6-27-2014

Traveling of Requirements in the Development of Packaged Software: An Investigation of Work Design and Uncertainty

Thomas Gregory

DISSERTATION

Traveling of Requirements in the Development of Packaged Software

An Investigation of Work Design and Uncertainty

Thomas A. Gregory

CENTER FOR PROCESS INNOVATION
J. MACK ROBINSON COLLEGE OF BUSINESS
GEORGIA STATE UNIVERSITY

27 June 2014

PERMISSION TO BORROW

Thomas A. Gregory

NOTICE TO BORROWERS

The author of this dissertation is:

Thomas A. Gregory

Center for Process Innovation
J. Mack Robinson College of Business
Georgia State University
35 Broad Street, NW, Suite 400
Atlanta, GA 30303

E-mail: tom@alt-tag.com
Homepage: http://alt-tag.com

The director of this dissertation is:

Dr. Lars Mathiassen
GRA Eminent Scholar
Professor, Computer Information Systems Academic Director, Executive Doctorate in Business Center for Process Innovation

J. Mack Robinson College of Business
Georgia State University
35 Broad Street, NW, Suite 427
Atlanta GA 30303

E-mail: lmathiassen@ceprin.org

Phone: +1-404-413-7855

Homepage: http://www.larsmathiassen.org

TRAVELING OF REQUIREMENTS IN THE DEVELOPMENT OF PACKAGED SOFTWARE: AN INVESTIGATION OF WORK DESIGN AND UNCERTAINTY

BY

Thomas A. Gregory

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree

Of

Doctor of Philosophy

In the Robinson College of Business

Of

Georgia State University

GEORGIA STATE UNIVERSITY

ROBINSON COLLEGE OF BUSINESS

2014

ACCEPTANCE

This dissertation was prepared under the direction of Thomas A. Gregory's Dissertation Committee. It has been approved and accepted by all members of that committee, and it has been accepted in partial fulfillment of the requirements for the degree of Doctoral of Philosophy in Business Administration in the J. Mack Robinson College of Business of Georgia State University.

H. Fenwick Huss, Dean

DISSERTATION COMMITTEE:

Dr. Lars Mathiassen (Chair)

Dr. Richard Baskerville

Dr. Balasubramaniam Ramesh

Dr. Sandra Slaughter

ABSTRACT

TRAVELING OF REQUIREMENTS IN
THE DEVELOPMENT OF PACKAGED SOFTWARE:
AN INVESTIGATION OF WORK DESIGN AND UNCERTAINTY

BY

THOMAS A. GREGORY

27 June 2014

Committee Chair: Dr. Lars Mathiassen

Major Academic Unit: Center for Process Innovation

Software requirements, and how they are constructed, shared and translated across software organizations, express uncertainties that software developers need to address through appropriate structuring of the process and the organization at large. To gain new insights into this important phenomenon, we rely on theory of work design and the travelling metaphor to undertake an in-depth qualitative inquiry into recurrent development of packaged software for the utility industry. Using the particular context of software provider GridCo, we examine how requirements are constructed, shared, and translated as they travel across vertical and horizontal boundaries. In revealing insights into these practices, we contribute to theory by conceptualizing how requirements travel, not just locally, but across organizations and time, thereby uncovering new knowledge about the responses to requirement uncertainty in development of packaged software. We also contribute to theory by providing narrative accounts of *in situ* requirements processes and by revealing practical consequences of organization structure on managing uncertainty.

ACKNOWLEDGEMENTS

The road, as it so often is, was longer than I first anticipated. I have been helped by so many mentors along the way that any attempt to enumerate them will inevitably be incomplete. Each are luminaries in their own right, and this simple acknowledgement is inadequate to express how blessed I am to have been able to interact with such great people.

To my advisor, Dr. Lars Mathiassen, I am forever grateful. His patience rivaled that of a saint as he uncomplainingly (to me at least!) endured my frustration and procrastination, and gently teased out a more coherent set of ideas and organization than I would have ever been able to accomplish on my own.

I am grateful to Lars, as well as Arun Rai and Richard Welke for not only being excellent mentors, but also coordinating the Center for Process Innovation and encouraging research that is engaged with business. The field is better for it.

As I marched—and sometimes plodded—down the path of the PhD program, many have provided assistance and advice along the way. I'm lucky to have learned from Detmar Straub, Dan Robey, Lisa Lambert, and many, many others here at Georgia State University. I am thankful for each of you. He may not know it, but this journey was, in part, inspired by advice from Dr. Dave Jennings and others at Brigham Young University.

Any list of acknowledgements would be woefully incomplete without recognition of the time and effort the dissertation committee put in to reading, advising, and challenging this document. You each deserve a very special thank you: Lars Mathiassen (Chair), Richard Baskerville, Bala Ramesh, Sandy Slaughter.

Lastly, to my family, I must say thank you for enduring late nights, long weekends (and long breaks) and the many sacrifices that come with a doctoral program.

I am grateful for you all.

