycle Maintenance*, who went insane with the scientific method because the answer to a question simply generates more questions. Rather than go insane, I prefer to regard this more as an opportunity to learn, and this study offers several opportunities for learning, such as:

- Gather additional data to elaborate *Reconciling Perspectives* to address some of the limitations of this study.
- Design and deploy a socio-technical software development methodology influenced by *Reconciling Perspectives*.
- Create the opportunity for practitioners to conduct academic research.
7.6.1 Elaborating Reconciling Perspectives

In *Reconciling Perspectives*, the *Acquirer* and *Supplier* can be members of the development team requesting technical features from one another, and I collected observational data that supports this relationship; however, many of the stories told to me by interview subjects described relationships with *Acquirers* from the business side of the organization, and in this study I was not able to interview individuals from the business side of the organization. While I have some observational data of individuals from the business side of the organization, I do not have any data that reveals what they think about the process of managing software development. There is an opportunity to collect additional data that may re-shape *Reconciling Perspectives* and answer questions that arose during analysis:

- Do *Acquirers Bunker*?

- How do anti-social practices such as bullying affect the process of software development?

- Why do people become *Exposed*?

- How closely should the end customer or stakeholder interact with the developer?

7.6.1.1 Do Acquirers Bunker?

A key finding is that *Suppliers* need *Bunkering* when they are creating *Work Products*. However, I do not have an explanation for the role of the *Acquirer* during *Bunkering*. The *Acquirer* actively participates in *Reaching Out, Negotiating Consensus*, and *Evaluating*, but not in *Bunkering*. Software developers often played the role of an *Acquirer* when they requested another developer to do a *Job* for them. This makes it difficult to determine whether a software developer was playing the role of a *Supplier* or an *Acquirer* during *Bunkering*. 
There is some evidence suggesting that Acquirers do indeed Bunker because a common complaint of Suppliers was about inaccessible Acquirers. There is also anecdotal evidence to suggest that those in the business side of the organization do not Bunker and therefore may not appreciate the need for engineering to have quiet focused time to Get the Job Done. If this is the situation, then we may be able to explain some of the tensions observed in this study in the relationships between Acquirers on the business side of the organization and Suppliers.

7.6.1.2 Cut-Off and Bullying: How Do Anti-Social Processes Affect Software Development?

A common theme in many of the stories of failed projects was Communications Cut-Off, where an entire development team cut themselves off from other individuals and teams. “Going Dark” is a term coined by Jim McCarthy (McCarthy & Gilbert, 1995) to explain this phenomena. Reconciling Perspectives offers some explanation for this, people need quiet focused time to Get the Job Done; but it should be obvious that going dark can create Waste and Surprise and, according to McCarthy, it usually does.

Some cases of “going dark” encountered during this study can be explained by lack of Experience or low Personal Strength or, from our discussion of social cognition, simply by an inability to recognize the need for Reaching Out. However, we also collected data of high Experience and high Personal Strength individuals and teams also “going dark”. So what is going on here because people who should seemingly know better were not Getting the Job Done? There were many stories where “going dark” appeared to be an explicit passive-aggressive tactic employed to defend an agenda. We can hypothesize that the individual or team was aware of the existence
of a Perspective Mismatch but refused to Reach Out, or to accept Reaching Out, because they did not wish to compromise (Broadening) their agenda. In straightforward terms, it appears the Supplier tried to get their way by bullying the Acquirer. Our industry has many anecdotal stories of managers bullying software developers to accept impossible schedules – and I collected some data for this in this study as well. However, we do not tend to think of software developers bullying their managers or customers.

7.6.1.3 Saturation: Why Do People become Exposed?

Another common theme in challenged projects and late deliveries was not filtering Communication and responding to every interruption. People allowed themselves to be interrupted when they were well aware that such interruptions would jeopardize Getting the Job Done, or knowingly took on more work than they could handle. Fear of losing one’s job if they do not say yes to every management request is not an answer to this question, yet people knowingly set themselves up for failure. This problem was certainly prevalent with younger less experienced developers, but the consequences were more serious when more experienced developers became Exposed or allowed themselves and their teams to become Exposed.

7.6.1.4 Customer Collaboration: How Close is Too Close?

Closely related to the question of why do people become Exposed is understanding more about the relationship between the Acquire and Supplier when the Acquirer is the actual end customer. The Agile Manifesto asserts the benefits of customer collaboration, and conventional Agile wisdom argues for close relationships with our customers. But how close is too close? We collected data in this study showing that customer-facing teams were more stressed than those
teams that were sheltered from direct customer contact. At Site 1, the team was sheltered from the client by a very strong project manager. At Site 2, there was a distinct difference between the sheltered product development team and the exposed professional services team. At Site 3, teams that were directly exposed to customers had significant challenges *Getting the Job Done*. This appears to run counter to the popular Agile philosophy of an actively engaged customer, and philosophy of co-location. In theory, there are Agile rules of engagement between customers and development teams, but if we have learned anything from this study, few people in software development play by the rules.

While there have been studies of the relationship between the customer and the development team, they did not specifically address this issue of interference. Such a study would help create policy for a more practical relationship between customers and development teams.

### 7.6.2 Create and Deploy a Socio-Technical Software Development Methodology

The motivation of this thesis was to generate a theory that will help inform the design of software development methodologies. The result is *Reconciling Perspectives*, which I have argued can be used to inform the design of a socio-technical software development methodology. The question, though, is what does a socio-technical methodology look like? How is it structured, and how is it deployed?

---

24 And in practice
I certainly do not believe that attempting to incorporate social practices into a methodology following a formal meta-model such as SPEM (OMG, 2005) will work, or even that it is desirable. Our failure with OpenUP (Kroll & MacIsaac, 2006) is a clear demonstration of this. Instead, I recommend employing an empirical approach, following the principles of validated learning, and what has widely become known as “Lean Start-up” (Ries, 2011), to design and deploy a socio-technical methodology. A good start would be using Reconciling Perspectives to explain and highlight the social aspects of existing software methodologies such as Scrum. Feedback from practitioners can then be used to design socio-technical methodologies with greater scope.

One structured approach that may be worth investigating is to follow and enhance Coplien and Harrison’s (2004) lead and develop a pattern language for socio-technical software development methodologies.