TABLE OF CONTENTS

TABLES AND FIGURES

ABBREVIATIONS

List of Abbreviations (in Alphabetical Order)

C&C	Command and control system, specifically the C&C software product developed by GridCo
CBS	COTS-Based System
CEPRIN	Center for Process Innovation (at Georgia State University)
COTS	Commercial Off-The-Shelf (Software)
CRM	Customer relationship management, a type of large software package
EJIS	*European Journal of Information Systems*
ERP	Enterprise resource planning, a type of large software package
FW	Firmware; low-level, embedded software
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers, a professional association
IRB	Institutional Review Board
IS	Information Systems
ISD	Information Systems Development
IT	Information Technology
MoU	Memorandum of Understanding
NPD	New Product Development
PMBOK	Project Management Body of Knowledge
SE	Software engineering
SW	Software; in the case of GridCo, generally a reference to the C&C SW
SWEBOK	Software Engineering Body of Knowledge (an IEEE standard)
TCE	Transaction Cost Economics

ABSTRACT

Software requirements, and how they are constructed, shared and translated across software organizations, express uncertainties that software developers need to address through appropriate structuring of the process and the organization at large. To gain new insights into this important phenomenon, we rely on theory of work design and the travelling metaphor to undertake an in-depth qualitative inquiry into recurrent development of packaged software for the utility industry. Using the particular context of software provider GridCo, we examine how requirements are constructed, shared, and translated as they travel across vertical and horizontal boundaries. In revealing insights into these practices, we contribute to theory by conceptualizing how requirements travel, not just locally, but across organizations and time, thereby uncovering new knowledge about the responses to requirement uncertainty in development of packaged software. We also contribute to theory by providing narrative accounts of *in situ* requirements processes and by revealing practical consequences of organization structure on managing uncertainty.

1 INTRODUCTION

1.1 Research Domain

Software is inherently complex (Brooks 1987), making its development a highly risky (Boehm 1991) and uncertain (Mathiassen and Pedersen 2008) activity. Yet, the outcomes of software development, as with development of any other product (Henderson and Clark 1990), are affected in multiple ways by the organizational context in which it is developed. We set out with the assumption that software requirements, and how they are constructed, shared and translated as they travel across the software organization, are expressions of uncertainties that software developers face and need to address through appropriate structuring of the process and the organization at large. Requirements are necessarily interpreted and negotiated as they travel through an organization on a journey intended to resolve the gap of uncertainty between customer needs and market options, on the one hand, and released software on the other. Thus, we use the lenses of uncertainty and work design to investigate the management and organization of software development as a complex human activity from the perspective of software requirements.

Software requirements are strongly analogous to task uncertainty, and are useful focal points for uncovering specific uncertainties in the development of software. By considering types of uncertainty developers might encounter (identity, complexity, and volatility uncertainties) (Mathiassen et al. 2007), the consequences to implementation of requirements and the task uncertainties they represent may be more fully elucidated. Moreover, adopting the language of task uncertainty (Galbraith 1973) enables the simultaneous investigation of work design in software development. Sinha and Van de Ven (2005) argued for reopening the study of work design within and between organizations, and provided a brief review of contingency theory. Contingency theory suggests that organizations build structures and processes to adapt to tasks and contexts (Drazin and Van de Ven 1985), and the consequences of these structures and processes are expressed as tradeoffs between mutually desirable, but exclusive goals. Thus, products developed in and between organizations, including software, may reflect attributes of the processes used to create them. Simultaneously, the organization adapts itself to the products it creates. As requirements must necessarily travel across vertical and horizontal boundaries, the selection, negotiation and interpretation of requirements is likely to change depending on how the development activity is structured. Hence, examining how requirements travel (Czarniawska and Joerges 1996) may help tease apart and understand this duality between the software and the organizational structure under which it is developed.

This collection of theories, namely, traveling (Czarniawska and Joerges 1996), organization contingency (Galbraith 1973), and uncertainty (Mathiassen et al. 2007), emerge naturally from the focus on requirements as they interact with and are adapted to the needs of their situating organization. Examining requirements across a system of development aligns naturally with conventional IS perspectives. The notion that requirements evolve throughout their life is readily explained by traveling theory. A view of organizational structure is necessary to describe *where* requirements travel, and we rely on the seminal work of Galbraith (1973) as well as modern simplifications (Sinha and Van de Ven 2005). Lastly, we utilize uncertainty not only because it is a core underlying concept of information processing in contingency theory, but also because it explains *why* requirements travel.

Using a specific corporate context of recurrent development of packaged software, this study looks deeply into uncertainties encountered during development and addresses the problems organizations face in ensuring requirements are effectively constructed, shared, and translated as they travel across vertical and horizontal boundaries, not only within a particular release cycle, but also between connected releases. The context of packaged software allows for temporal effects as development is recurrent—the same product is iterated over multiple releases—and this may be reflected in the traveling behavior of requirements. In addition to richer insight in uncertainty in a software development context, this research extends the sparse packaged software literature by providing evidence to confirm or challenge the field's understanding of the nature and contextual effects on packaged software development.

1.2 Research Questions

In order to respond to these general themes of software requirements and work design, it is necessary to examine in detail how requirements behave in particular organizational contexts:

> ***RQ 1:*** *How are requirements constructed, shared, and translated in recurrent development of packaged software?*

Zooming in (Nicolini 2009) on the relationship between work design and requirements, we adopt the traveling metaphor (Czarniawska and Joerges 1996; Nielsen et al. 2013) in conjunction with the notion of boundaries (Carlile 2002) to examine how requirements change within and across boundaries in their journey towards software delivery. In an organizational context, such boundaries might be horizontal or vertical (Sinha and Van de Ven 2005). This leads to the second research question:

> ***RQ 2:*** *How do requirements travel across vertical and horizontal boundaries in recurrent development of packaged software?*

In addressing these questions, this dissertation seeks contribution to theory by uncovering new knowledge about the sources of and responses to requirement uncertainty in recurrent development of packaged software. In this way, it answers recent calls (Austin and Devin 2009) for inductive qualitative research of design of software processes based on contextual factors. Further, it contributes to theory by providing detailed accounts of an organization's contextual responses to managing requirements as they travel across boundaries, reaffirming the need for process reinforcement that supports the role of boundary spanners.