7.6.3 Are Software Developers and Pilots the “Same Breed of Cat”?

This dissertation cited airline pilot Crew Resource Management (CRM) training as an example of where social skills training improved the performance of a technically skilled work group. CRM training has been adapted to medical professionals – specifically to those working in the emergency room. While I am an aviation enthusiast, and was a pilot, I have not experienced CRM training myself. This training was cited because I personally believe that engineers and pilots are the “same breed of cat” – both have a love of technology and take incredible pride in their technical skills, and yet both need to use their social skills to fulfill their job. This assumption, however, needs to be validated. The first step is to study actual CRM programs and
gather data from participants to better understand these programs and their relevance to software engineering.

7.6.4 What Can We Learn from Others?

During the literature reviews for this study, one thought I had repeatedly is that our discipline of software engineering is at least 50 years behind the curve as far as our ability to apply the socio-technical lessons learned in other disciplines, especially considering Eric Trist visited the Haigh Moor coal mine in 1947 (Burns & Stalker, 1961), or considering that it was in 1967 that Lawrence and Lorsch declared that an appropriate mode of organization for innovative organizations was the use of “multidisciplinary project teams and the appointment of personnel skilled in the art of coordination and conflict resolution” (Lawrence & Lorsch, 1967). Many of the practices we are embracing as bold and innovative, such as “lean”, have their origins in other industries, such as automotive manufacturing (Ohno, 1988).

There is much we can learn from other disciplines, and an effort to make such knowledge more accessible to software engineering researchers would be a good first step. This may proceed in the form of a systematic literature review of organizational behavioural research on small groups and innovation. Another area of interest may be to conduct a review of research related to “relational coordination” (Gittell, 2001), something which Southwest Airlines has used to successfully out-compete legacy carriers (O’ Reilly & Pfeffer, 1995).
7.6.5 Creating Opportunities for Practitioners to Conduct Software Engineering Research

While conducting this study, economic concerns (and also a desire to work) forced me to return to work, and during the time I was working, I had access to amazingly rich data that I could not use for this study\textsuperscript{25}. Advice given to me during this study was to not confuse my role as a researcher with my role as a consultant. Among my professional colleagues, there are a number who would be strongly interested in pursuing graduate degrees if they could combine their daily work with their research. Software engineering research is generally not a funding agency priority (at least, not in Canada), and the opportunity for practitioners to collect and analyze high quality field data as part of their day-to-day job cannot be overlooked. Therefore, we need to investigate ethics guidelines that would allow field practitioners to collect data as researchers while they are professionally engaged with clients.

\textsuperscript{25} I think I now know how a crown prosecutor feels when their evidence is ruled inadmissible.
BIBLIOGRAPHY


Fish, S. E. (1980). *Is There a Text in This Class?: The Authority of Interpretive Communities*: Harvard University Press.


Voas, J. (1997). Can clean pipes produce dirty water


An Invitation to Participate in a Field Study of Software Development Issues and Their Resolution.

Principal Investigator:
Dr. Philippe Kruchten, P. Eng., Professor
Department of Electrical and Computer Engineering
E-mail: [Redacted]

Co-Investigators:
William S. Adolph, student at UBC / ECE, pursuing a Doctoral degree in Applied Science

Overview
We are inviting your organization to participate in a field study of software engineering practices and social dynamics. This study will require a researcher regularly visit your site to conduct observations and interviews.

Purpose:
The purpose of the study is to develop a substantive theory of how people manage the process of software development. We are specifically interested in understanding the social factors that influence individual and team performance. Our intent is to use the knowledge gained during this study to develop guidelines that can be used by the industry for the design of software development methods.

We are inviting your organization to participate in this study because your organization is actively engaged in software development, is representative of the industry, and possess the expertise that is relevant to this study.

Study Procedure

Description
This is a field study where the co-investigator shall be frequently present onsite collecting field data. The researcher will require access to team members but will not actively engage with team or participate in the project. Therefore the organization does not need to provide the researcher...
with a desk and computer resources. Study data shall be gathered using three collection mechanisms:

Participant observation: The study co-investigator shall take a “fly-on-the-wall” approach observing team activities. It is expected the co-investigator will observe team planning and technical meetings. In addition, the co-investigator shall on occasion locate himself in the team area and observe “day-in-life” activities of team members.

Semi structured interview: Some subset of the team under observation will be asked to participate in study interviews. An interview will require approximately one hour of the subject’s time and the subject may be interviewed between 2 to 3 times during the course of this study.

Document Analysis: With the permission of your organization, the researcher will collect and analyze documents deemed relevant to the process of software development. This will include formal documents and memos related to the specification of the software development process and sample work products resulting from the process (e.g. requirements documents, design documents, test plans).

Process
At the commencement of this study, the co-investigator will negotiate with your organization an appropriate start date. On start date the researcher shall make a brief presentation to introduce himself to the team and to describe the study. The researcher shall then spend two to three days on site observing the team to become familiar with the team. During this initial observation the researcher shall identify individuals roles on the team and begin the interview selection process. Also during this period the researcher shall request the senior project manager or engineer brief him on the software development process.

After this initial observation period, the researcher shall be on site periodically. The researcher shall negotiate an appropriate schedule for the researcher. This schedule shall include regularly schedule observation dates (e.g. one day every other week) and ad hoc observations scheduled around specific significant project events (team planning meetings, design meetings, defect triage meetings, etc).

Team Selection
The researchers shall negotiate with you to select a team appropriate for this study. An appropriate team is a team involved in a non-trivial project (greater than 50 staff months effort, has some element of technical complexity and risk). We would like to observe all development activities (e.g. requirements, design, implementation, test). Therefore a suitable project shall have an expected duration of approximately six months or be following an iterative software development methodology.

Team for this study means all persons having a direct interest in the software development project. This includes software developers, quality assurance, product management, project management, and relevant stakeholders.

1/14/2009 2/4 Version 1.2
Resource Commitments
While the co-investigator will be taking a “fly on the wall” approach with data collection it is misleading to state the co-investigator will not have any interaction with the team or not require any time on the part of team members. The study shall require some time from team members for activities related to the administration of this study which may include planning times for the co-investigator to be on site, scheduling interviews, and requesting assistance to discover relevant documents.

Sharing and Validation of Observations
The researcher shall share a summary of his observations with the organization and participating individuals to:
- provide an opportunity for the organization to comment on the observations
- validate the observations, and
- share the observations with the organization

Confidentiality:
No one other than the named investigators will have access to field notes and interview transcripts. The co-investigator shall present a summary of gathered data, but no personally identifying information shall be reported.