Despite the wealth of requirements research in the software engineering tradition, based on evidence from analyses (Hassan and Mathiassen Forthcoming) and reviews of requirements literature in information systems (Mathiassen et al. 2007), little is known in the information systems (IS) field. Especially, we lack knowledge about the human dynamics involved in constructing and translating requirements as multiple actors negotiate meaning and resolve uncertainty across organizational boundaries. Additionally, the consideration of software requirements in packaged software is novel within the scope of IS literature, as compared with traditional, in-house or outsourced software development.

This dissertation seeks to answer these research questions via an empirical qualitative study. We list the consequential contributions to knowledge in Table 1-1.

Table 1-1: Contributions to Knowledge

Target	Gap	Contribution
IS research	Sparse requirements literature in IS; requirements under-represented in literature	Exploration of requirements practices as they unfold in an organization
	Sparse application of traveling metaphor in IS literature	Reinforce utility of traveling metaphor to support process studies and theory building
Software development research	Software development should be examined in the context of its implementing organization	Investigation of connection between work design and uncertainty in software development
	Uncertainty avoidance and mitigation	Traveling metaphor reveals new insights into management of uncertainties in development practices
Packaged software research	Area is under-studied and only speculatively defined	A detailed empirical account of packaged software development informs beliefs
	Lack of grounded concepts about development of packaged software	Conceptualization of the different ways in which packaged software requirements travel and consequences for organizing the process

1.3 Summary of Dissertation

This dissertation presents relevant research and extant theory; describes the empirical setting, methodology, and analysis; and discusses findings and contributions to theory according to the following structure:

- **Chapter 2** considers uncertainty as it appears in different literature streams, and connects uncertainty to both the study of organizations and software development. Task uncertainty is explained, and differentiated from requirements uncertainty and environmental uncertainty.
- **Chapter 3** reviews the packaged software literature and makes the case for studying the domain of packaged software. The anticipated effects of packaged software on organizations and development processes are contrasted with development of other types of software.
- **Chapter 4** summarizes the requirements management literature both from software engineering and information systems perspectives and emphasizes the lack of interactions between the two streams. Requirements processes are described as iterative and parallel. Requirements are related to uncertainty.
- **Chapter 5** presents modern and seminal theory of work design and how it relates to uncertainty. Horizontal and vertical work design structures are explicated. The problems arising from particular work design structures are highlighted and framed as uncertainties. Organizations are similar to software in the sense that they are the result of design as well as emergence.
- **Chapter 6** presents and adapts the “travel of ideas” literature. The central concept of “traveling” is discussed and further dissected to provide greater clarity in a software development context.
- **Chapter 7** describes the setting and design for this research, and details data and method of collection. We use a qualitative, case study method. GridCo has a

structured new product development (NDP) method. Analysis is delineated, and the intended coding scheme is justified.

- **Chapter 8** describes how requirements at GridCo traveled in certain ways within and across release cycles as participants addressed uncertainties. We identified and analyzed three major categories of traveling behavior: local, cross-layer and cross-cycle.
- **Chapter 9** using GridCo in relating analyses to theories of traveling, work design, and recurrent packaged software. Strengths and weaknesses of GridCo's development practiced are presented as engaged scholarship. We summarize contributions and limitations.

2 UNCERTAINTY

Uncertainty is defined and discussed based on different literature streams. Uncertainty is connected to both the study of organizations and software development. Task uncertainty is explained, and differentiated from requirements uncertainty and environmental uncertainty.

2.1 Uncertainty in Information Systems and Organization Research

Uncertainty is no stranger to information systems research, and has many facets. Broadly speaking, uncertainty is the absence of complete information, and has been called by Thompson (1967) the primary issue facing senior managers (Nidumolu 1995). Uncertainty includes (to paraphrase former U.S. Secretary of Defense Donald Rumsfeld) both known-unknowns and unknown-unknowns (Pawson et al. 2011).

The concept of uncertainty appears in most streams of IS and organization research, as decisions must be made and work produced in the absence of complete information. Consequently, uncertainty appears in strategy (Jauch and Kraft 1986; Milliken 1987), Transaction Cost Economics (TCE) (Williamson 1991), software development (Nidumolu 1995), project management (Jiang et al. 2009), requirements management (Nidumolu 1996), and the study of work design (Galbraith 1973), just to name a few. The term "uncertain" is also applied to information, particularly in the fields of database and knowledge base systems, where uncertainty is used to mean "the representation of and query support for information that is fuzzy, unknown, partially known, vague, uncertain, probabilistic, indefinite, disjunctive, possible, maybe, incomplete, approximate, erroneous, or imprecise" (Dyreson 1997, p. 413).

As Downey and Slocum (1975, p. 562) point out in their review of what came to be known as "environmental uncertainty" in the strategy literature, "uncertainty is a term which is used daily in a variety of ways. This everyday acquaintance with uncertainty can be seductive in that it is all too easy to assume that one knows what he is talking about." Certainly, each research domain seems to have subtly different definitions of uncertainty that are not always reconcilable.

One of the most basic questions in comparing literature on uncertainty is determining whether it is an objective or perceptive state (Downey et al. 1975; Downey and Slocum 1975; Milliken 1987). The fields of information processing, decision sciences, and computer engineering refer to uncertainty as an objective property of information, when a result is between states, unknown or incomplete (Dyreson 1997). In contrast, most organizational and strategy research takes the psychological viewpoint that uncertainty is a state faced by some deciding actor, which may occur because of missing, incomplete, conflicting, transient, or complex information.

A common subcategory of uncertainty, particularly in the governance literature, is *ambiguity*, the absence of information in decision making or the unknowability of outcomes, including a lack of understanding of cause–effect relationships, unknown variables, or unknown alternatives (Carson et al. 2006; Milliken 1987). Imprecision of data (Morrissey 1990) similarly results in ambiguity. Ambiguity is a consequence of complex, dynamic, or emergent systems (Snowden and Boone 2007). This wild combination of too much and not enough information is attributed by some software development researchers to communication gaps, and are labeled *identity* concerns (Mathiassen et al. 2007).

Other classifications of uncertainty include *volatility*, the unpredictable rate of change in market demand or supply availability (Carson et al. 2006), as well as technological uncertainty, which indicates future changes in technology are unknown and could led to significantly different future costs for a considered technology or render it obsolete (Choudhury 1997). Strategy researchers use a similar construct, dynamism, or the rate and unpredictability of environmental change (Miller and Friesen 1983; Newkirk and Lederer 2006).[1] Within software development research, volatility concerns may arise because of changing market pressures, customer preferences, or business needs, to list a few of many reasons, and may manifest as budget or schedule changes.

As in TCE, a basic assumption behind uncertainty is bounded rationality, the finite ability of humans to access, store, and process information (Simon 1979). Although generally unspoken, the assumption of bounded rationality undergirds the information processing perspective (e.g., Galbraith 1973) typical of research in organization

1 Strategy research considers the broader label, environmental uncertainty, with three dimensions: dynamism, heterogeneity (complexity of factors in the environment), and hostility (degree of external competition) (Miller and Friesen 1983), although a review of resource-based view (RBV) literature applies munificence (the extent to which a business can grow) instead of hostility (Wade and Hulland, 2004). Although now collectively labeled environmental uncertainty, under the original conception only dynamism was considered uncertainty (Miller and Friesen 1983), and was defined similarly to the constructs volatility and demand uncertainty in other streams as mentioned above. However, because of bounded rationality, heterogeneity and hostility increase the likelihood of ambiguity, so collectively considering the three dimensions as uncertainty is appropriate. This is particularly true, as these three dimensions have been shown to interact when considering the extent to which managers make erratic strategic decisions.

Mitchell, R.J., Shepherd, D.A., and Sharfman, M.P. 2011. "Erratic strategic decisions: when and why

Wade, M., and Hulland, J. 2004. “The Resource-Based View and Information Systems Research: Review, extension, and suggestions for future research,” *MIS Quarterly* (28:1), pp. 107-142.

structure and work design. Because of bounded rationality, uncertainties resulting from ambiguity are magnified when decision makers are also faced with complexity; the two facets of uncertainty are deeply intertwined. Thus, software development, a highly complex endeavor, is guaranteed to encounter uncertainty.

2.2 Uncertainty in Software Development

Uncertainty pervades software development (Brooks 1987) and its attendant processes, such as project management and requirements management. Zmud (1980) described the effects of uncertainty in software development, saying, “most difficulties can be traced to the uncertainty that pervades software development.”

One facet of the uncertainty found in software development has been called *requirements uncertainty*, meaning “the difference in the information necessary to identify user requirements and the amount of information possessed by the developers” (Nidumolu 1995, p. 136). Although this definition is useful, it too narrowly considers only the relationships between users, requirements, and developers. An earlier and more general concept, task uncertainty, captures the essence of requirements uncertainty.

Galbraith (1973, p. 5), defines *task uncertainty* as “the difference between the amount of information required to perform the task and the amount of information already possessed by the organization.” Software development is the act of creating information (as represented by, for example, source code), and is, by its nature, uncertain, and so a simpler but perhaps less precise definition of task uncertainty in a software development context is the difference between what has been done and what has yet to be done in response to specific customer needs and market demands. This difference is expressed in agreed upon requirements and the completion status of the next software release.

Galbraith (1973) argues for a correlation between the amount of information to be processed and the level of task uncertainty. By this definition, the potential for uncertainty in software development grows with the size of the software, as the amount of information needing processing—for example, the size of the code base—increases over time, and the possibility of unintended interactions between modules increases. Developers attempt to manage growing complexity over time with the use of software patterns and modularization, but it is the nature of software—as with other endeavors—to become more complex, and thus more uncertain, over time. The same argument applies to a growing organization faced with an increasing number of strategic customers whose demands must be satisfied: as their number increases, the potential for unintended negative interactions likewise increases, as does uncertainty.

Conversely, the potential for uncertainty decreases as the amount of information already possessed by the organization increases. Although over time, uncertainty is expected to go both up and down simultaneously, the resulting dynamics suggests a net increase in uncertainty over time. Knowledge as represented by a growing code base as well as the increasing institutional knowledge of developers over time tends to reduce overall uncertainty, but this benefit is mitigated by the increased complexity and increased likelihood of ambiguity.