Benefit
The participating organization will receive two intangible benefits from participating in this study:
1. Third party observation of software development process that may assist in identifying sub-optimal processes. While this is an observational study, it is not intended to identify specific inefficiencies in the software development process. However, the summary of observations that will be presented to the organization may help identify sub-optimal processes.
2. Goodwill from participating in a study to identify key concerns of those involved in software development.

Contact for information about the study:
If you have any questions or desire further information with respect to this study, you may contact the co-investigator, William (Steve) Adolph (###-###-####).

Contact for concerns about the rights of research subjects:
If you have any concerns about your treatment or rights as a research subject, you may contact the Research Subject Information Line in the UBC Office of Research Services at 604-822-8598.

Consent:
Your participation in this study is entirely voluntary and you may refuse to participate or withdraw your organization from the study at any time. Your signature below indicates your
consent and support for this study on behalf of your organization and your organization's permission for the investigator to work onsite during this study.

Consent on the part of the organization to participate in this study should not be construed as providing blanket consent for individual members of the organization to participate in the study. The investigator must obtain the informed consent of the specific individuals within the organization who wish to participate in the study.

Subject Signature Date

Printed Name of the Subject signing above

1/14/2009

4/4

Version 1.2
Consent form for participation in the study:

A Grounded Theory of Software Development Issues and Their Resolution.

Principal Investigator:
Dr. Philippe Kruchten, P.Eng., Professor
Department of Electrical and Computer Engineering
E-mail: [Redacted]

Co-Investigators:
William S. Adolph, student at UBC / ECE, pursuing a Doctoral degree in Applied Science

Purpose:
The purpose of the study is to develop a substantive theory of how people manage the process of software development. We are specifically interested in understanding the social factors that influence individual and team performance. Our intent is to use the knowledge gained during this study to develop guidelines that will be used by the industry for the design of software development methods.

You have been invited to participate in this study because:
- the organization you work for has agreed to participate in this study, and
- you have knowledge and experience of software development that is relevant to this study.

Study Procedure:
Study data shall be gathered using three mechanisms:

Participant observation: This will take place over an extended period of time (from one month to potentially 6 months) during which time the co-investigator will be periodically on site observing your team interactions during a specific software development project. The co-investigator shall at times be present in team meetings and at other times simply situated onsite observing the day to day team interactions. The co-investigator will create field notes recording his observation. The co-investigator shall be an observer and not engage with you in software development activities.

No personally identifying information shall be recorded in the field notes. The field notes shall not be available for review by anyone other than the researcher and his supervisor. However, the researcher will provide you with summary of his observation. The team shall be invited to comment on the accuracy of the observations.

Semi structured: You may be asked to participate in an interview. You may decline.

11/25/2008 1/3 Version 1.0
interview: participation.

The co-investigator will conduct the interviews. The interviews are semi-structured and shall require approximately one hour of your time. At this is a longitudinal study, you may be requested to participate in two to three interviews during the course of the study. You may decline any interview request during the course of the study.

The investigator will make an audio recording of these interviews and also take field notes during the interview. During the interview, the only personal information that will be requested is:

- your post-secondary education,
- work experience,
- job role, and
- how you demographically identify yourself.

At any time you may decline to answer a question or terminate the interview.

You will have the opportunity to comment and provide feedback on the analysis of their interview.

With the permission of the organization under study, the researcher will collect and analyze documents deemed relevant to the process of software development. This will include formal documents and memos related to the definition of the software development process and work products resulting from the process (e.g. requirements documents, architectural documents).

Confidentiality:
Data collected from the interviews will be analyzed and may be used as part of the graduate thesis for co-investigator, William Adolph, but your identity will be kept strictly confidential and will not be used in any published work. The completed consent form, with your name and signature, will be kept separate from all survey results and other documents to protect your identity. You will be assigned an ID number and all data will be identified only by this ID number and stored in a locked filing cabinet in the principal investigator’s office, or on a password-protected disk. Only the named investigators will have access to these documents.

Audio data shall be transcribed using a professional transcription service. Only the named investigators shall have access to the interview transcripts. The audio data, transcripts and field notes shall also be kept in a locked filing cabinet in the principle investigator’s office. A back-up copy of the audio data will be made and stored in the co-investigator’s safety deposit box.

Remuneration/Compensation:
No compensation shall be provided for participation in this study.

Contact for information about the study:
If you have any questions or desire further information with respect to this study, you may contact the co-investigator, William (Steve) Adolph ( Contact Information ).

11/25/2008 2/3 Version 1.0
Contact for concerns about the rights of research subjects:
If you have any concerns about your treatment or rights as a research subject, you may contact the Research Subject Information Line in the UBC Office of Research Services at 604-822-8598.

Consent:
Your participation in this study is entirely voluntary and you may refuse to participate or withdraw from the study at any time. Your signature below indicates that you have received a copy of this consent form for your own records.

Your signature indicates that you consent to participate in this study.

________________________________________________________________________________________

Subject: Signature Date

________________________________________________________________________________________

Printed Name of the Subject signing above
**H08 0238 A Grounded Theory of Software Engineering Issues and Their Resolution – Interview Guide**

**Personal Profile Questions**

- Your job function and duties?
- Experience?
- Professional background?
- Education and Training?
- Demographically how would you classify yourself?

**Initial Questions**

1. What is the process of software development for your project?
   a. How did you begin this process?
   b. How have things changed over time?
   c. What surprised you the most? (Look for positive and negative surprises)
   d. Could you describe a situation where you knew how things worked?

2. What are the symbols/elements associated with the process of software development in this organization?
   a. What are the elements of software development that are important to you?
   b. What elements contribute to how you manage?
   c. How did you use those elements in software development?
   d. What events stand out for you?

3. How does this change over time since you joined this organization?
   a. How have you changed?
   b. What are your views about those changes?

**Potential “Probe” Questions**

**Team structure and communication mechanisms**

- What are your specific team roles and responsibilities?
  - How was this established?
  - What decisions do you usually make in your role?
• Who do you communicate with in the course of your job? (who do you send communications to? who do you receive communications from? How did you know?)
• What are the (a) formal and (b) informal channels through which you receive/send information? (Team structure?)
• How do you know what work you have to do?
• How do you report progress? For the people you communicate with, what is their role in the project?

Workflows and task dependencies
• What are the ‘items/resources’ (documents, information, software, budget, people’s time?) you need to do your job?
  o What are your expectations of other peoples’ work?
  o To what extent do you rely on them doing their job as you expect?
  o Do the people you receive ‘items/resources’ from know what you need/expect of them?
  o What are the ‘items/resources’ (documents, information, software, solutions, time?) you produce or make available so that other people can do their job?
• For any of these items/resources, do some of them have to conform to predefined standards or requirements if they are to be of use?
• What are your priorities in doing/delivering your work (e.g. what aspect of detail do you devote most attention to?)

Communication patterns with respect to above dependencies?
• Who/What/When/Where/Why/How analysis of prevalent communication processes? e.g. how are any/all of the following used? email? configuration management tools? project database? project plan? timesheets?
• How often prior/after the introduction of methodology did team members collaborate directly and informally? (e.g. whiteboard).

Examples of co-ordination in context?
• Describe one or more occasions recently where you were involved in or witnessed good or poor co-ordination... e.g. project clarification situations?, working under timescale pressures?, conformance to quality procedures/ standards? What happened? Where did it occur? Who (what roles) were involved? What was the outcome? Why was the interaction considered to be effective or ineffective? What were the implications of this incident for project team performance?
APPENDIX D  PARTICIPANT OBSERVATION PROTOCOL
<table>
<thead>
<tr>
<th><strong>Appearance</strong></th>
<th>clothing, age, gender, physical appearance.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Verbal behavior</strong></td>
<td>Who speaks to whom and for how long; who initiates interaction; languages or dialects spoken; tone of voice</td>
</tr>
<tr>
<td>and interactions</td>
<td></td>
</tr>
<tr>
<td><strong>Physical behavior</strong></td>
<td>What people do, who does what, who interacts with whom, who is not interacting.</td>
</tr>
<tr>
<td>and gestures</td>
<td>How people use their bodies and voices to communicate different emotions; what individuals' behaviors indicate about their feelings toward one another, their social rank, or their profession. Who defers to whom?</td>
</tr>
<tr>
<td><strong>Personal Space</strong></td>
<td>How close does the subject stand to others?</td>
</tr>
<tr>
<td></td>
<td>What individuals' preferences concerning personal space suggest about their relationships</td>
</tr>
</tbody>
</table>
APPENDIX E       SAMPLE CODED INTERVIEW
Interviewer: Steve A. 

Steve: I'd like to look at a little bit more this design review board. You were saying there were incidents where people were making changes that were not necessarily desirable. Were you involved in any of these? Or were you...

Sub 3: This is more in a phase when I was a QA. So sort of starting out I did my first project at the company. There was a fair amount of stuff. It was just a huge rush to get everything done. That—we didn't have a single QA at the company at that point. We had only one product manager who was really working in the role of sales engineer—a lot of the time. So, there wasn't particularly a body or you know a person to go and speak to about various requirements was what was asked of you. So that's where you—kind of realized a lot of kind of QA stuff. When you see something that's very young and not very logical to come up and ask what's this thing—how is it acting? I don't think that it makes sense. That also introduces some other problems because occasionally you know you'll have different QA's that have different notions of how the system should work. And obviously not having customer interaction sometimes those notions are somewhat pigeon-holed. So sometimes you'll also introduce desirable functionality that way—by having people make up requirements that are lacking to begin with. And in that project I think that happened with a lot of developers. The—they were given a very broad overview of what they needed to do. But a lot of the lower-level stuff which really should not be designed by a developer—they had no use cases to work against. No user stories to work with. It was the extent of basically given you know they give you the name of the feature and tell you what the feature does and you go off and develop it. And you have no idea about the integration points with other aspects of the system. So that leads to a very messy situation. So I've seen it in that instance.

Steve: Sorry. Just making some notes to myself. Have you been involved in the—in situations where for example you've spoke to the developers—a use case—user interface specifications—and you know they've sort of given you the nod—yeah—yeah—yeah—and then gone off and come back to you and said—what the bleek is this? You—can you tell me some of the stories behind that?

Sub 3: In my instance I wouldn't say that I've had that experience. I've seen slight deviations. But the fact that we work with iterations—at least in the projects that I've worked on so far as a BA they've been pretty locked solid so you assign the work for 2 weeks—we agree on it—I'll see at the end of 2 weeks minimum—typically we tend to review the build actually every single morning. So that's a bit of a comment. Final aspect of me (laughing). So I like to see what they've done every night. I get to see in their daily reports what they've done. It also allows me to give them a faster feedback on maybe some places where they're deviating from requirements. I also realize that my requirements are not perfect—so that even though I provide use cases and different UI that there are different scenarios that I don't account for necessarily and I try not necessarily to wait for QA to get there. But by doing kind of daily—you know small kind of regression testing in areas that they've done it allows me to see maybe some holes in
Monday August 31 2009

SAP
- Staff meeting

- Start with a surprise that mayor client has a "hang"

- Structure of meeting (regular ceremony)

- Follow up from previous meeting when Kevin was not present - on holiday

- Issue re the hang - argument as to whether this is a known hang.

- Belli story of cruise ship - except fully discusss ship's technology - Bow thruster and stabilizer pm.

- Continues follow up from last week staff meeting #Is.

- Discussion of setting plan for upcoming show - concern expressed about country...the "I" dead seat in their inventory.
- Sarcastic humour regarding delay of 18 months to Sept 21 but no one will move on Sept 21.
  - Manager (louell) asks if he can schedule his project this way.
  - Louell: "It seems kind of silly to move work for the sake of a week.
  - Discussion about who "won" and who "lost" who got the good seat and who got the crapping seat.
  - Score keeping.

Next agenda item: customer (threads out of memory)
  - We want to play with these parameters to see what happens.
  - (Exploring the problem uncertain)
  - Talk about safety features to fail elegantly.
Howell constantly probing
question like:
-did we do X?

PM also asserts priorities - given
direction (Forbush driving agenda)
- he's driving the agenda
but necessarily forcibly

> giving focus.

Project manager provides background to help team form
Howell explains background behind
a customer request
(Clarification - Shell wants demo)

Focus to explain

Howell provides more background
on the character of other
 customers - “Watch these guys
they are grasping at straws”
explains why the process is the
way it is - need to use the
person as a shield to drive
another group.

"I can’t be the spokesperson
for those guys" conflict

Team conflict or perspective conflict

316
The only place we can call ourselves agile is in development.

Discussion of requirements process and how to integrate existing requests.

- Significant concern over cut to travel budget.
- Concern over low margins and impact on xerox party.
- Internal transfer - point we have 2 heads and how to make use of this to the best interest of the team.
Perspective conflict "I don't think he got what I am talking about."
Negotiating Consensus

Added by Steve Adelphi, last edited by Steve Adelphi on Oct 28, 2017

What is it?