Task uncertainty is also a function of the way the organization is structured.[2] The design of an organization itself leads to additional uncertainties (Sinha and Van de Ven 2005), and thus as the organization grows uncertainty will tend to increase (as evidenced by coordination cost (e.g., Kraut and Streeter 1995)), although this can be mitigated with strategies such as vertical information systems. Galbraith (1973) advocates hierarchical structures, in part because coordination costs are limited to logarithmic rather than exponential growth. However, any mitigation of uncertainty is limited by the cognitive ability of participants (Galbraith 1973).

As previously noted, a major distinction between requirements uncertainty and task uncertainty (on which it is based), is its narrow focus. Requirements uncertainty considers only information available from users, whereas task uncertainty encompasses all information needed to complete a task, including the tools used in the development process or understanding of appropriate software patterns or previous solutions which might be applicable to the problem being considered. Further information not encompassed by requirements uncertainty but within the umbrella of task uncertainty might include the extant state of the code being modified, and any feedback regarding iterative development steps, including such trivialities as syntax errors or more substantive feedback such as failed unit tests. In software development, some of this information (e.g., test feedback) is not available until a solution has been attempted. Galbraith's (1973) broader definition of task uncertainty, which references the

[2] Here the duality between software and organization is again manifest. Uncertainty is a function of work design while at the same time also conditions and informs work design. The lower the task uncertainty, the more structure one can use in designing work (high programmability); the higher the task uncertainty, the more organic structures would need to be applied (low programmability).

information required to perform a task, must thus necessarily include information that reflects whether the task was completed. In the context of software development, this suggests the outcome of development effort is uncertain until it is completed, a view consistent with the industry understanding of uncertainty in software development (Brooks 1987).

2.3 Uncertainty in Software Requirements

Researchers have adopted numerous categorizations for uncertainty in requirements and software development, including Mathiassen et al. (2007), who separate uncertainty into:[3]

1. *Identity*, the knowing of requirements caused by communications gaps[4];
2. *Volatility*, the changing of requirements whether for internal or external reasons, such changes in market and customer preferences, budget or priority changes, or timing and schedule changes; and
3. *Complexity*, the difficulty in specifying and communicating requirements, as well as the cognitive load required to understand the effects of implementation due to, for example, dynamic systems, lack of modularity, or quantity of constituent components or connections.

[3] More precisely, Mathiassen et al. (2007) describe these as risks rather than dimensions of uncertainty. This distinction will be addressed later (Section 2.4).

[4] In requirements management literature, communications gaps are almost always in reference to gaps in communication with customers. However, as these breakdowns can occur anywhere along the potentially numerous organizational and process boundaries that separate users from the developers writing source code, it is more useful to simply refer to these as communications gaps.

The approach of Mathiassen et al. (2007) both aligns and diverges from earlier work by Nidumolu (1996), who identifies a flavor of uncertainty mentioned previously, *requirements uncertainty*, in reference to the uncertainties encountered during management of software requirements. Requirements uncertainty stems from the information processing viewpoint of Galbraith (1973), and is defined as the difference between the information possessed by developers and the information necessary to determine end-user requirements (Liu et al. 2011; Nidumolu 1996). Other researchers have temporally bounded the idea of requirements uncertainty as occurring in only the planning or analysis phases of the IS development process, with consequences felt in design, implementation, and maintenance phases (e.g., Benslimane et al. 2010), although it is not clear whether this interpretation is commonly held. An increase in requirements uncertainty has been shown to have a positive relationship with inter-personal conflict among stakeholders (Liu et al. 2011). Both requirements uncertainty and interpersonal conflict are primary factors for the all-to-common failures in software development (Liu et al. 2011; McFarlan 1981; Robey et al. 1993).

Nidumolu (1996, p. 136) described three dimensions of requirements uncertainty:

1. “Requirements diversity, the extent to which users differ among themselves in their requirements”;
2. “Requirements instability, the extent of changes in user requirements”;
3. “Requirements analyzability, the extent to which the process for converting user needs to a set of requirements specifications can be reduced to mechanical steps or objective procedures.”

These dimensions match closely with the conclusions of Mathiassen et al. (2007). Requirements diversity is a subset of potential identity issues, requirements instability maps directly to volatility, and analyzability is a reasonable proxy for complexity. The primary difference between these descriptions is that Mathiassen et al. (2007) seem to be taking a broader view of the development process.

Software requirements are strongly analogous to, and representations of, task uncertainty. In an information-heavy context like software development, requirements document goals to be achieved, and thus represent the difference between what needs to be done and what has been done. Requirements are statements intended to define these gaps, and are thus attempts at setting boundaries around task uncertainty so that it might be managed during the development process. This dissertation consequently treats requirements as expressions of task uncertainty, with the expectation that the organization reveals additional uncertainty (identity, complexity, volatility) as it attempts to resolve acknowledged task uncertainty. In doing so, this research uses requirements as a point of entry to examine simultaneously the theoretical consequences of uncertainty in contingent organization design.

2.4 Uncertainty and Risk

Uncertainty and risk have a close relationship. Partly due to the differing perspectives on uncertainty, some researchers have considered uncertainty and risk as separate constructs, while others view them as interchangeable. We shall attempt here to untangle the difference.

Risk, most simply, is the probability of a future negative event multiplied by the adverse impact on outcomes of the event (Boehm 1991). Both the probability and impact are uncertain because outcomes are unknowable (otherwise, they would be certain, and there would be no risk), and an actor undertaking risk analysis likely lacks a complete understanding of cause–effect relationships or fails to consider unknown variables (Milliken 1987). Impact and probability are both uncertain due to the complexity of contributing factors, and ambiguity in understanding the cause–effect relationships involved in predicting outcomes. So, risks involve uncertainties, but with respect to a desired outcome. Uncertainties may imply risks, because events may occur in addressing the uncertainty and these events may have adverse effects on outcomes. However, not all uncertainties imply risks, as there may be no comparative final state, or the difference between possessed information and sufficient information may have no bearing on a desired final state.

Risk concerns itself with negative outcomes (it considers only adverse effects), yet is connected inexorably to uncertainty. Like two sides of the same coin, uncertainty considers the gap prior to an event, while risk considers the (negative) outcomes of an event.

In software development and project management, the final state is presumed to be program completion or delivery; at a somewhat more micro scale, completion of a software requirement. Thus, the software development literature sometimes use the terms uncertainty and risk interchangeably (e.g., Ramesh et al. 2010)[5]. This research

[5] Unfortunately, the same development literature also occasionally conflates risk with risky behaviors (that is, behaviors that increase risk).

considers uncertainty to be a psychological state of an actor, while risk is defined as an objective (albeit, objectively uncertain) probable outcome. Still, not many researchers in software development respect this distinction, and it is sometimes useful to treat risks identified by some researchers as uncertainties.

In the narrower field of requirements uncertainty, by definition, requirements uncertainty occurs at early project stages, while “residual performance risk”, by definition, is risk that occurs at later stages of a project (Na et al. 2004; Nidumolu 1995; Nidumolu 1996). Research into requirements uncertainty takes the view that requirements uncertainty is a driver of performance risk, and views performance risk as the difficulty in estimating what a project’s performance is likely to be, that is, a lack of information about project *outcomes*. This research stream considers requirements uncertainty as a lack of information regarding the *inputs* to a project (Nidumolu 1995). Nidumolu (1996), in choosing to define risk constructs as occurring after the design and analysis phases, frames the distinction between uncertainty and risk partly as temporal, suggesting that performance risk as measured at different times in a project would vary greatly, as major decisions (i.e. more information) such as elapsed time or project costs would become available as the project progressed.

Given our focus on how requirements travel across vertical and horizontal boundaries, it is the broader task uncertainty—not the narrower perspective of requirements uncertainty—that is of interest. Outcomes from one part of the organization or process may be inputs to a different part. Yet, the distinction between focus on inputs and outcomes is useful, because it highlights the traveling of information across organizational boundaries. Further, understanding that risks are an obverse of uncertainty permits the inclusion of a broader range of uncertainty research.

3 PACKAGED SOFTWARE

This chapter reviews packaged software literature, and makes the case for the domain of packaged software. The anticipated effects of packaged software on organizations and development processes are contrasted with development of other types of software.

3.1 What is Packaged Software?

The study of packaged software (Carmel and Becker 1995) is a distinct subset of the IS development literature. Packaged software is usually contrasted with custom development; it imposes different demands and constraints on the development process that are not found in all settings, such as time-to-market pressures, particularly at the industry and firm level (Sawyer 2000). Within the requirements engineering literature, the notion that packaged software behaves differently is being accepted (Regnell et al. 2001), noting in particular that time pressures often lead to incremental releases, accomplished by recurrent development. As Xu and Brinkkemper (2007, p. 533) point out, "The boundaries distinguishing shrink-wrapped software, commercial off-the-shelf software (COTS), packaged and commercial software are blurred, but the principle of 'Make one, sell many' is a common to them all."

A similar notion exists in the software engineering and requirements literature, where packaged software may be referred to as market-driven software (e.g., Karlsson et al. 2007) or as COTS (Commercial Off-The-Shelf). No standard empirical definition of COTS exists, although Torchiano and Morisio (2004) adopted a broad definition for their empirical study ("[software] acquired from a vendor and used as-is or with minor modifications" p.90), and COTS software has been described as systems which meet the following criteria (Basili and Boehm 2001):

- The buyer has no access to developed source code,
- The vendor controls development,
- The software serves multiple customers (non-trivial install base).

In addition, it has been hypothesized that COTS products typically have a new release every eight or nine months, although there is wide variation in the population of COTS products (Basili and Boehm 2001). Although the weakened bargaining position of customers relative to providers of COTS software suggests customers or integrators (those using COTS components to build a COTS-based system, or CBS) would have no input into COTS development, some researchers assert an interaction with a COTS software component provider is important (Jingyue et al. 2009; Torchiano and Morisio 2004).

As may be seen from these definitions, the vast bulk of COTS research is regarding development of systems *with* COTS components, rather than *of* the COTS software itself, so its applicability to this research is limited. It does, however, provide validation for the claim that development of packaged software imposes unique contextual constraints as compared with software development generally, or "one-and-done" internal software projects commonly reported on in the IS literature.