Negotiation is a communications process for a requester and a provider to determine what the Job is. Negotiation occurred at all levels and all scales, from CTOs negotiating with customers product features to individual developers negotiating an interface. It seemed everyone was always negotiating, negotiating, negotiating permeates almost every aspect of software development, clients negotiate features with the development team. Development teams negotiate resources. Developers negotiate APIs and allocation of behaviour with their colleagues and other teams.

Dimensions

influencing

- Transparency
- Leadership

- Single Phase or multi-phase - related to Feedback - Visibility and Sheltering. Multi Phase negotiations occur when negotiations are open, Feedback - Visibility is generated quickly. Coarse grain/Fine Grain
- Participation - Collaborative or imposed (simply agreed to what was given them) - did the individual participate in the negotiation and decision, (even if it was to disagree) or at least have an opportunity to have their opinion heard.
- Flexibility - is the opportunity for compromise and is tightly related to participation. A negotiation may be extremely rigid, there is very little opportunity to make trade-offs to reach a compromise. Rigidity is often related to the ability to make trade-offs. In software development these often involve resource - time and people, versus features. This is an inelastic compromise (e.g. a staff year is not 730 people doing a job before lunch). Flexible Response
- "relative power" are both negotiators evenly matched or is one at a disadvantage - may be coerced into agreeing to terms that are not appropriate or fair to them. Is one party desiring to "force" their terms or the other (e.g. impose a schedule) is one party simply (and naively) agreeing to any terms presented to them. Closely related to the above, negotiation leverage - is customer internal or external, an external customer pays real money.
- Formal or Informal - A formal negotiation is a scheduled negotiation where the participants arrive prepared at the meeting with an agenda and the expectation some resolution shall be made at the end of the meeting. An informal negotiation is one where the participants meet on an ad hoc basis to negotiate an issue.
- Spirit - agreeableness to intransigence - this is controlled by bunkering state. (probably subsumes the "relative power state"

descriptive

- Scheduling or Design
- Negotiators By Proxy (or gate keepers) or principle (e.g. gate keep BH all changes go through me) or are developers approached directly
- role in the negotiation, direct participant or only supplying information to the negotiator
Related to

- **Uncertainty** - Uncertainty associated with the scope of the negotiation's Job - what is negotiated
- **Communication** - without it there is no negotiation
- **Reaching Out** - as part of the negotiation process because effective negotiations require people to "walk a mile" in another man's shoes.
- **Flexible Response** - there must be something to negotiate
- **Elaboration** - generates input to planning
- **Compromise**

The Negotiation Process

Single phase negotiations

The problem with single phase negotiations is they are conducted at the beginning of an effort and therefore made during maximum ignorance. Without a mechanism to reopen the negotiations (e.g. multi-phase) in the light of new information the provider is often placed in a difficult position of living up to their commitment without the resources needed to perform the job.

Negotiation

Negotiation may be as simple as a single phase affair, where the negotiators are the principle decision makers requester: "this is what I want" provider: "ok"

or it may be a long drawn out multi phase affair with a repeating cycle of Negotiating Consensus Elaboration Feedback - Visibility. A common cycle was a phase negotiation where a requester stated their desire in broad terms, the provider would then elaborate the terms slightly to get a better idea of what the features were and the resource requirements. If the requester liked what they saw they approved, otherwise the cycle may repeat again.

Negotiation is not always about Balancing competing demands by Reaching Out and finding Compromise, it frequently is a search for a solution. A requester has a problem that must be solved, and a solution is required to solve it.

Uncertainty has a strong influence on negotiation. Providers prefer not to commit to a job where they cannot reasonably predict effort. Providers do not want release a job when they are not sure what they are getting.

For experienced and Knowledge individuals (both requesters and providers) a single phase negotiation may be all that is required. There is minimal risk and uncertainty, therefore the need for bunkering is minimal. When an individual is Exposed, young inexperienced providers may simply take what is given them on the terms given to them by requester.

there is usually significant Tension between the Perspective of the requester and the supplier, and these must be negotiated.

Is an important strategy for coping with resource constraints and the inherent Tension that results from different perspectives. Negotiation represents an attempt to honestly bring the work load into line with the available resources.

Negotiations are strengthened by Knowledge however most negotiations are often conducted in uncertainty, hence the multi-phase negotiation and the need to maintaining good communications and avoid surprise.
Software development is almost one ongoing negotiation, negotiating features, negotiating resources, negotiating schedules, negotiating interfaces.

There is no negotiation possible when there is no compromise. This is where several of our subjects told of prior Hitting the Wall stories where rigid expectations lead to disaster Flexible Response Deadline

- **Bunkering** escalating messy negotiations to manager or delegate (e.g., a gatekeeper) Team member escalates negotiation seeking shelter from manager. It is interesting to note that this form of sheltering had to be constantly and actively reinforced by the manager. Is there a relationship to Leadership here?
- **Communication** or Cut off is everyone talking to each other all the time (e.g., EP site 2 subject 1 "everything is on the walls" so there is no surprise It seems surprise causes a hardening of positions. Many examples of this, like when an apparently inconsequential report was removed at site 1, it causes a major event.

An important cycle that develops is Resolving Mismatches where negotiation is a critical part. Good Feedback - Visibility minimizes Surprise and quickly communicates mismatches which are then negotiated. Long delays in the Feedback - Visibility cycle that may result from Cut off can lead to big surprises which then alter the relationship to one can catch the heat. The Cut off cycle may be self-sustaining because the more heat the greater the bunker

**Discussion**

Software development is negotiation, a constant negotiation of features and resources. We saw several practices that fall under the name of negotiations

An effective Leadership is able to shelter their team, otherwise the team becomes exposed

**type of negotiation**

One type of negotiation is time, resources versus features. How much time and resources will stakeholders grant the development team to produce a feature set. There is a non linear trade-off between resources, time, and features (**references for this**

Even common sense tells us this is the case, Fred Brooks humorously put this as 9 women cannot make a baby in one month, or a staff year is not 350 people trying to get the job done before lunch.

Uncertainty in the project makes it difficult for anyone to have a firm position to stand on so much has to be placed in the sheer will to stand firm and hold a position - it is not in the interest of the development team to take on more work than they can promise to deliver (see examples of low quality (site 1 subject 1). Many organizations shelter the development team or those who may cave into pressure to accept more work by having one person or a select group engage in the negotiations.

Multi-stage negotiation - fuzzy estimate initially made, based on initial scope to get a ball park figure, teams tend to.

There was a lot of multi-phase negotiation where an initial fuzzy offer is made - as long as it is in the right direction, and the work is authorized to proceed and refine the precision and accuracy of the estimates - this is where design comes in. (see site 1 subject 1)

Negotiations will proceed fairly well under plotting, good communications, willingness to be flexible and compromise.

Software development seems to be one big ongoing negotiation.

Managing uncertainty in a resource constrained environment results in much negotiation. Each group has a perspective, a list of what they believe is relevant to the project, features, technical debt, etc. Resources are constrained, the objective is treaty making.

Treaties are negotiated
Treaties can be broken, there are consequences when treaties are broken. 
Treaties may be imposed - forcibly driving the agenda
Treaties are enforced and monitored...
relate treaties and bunkers
Treaties and flexibility
Bunkers to limit uncertainty, treaties are negotiated between bunker.

What can often throw off the delicate balance of the treaty is different perspectives and the friction resulting from conflicting perspectives. The technical community prizes perceived efficiency (emphasis on the word perceived).
look how many agilists believe the software development team is more productive when "isolated" or disconnected from the the corporate eco systems. This is something important to put in the discussion how the software development team tends to isolate itself from the corporate eco-system, characterized by my observations of policies advocated by Scrum (codified in methods like Scrum). This is where the bunkering term comes from is the more tech people tend to bunker. The closer to the technology a person was, the more they tend to bunker.

-----------------------

Negotiation seems to be used in several ways
1) non trade-off, as a feature discovery tool
2) trade-off - negotiate features and effort - team is asked for estimates. Team comes under pressure resorts more to bunkering when in a position where they believe they cannot deliver what is being asked of them.
3) non negotiation - develop a backlog and commit only to short increments - micro negotiation as a way of coping with uncertainty in scope and effort.

Children (17)
- Indicators Negotiating Consensus Site 3 Subject 2
- Indicators Negotiating Consensus Site 3 Subject 5
- Indicators Negotiating Consensus Site 3 Subject 6
- Indicators Negotiating Site 2 Subject 3 Follow-up
- Indicators Negotiating Site 3 Subject 4
- Indicators Negotiation Site 3 Subject 7
- Elaboration
- Indicators Negotiation Site 3 Subject 3
- Flexible Response
- Deadline
- Compromise
- Expectation
- Indicators Negotiation Site 1 Subject 2
- Indicators Negotiation Site 1 Subject 1
- Indicators Negotiation Site 2 Subject 1
- Indicators Negotiation Site 3 Subject 3
- Indicators Negotiation Site 3 Subject 1

Comments (5)

Steve Adelphi says:

Dec 14, 2009
it is interesting that I did not classify any of site 1 subject 3 story as negotiation. I wonder if this has anything to do with him being a new hire in the organization and still not sure of his place within the organization. Most of his concerns with unpredictability were technical scoping and not estimating. More senior developers are typically responsible for negotiating or re-prioritizing the backlog.

Steve Adeolph says:

Jan 06, 2010

Negotiation is done in the context of imperfect information, stakeholders have a fuzzy idea of what they want and developers have an even fuzzier idea of what they can deliver. Yet there is still an iron triangle of time, resources, and quality. Estimates are usually very optimistic and requirements are never stable change tends to blow the estimates out.

A lot of the estimation strategy involves standing firm. Or relying on one person to shelter the team (essence of bunkering)

Steve Adeolph says:

May 13, 2011

Negotiating Consensus is at the core of the agile philosophy, consider planning poker - we observed several planning poker sessions - the intent is to get all ideas on the table to "broaden the perspectives" and come to agreement for the size of a feature. Subverting this process, but limiting its time, or by being "inflexible" willing to consider alternatives leads to failure.

Steve Adeolph says:

May 30, 2011

All the teams we observed and interviewed were Scrum teams and prioritization of the backlog was a constant negotiation.

Steve Adeolph says:

Jun 13, 2011

Negotiating really consumbed or subsumed Planning really when I ask the question "what is happening here" I see planning is a negotiation.
lack of thought leadership  I mean a lot of that is a lac—a lack of—thought leadership around that area around PS. Typically, it comes from the senior devs and there's only 1 or 2 per team. So they just don't have the bandwidth to try and think about the process improvement things.

without leadership stepping up EP6 would not have happened  And I put a lot of effort into making sure it went out. And the bugs—we got to any form of stability. I don't think—without that step up in leadership—and the other team lead—once he figure it out as well stepped up as well. And without that EP-6 wouldn't—I don't think I be here talking to you—not in this company anyway. I think it would have been—if I'd carried on—it just never would have been released. So it—stepping up—getting everyone to talk—getting everyone focused. I mean—the more

Comments (2)

Steve Adelph says:  Jun 30, 2011

lack of thought leadership here really is focused on the area of SPI, but I find it interesting how without the leadership stepping up as leaders things don't change. This is seriously a theme in reconciling perspectives that relates back to the idea of personal strength. We need people to step up and lead.

Steve Adelph says:  Jun 30, 2011

Good elements of leadership, not all part of reconciling perspectives, but essential to reconciling perspectives or any software process. This could have maybe been an alternate theory.
APPENDIX I  THE THEORY EMERGES FOR THE FIRST TIME
## APPENDIX J  GLOSSARY