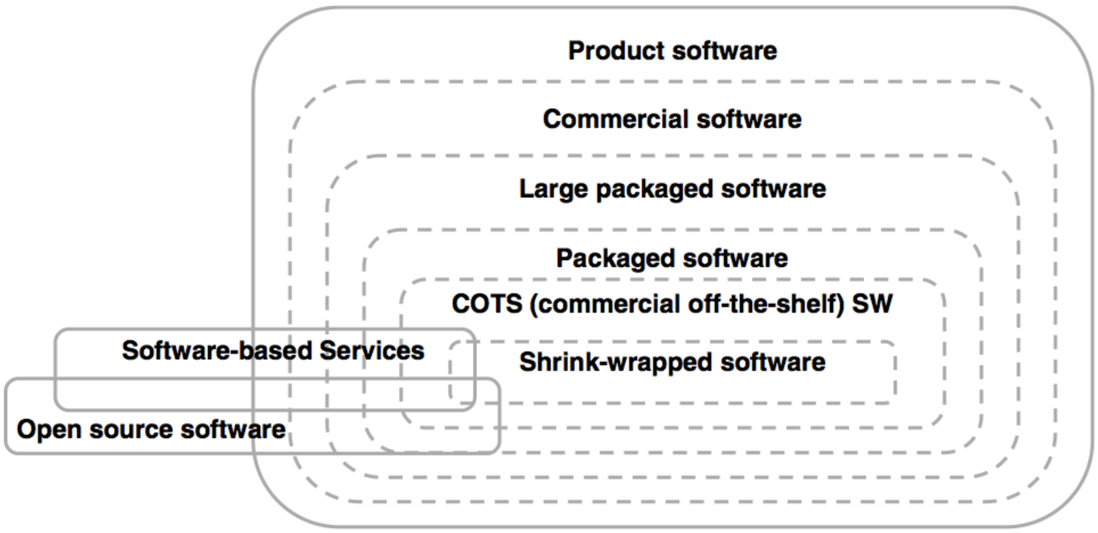

Figure 3–1: Product Software as described by Xu and Brinkkemper (2007)

Thankfully, Xu and Brinkkemper (2007) attempted to clarify murky boundaries and synthesize the several terms used in research under the umbrella "product software":

- *Shrink-wrapped software* is the mass-produced type typically sold in stores, boxed and shrink-wrapped. More modernly, this category might include software downloadable from the Internet, such as via the Mac App Store. Shrink-wrapped software is intended for large volumes of customers.
- *COTS software*, as with shrink-wrapped software, targets a market rather than individual customers. In contrast to shrink-wrapped software, it may be a component rather than a stand-alone software package. Further, COTS software may be part of a complex system ("complex COTS" or "customized information system").
- *"Packaged software* describes ready-made software products that can be readily obtained from software vendors and which generally require little

modification or customization" (Xu and Brinkkemper 2007, p. 534). According to Xu and Brinkkemper (2007), packaged software modernly refers to large enterprise software systems, such as ERPs and CRMs, that although despite being available "out of the box", often require some customization to be ready for use that may take weeks or months for large packages.

- *Commercial software* is controlled by licensing restrictions, and is typically available via retail outlets.

The difficulty with the classification scheme presented by Xu and Brinkkemper (2007) is that categorizations are based on multiple dimensions that are not kept consistent throughout: market versus niche orientation, retail channels utilized (and this dimension is inadequately elucidated for modern software delivery), and whether source code is publicly, privately, or not at all available to the end user. The distinction between COTS and shrink-wrapped software is not clear, except for a reference to its physical packaging, which is becoming less and less relevant in an era of digital distribution. Using the language of Xu and Brinkkemper (2007), it is similarly difficult to distinguish between COTS software and packaged software. This, indeed, may be their point (despite their diagram), that as a field we have attempted to classify software on changing and vague external attributes rather than focusing on the different pressures that affect its development.

Considering open source software as a distinct category further muddies these classifications. Most definitions of shrink-wrapped software, COTS, and packaged software suggest the software source is, by definition, not available. It is not clear, however, whether this distinction marks useful differences in how software is developed by formal organizations. For example, the source code for Mozilla Firefox is publicly available[1], although only the smallest fraction of users even views it. Similarly, the open source database MySQL[2] meets all of the criteria for packaged package software except source code availability; it has been developed by a corporation throughout its history[3], and is likely to experience many of the same effects during its development as other packaged systems. The same could be said of SugarCRM[4], or any other large open source system with a community–enterprise (or similar) dual-licensing model. The distinction of whether a packaged software product is open source (itself a muddy term) is then only useful if measurable effects of that classification are distinct from closed source packaged software products. It may be that the effects researchers have observed in open source development stem from the nature of the software (packaged versus custom), rather than the license of the resulting source code. Torchiano and Morisio (2004) concur that open source software can act as a COTS product, particularly in situations where the packaged source code, although available, is not modified, and the software is treated as if it were closed source.

[1] https://developer.mozilla.org

[2] http://dev.mysql.com

[3] MySQL was initially published in 1995 by MySQL AB, which was purchased by Sun Microsystems in 2008, who were then wholly acquired by Oracle in 2010.