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accepted Work Product</td>
<td>An Accepted Work Product is a Work Product that is accepted by the Acquirer during Evaluating which also satisfies the Supplier’s needs for technical excellence.</td>
</tr>
<tr>
<td>Acquirer</td>
<td>An Acquirer may be an individual or a representative of an organization that needs a Job done. While it is usual to think of Acquirers as representatives of the business side of an organization, individuals from the engineering side of the organization play the Acquirer role when they ask a colleague to provide a service they require.</td>
</tr>
<tr>
<td>Alerting</td>
<td>Alerting is a Leadership role that helps individuals stay connected to the Ecosystem and bringing suspected Perspective Mismatches to the attention of individuals and teams.</td>
</tr>
<tr>
<td>Broadening</td>
<td>Broadening is one of three strategies for Negotiating Consensus and is a strategy used when the Acquirer and Supplier work to understand a problem from the other’s point of view. This is sometimes classically referred to as “walking a mile in another man’s shoes”.</td>
</tr>
<tr>
<td>Bunkering</td>
<td>Bunkering is the third sub-stage of Reconciling Perspectives and is a quiet “heads down” stage where the Supplier seeks quiet focused time to Get the Job Done. This stage is entered when a Consensual Perspective is reached and the process transitions to the Validating stage.</td>
</tr>
<tr>
<td>Communication</td>
<td>Communication is the exchange of information between two or more individuals; it connects individuals to the organizational Ecosystem.</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Consensual Perspective</td>
<td>A Consensual Perspective is the result of Negotiating Consensus when the Acquirer and Supplier consensually agree that they are “on the same page” and have Converged their Perspective Mismatch. The Consensual Perspective results when all participants in the process believe they have resolved the Perspective Mismatch because they are not hearing anything in their Communications that would suggest otherwise.</td>
</tr>
<tr>
<td>Convergence Stalling</td>
<td>Convergence Stalling is the failure path through Reconciling Perspectives explaining what happens when the negotiation never reaches an explicit Consensual Perspective and both Supplier and Acquire continue to work with a Perspective Mismatch.</td>
</tr>
<tr>
<td>Converging</td>
<td>Converging is the first stage of Reconciling Perspectives, where individuals recognize that a Perspective Mismatch is impeding Getting the Job Done and Reach Out to engage others in Negotiating Consensus to converge their Perspective Mismatch into a Consensual Perspective. Converging has two sub-stages: Reaching Out and Negotiating Consensus.</td>
</tr>
<tr>
<td>Cut-Off</td>
<td>Cut-Off is an extreme end point of the Social Dynamic property of Openness where an individual or team filters out or blocks all Communications or is unwilling to compromise during Negotiating Consensus.</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>Cycle Time is an observable property of Reconciling Perspectives and is the interval of time from when an individual first Reaches Out until the Job Work Products are Evaluated.</td>
</tr>
<tr>
<td>Diversity</td>
<td>Diversity is a property of Communications and is the variation in the sources of Communications.</td>
</tr>
<tr>
<td>Drum Beating</td>
<td>Drum Beating is a Leadership role that moves the process forward and avoids Convergence Stalling. Drum Beating creates a rhythm that keeps the process from stalling, or restarts a stalled process.</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Ecosystem</td>
<td>The <em>Ecosystem</em> is the organizational context in which software development occurs. The term <em>Ecosystem</em> attempts to capture the diverse and chaotic mix of software methods, corporate policies, competing interests, personal agendas, personality types, and all variety of formal and informal relationships, task forces, alliances, and in-groups that must all interact to create software.</td>
</tr>
<tr>
<td>Evaluating</td>
<td><em>Evaluating</em> is the fourth and final sub-stage of <em>Reconciling Perspectives</em>, and the second stage of <em>Validating</em>. <em>Evaluating</em> starts when an <em>Acquirer</em> presents <em>Work Products</em> for a <em>Job</em> to the <em>Acquirer</em>. <em>Evaluating</em> is entered from <em>Bunkering</em> when:</td>
</tr>
<tr>
<td></td>
<td>1. A pre-agreed time for delivery of the <em>Work Products</em> is reached (e.g. a milestone)</td>
</tr>
<tr>
<td></td>
<td>2. The <em>Supplier</em> presents the <em>Work Products</em> to the <em>Acquirer</em></td>
</tr>
<tr>
<td></td>
<td>3. The <em>Acquirer</em> demands the <em>Work Products</em> from the <em>Supplier</em></td>
</tr>
<tr>
<td></td>
<td>The outcome is either an <em>Accepted Work Product</em> which validates the <em>Consensual Perspective</em>, or <em>Waste</em>. Significant <em>Waste</em> may also result in <em>Surprise</em>.</td>
</tr>
<tr>
<td>Exposed</td>
<td><em>Exposed</em> is an extreme end point of the <em>Social Dynamics</em> property of <em>Openness</em> which occurs when individual or team does not filter or restrict <em>Communications</em> and engages in all <em>Communications</em>.</td>
</tr>
<tr>
<td>Failure to Reach Out</td>
<td><em>Failure to Reach Out</em> is the failure path through <em>Reconciling Perspectives</em> explaining what happens when an individuals or organizations noticing a <em>Perspective Mismatch</em> never reach out, or other parties decline to engage in the process.</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Getting the Job Done    | *Getting the Job Done* is the central concern of everyone involved with the process of software development (both Acquirers and Suppliers); it is delivering the best Work Products they knew how create. Impediments to *Getting the Job Done* are a major source of frustration.  
*Getting the Job Done* is creating working software that:  
* Appeases the customer or even makes the customer happy,  
* Satisfies the software team members’ needs to see the end and achieve a feeling of accomplishment, and  
* Satisfies the team members’ desires to minimize technical debt. |
<p>| Heat Calculation        | <em>Heat Calculation</em> is analogous to calculating expected value. The <em>Heat Calculation</em> is a subjective calculation made to select an alternative from a set of undesirable alternatives where an individual or team will “catch the least heat”. |
| Invalidation            | <em>Invalidation</em> occurs during Bunkering when new Communications challenge the Consensual Perspective. The process returns from the Validating stage back to the Converging state. |
| Isolation               | <em>Isolation</em> is the failure path through Reconciling Perspectives explaining what happens when during Bunkering, the Supplier Cuts-Off all Communications and ignores Communications that may Invalidate the Consensual Perspective or indicate further Perspective Mismatches. |
| Iterative Flow          | <em>Iterative Flow</em> is a success path through Reconciling Perspectives that results in an Accepted Work Product. The <em>Iterative Flow</em> is a variant of the Simple Linear Flow and occurs when the Consensual Perspective is Invalidated. An <em>Iterative Flow</em> can degenerate into Convergence Stalling if a new Consensual Perspective is not reached in a timely manner. |
| Job                     | A <em>Job</em> is a unit of work that an Acquirer wants done. A Supplier creates Work Products to satisfy the Acquirer’s expectations for the Job. |</p>
<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Leadership</strong></td>
<td><em>Leadership</em> is the ability of individuals to establish a culture that helps individuals and teams manage <em>Communications</em>, to remain connected to the <em>Ecosystem</em> without becoming <em>Exposed</em>, and move the process forward.</td>
</tr>
<tr>
<td><strong>Negotiating Consensus</strong></td>
<td><em>Negotiating Consensus</em> is the second sub-stage of <em>Converging</em> and is a dialog between <em>Acquirers</em> and <em>Suppliers</em> intended to converge their <em>Perspective Mismatch</em> and create a <em>Consensual Perspective</em>.</td>
</tr>
<tr>
<td><strong>Openness</strong></td>
<td><em>Openness</em> is the <em>Social Dynamic</em> property that characterizes how permeable to <em>Communications</em>, and how open to compromise, individuals and teams are, and it ranges from <em>Cut-Off</em> to <em>Exposed</em>.</td>
</tr>
<tr>
<td><strong>Personal Strength</strong></td>
<td><em>Personal Strength</em> is a personal trait that is elusive to define beyond characterizing it as an individual’s ability to reach out and engage others when they suspect something is amiss. It is also the ability to avoid becoming <em>Exposed</em> and knowing when to say “no”.</td>
</tr>
<tr>
<td><strong>Perspective</strong></td>
<td>A <em>Perspective</em> is an individual’s point of view, which they use to interpret the <em>Job</em> and establish what they believe “done” means. A <em>Perspective</em> establishes expectations for what <em>Work Products</em> should be delivered for a <em>Job</em> and informs the <em>Job’s</em> trade-off priorities.</td>
</tr>
<tr>
<td><strong>Perspective Mismatch</strong></td>
<td><em>Perspective Mismatch</em> is an impediment to <em>Getting the Job Done</em> that results from different points of view or <em>Perspectives</em> of those involved in the process of software development. The mismatch may result from different <em>Perspectives</em> regarding <em>Job</em> features, priorities, development methods, and/or status.</td>
</tr>
<tr>
<td><strong>Reaching Out</strong></td>
<td><em>Reaching Out</em> is the first sub-stage of <em>Converging</em>. <em>Reconciling Perspectives</em> begins when individuals reflect on the <em>Communications</em> they are receiving about the <em>Job</em> and believe <em>Getting the Job Done</em> is being impeded by a <em>Perspective Mismatch</em>, and therefore begin <em>Reaching Out</em>. They <em>Reach Out</em> by inviting others to engage with them to resolve the mismatch.</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Reconciling Perspectives</td>
<td>Reconciling Perspectives is a basic social structural process that people managing the process of software development use to resolve the impediment to Getting the Job Done created by a Perspective Mismatch. Acquirers and Suppliers reconcile a Perspective Mismatch by Converging the Perspective Mismatch into a Consensual Perspective which is Validated by the construction of and acceptance of Work Products. Waste results if the Work Products are not accepted.</td>
</tr>
<tr>
<td>Recursive Flow</td>
<td>A Recursive Flow is a success path through Reconciling Perspectives that results in an Accepted Work Product. A Recursive Flow occurs when the reconciliation of a Perspective Mismatch raises another related Perspective Mismatch that must be resolved before the original mismatch can also be resolved.</td>
</tr>
<tr>
<td>Saturation</td>
<td>Saturation is the failure path through Reconciling Perspectives explaining what happens when, during Bunkering, the Supplier does not filter any Communications, becoming distracted and unable to focus on creating the Work Products.</td>
</tr>
<tr>
<td>Scope</td>
<td>Scope is a property of Reconciling Perspectives that characterizes the size of the process in terms of the number of individuals the outcome of the process may influence. Reconciling Perspectives may occur between a pair of individuals reconciling a Perspective Mismatch that is only of concern to themselves, or they may be reconciling a Perspective Mismatch that will influence their respective organizations.</td>
</tr>
<tr>
<td>Scouting</td>
<td>Scouting is one of three strategies for Negotiating Consensus and is a strategy used when it is agreed that the gap between the Perspectives cannot be reconciled without more information.</td>
</tr>
<tr>
<td>Scouting Blowout</td>
<td>Scouting Blowout is Scouting that is continued without end and results in Convergence Stalling.</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Sheltering</strong></td>
<td><em>Sheltering</em> is the action of keeping individuals connected to the <em>Ecosystem</em> while protecting them from becoming <em>Exposed</em> (sometimes called buffering, or shielding).</td>
</tr>
<tr>
<td><strong>Simple Linear Flow</strong></td>
<td>A <em>Simple Linear Flow</em> is a success path through <em>Reconciling Perspectives</em> that results in an <em>Accepted Work Product</em>. The <em>Simple Linear Flow</em> is a stage by stage progression through Reconciling Perspectives that reconciles a <em>Perspective Mismatch</em> and results in a set of <em>Accepted Work Products</em>.</td>
</tr>
<tr>
<td><strong>Social Dynamics</strong></td>
<td><em>Social Dynamics</em> is a property of <em>Reconciling Perspectives</em> and moderates <em>Reconciling Perspectives</em>. <em>Social Dynamics</em> is the capability of a team to manage <em>Communications</em> and is characterized by four properties:</td>
</tr>
</tbody>
</table>
|                  | • *Leadership*  
|                  | • *Openness*     
|                  | • *Personal Strength* 
<p>|                  | • <em>Experience</em>   |
| <strong>Supplier</strong>     | A <em>Supplier</em> is an individual, team, or organization that has the ability to create <em>Work Products</em> that may satisfy the needs of an <em>Acquirer</em> as expressed by a <em>Job</em>. |
| <strong>Supporting</strong>   | <em>Supporting</em> is a <em>Leadership</em> role that creates an environment that encourages <em>Reaching Out</em>. |
| <strong>Surprise</strong>     | <em>Surprise</em> is a reaction to dramatic, unexpected <em>Waste</em>. |
| <strong>Translation</strong>  | <em>Translation</em> is one of three strategies for <em>Negotiating Consensus</em>. It is used when a <em>Perspective Mismatch</em> is apparently resulting from the use of different terminology by the <em>Acquirer</em> and <em>Supplier</em>. |
| <strong>Volume</strong>       | <em>Volume</em> is a property of <em>Communications</em> and is the frequency and the quantity of relevant information exchanged. |</p>
<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waste</td>
<td><em>Waste</em> is loss of value that results when the delivered <em>Work Products</em> do not meet expectations. From the <em>Acquirer’s</em> point of view, <em>Waste</em> is:</td>
</tr>
<tr>
<td></td>
<td><em>Missing or deficient features,</em></td>
</tr>
<tr>
<td></td>
<td><em>Unnecessary or over engineered features,</em></td>
</tr>
<tr>
<td></td>
<td><em>Opportunity cost and lost revenue resulting from late delivery work products.</em></td>
</tr>
<tr>
<td></td>
<td>From the <em>Supplier’s</em> point of view, <em>Waste</em> is either the accumulation of technical debt they believe will impede future growth of the system, the loss of satisfaction with the system, or the development of features that are not required.</td>
</tr>
<tr>
<td>Work Products</td>
<td><em>Work Products</em> are the deliverables created by the <em>Supplier</em> to satisfy the <em>Acquirer’s</em> expectations for the <em>Job.</em></td>
</tr>
</tbody>
</table>