[4] http://www.sugarcrm.com

3.2 Research Opportunity

Although packaged software has demonstrably unique characteristics, it is still considered “a poorly understood phenomena in the information systems research community” (Light and Sawyer 2007, p. 527). A special issue of the *European Journal of Information Systems* (*EJIS*) in 2007 brought attention to the issue, but packaged software remains poorly represented in published research. In their editorial introducing the issue, Light and Sawyer (2007) argue that although ERP systems, which are manifestations of packaged software, have been widely studied in IS, such research tended to be too specific to ERPs, or too broadly generalized to systems development, without consideration for the differences between packaged and custom software. The special issue adopted the same slant as the reviewed requirements literature, as three of the five articles in the special issue of *EJIS* emphasized the consumption and use of packaged software, rather than its production.

This suggests the packaged software literature is immature, incomplete, or simply muddied by its mixing with research on custom or internal development. There is an opportunity for researchers to tease apart which effects are due to market orientation (custom, niche market, mass market), customer segmentation (business market, consumer market, or both), source code availability (open, community–premium hybrid, or closed), or product complexity (stand-alone applications versus systems). In short, we as researchers have done a disservice to our field by considering all software development to be alike, and have not developed a consistent language to permit the teasing apart of observed effects based on characteristics of the developed software.

Additionally, one under-emphasized aspect of packaged software is that its development is recurrent, meaning the same organization regularly revisits the code base and produces incremental versions for the market. Such iterating leads to a shared vocabulary and increased organizational learning, which may, over time, result in reduced uncertainty due to translation as requirements travel. Further, one particular form of traveling is that a requirement may occur in multiple release cycles: a requirement may be in a backlog but not prioritized high enough to receive attention in a particular release. Alternatively, the organization may spend some time developing toward a requirement that is not included in an initial release, but is fully realized in a future one. Thus, requirements in packaged software may not only travel across an organization but also in time across releases. This is in addition to any refinement or elaboration that might occur (features related to a specific requirement expand or evolve over time). These are examples of traveling that would not be possible if development were not recurrent, as it is in packaged software.

3.3 Contrasting "Packaged" and "Custom" Software

For simplicity, this paper adopts two broad categories of software, packaged software, and custom software. In general, custom software targets a single customer (or trivially few customers), whether the organization itself or, in the case of contract development, via an outsourcing arraignment. It is produced in or for a single project (Torchiano and Morisio 2004). Packaged software, on the other hand, targets more than a trivial number of customers, which leads to different behaviors in exploring and managing software requirements (Sawyer 2000; Xu and Brinkkemper 2007). Packaged software is intended to be a “going concern” (as accountants might say it), that is, to have an extended useful life, and is developed over recurring cycles with the intention of long-term maintenance and improvement. As Xu and Brinkkemper (2007) point out, this has implications for the likely level of care taken in architecting the software. Packaged software faces time-to-market pressures (Sawyer 2000; Xu and Brinkkemper 2007), although this may also be true of contracted custom software. More specifically, packaged software organizations are more likely concerned with maintaining schedules than with project costs (Sawyer 2000).

Some of these effects (e.g., differences in requirements management) may depend on the complexity of the developed product instead of on whether the software is packaged or custom; it may be that the greater the complexity of the software, the more likely it is to be productized so that development costs are shared among multiple customers. Some of the observed effects may also be industry-level effects.

Using two small case studies and a review of practitioner and academic literature, (Sawyer 2000) proposed what is probably the most well thought-out list of differences between packaged and custom development (Table 3-1). There are some weaknesses in Sawyer’s (2000) analysis—which should be expected, as Sawyer (2000) refers to these as empirical speculations rather than empirical results—namely the similarities in the sampled teams: all were focused on “small” products or product components (rather than complex systems or networks of products), all were delivering a first-generation product (rather than a new version of an existing product), and two of the three teams studied were in small, isolated settings. Still, Sawyer’s (2000) article is perhaps the most comprehensive examination of the differences between product and custom development.

Packaged software companies' emphasis on time constraints rather than cost constraints has already been noted; Sawyer's (2000) contended this was because packaged software companies tended to be very rich (large and established) or very poor

Table 3-1: Differences Between Packaged and Custom Software (Sawyer, 2000)

	Packaged software	Custom software
Industry	• Time to market pressures • Success measures: profit, market share, mind share	• Cost pressures • Success measures: satisfaction, user acceptance, ROI
Software Development	• Line positions • User is distant and less involved • Process is immature • Somewhat integrated design and development • Design control via coordination	• Staff positions • User is close and more involved • Process is more mature • Separated design and development • Design control via consensus building
Cultural Milieu	• Entrepreneurial • Individualistic	• Bureaucratic • Less individualistic
Teams	• Less likely to have matrix/project structure, more likely to be self-managed • Involved in entire development cycle • More cohesive, motivated, jelled • Opportunities for large financial rewards • Likelier to be small, collocated • Share a vision of their product(s)	• Matrix managed and project focused • People assigned to multiple projects • Work together as needed • Salary-based • Grow larger over time and tend to disperse • Rely on formal specifications/documents

Table content wholly from Sawyer (2000), p. 50 (just starting out). Other differences between packaged software and custom software deserve mention as well (Table 3-1). Sawyer (2000) speculated packaged software companies would measure success by profit and market share, while internal or custom development would measure use, satisfaction or return on investment. The software development teams in packaged software organizations were more likely to be central to the organization's structure and focused on individual skill and individual achievement, and processes are adapted or evolve around an individual's strengths. In contrast, developers in custom software development, Sawyer (2000) contends, are typically relegated to staff positions where process dominates and development resources are fungible. Developers of packaged software are more likely to be separated from users by intermediaries, more likely to have incentives based on project success, and more likely to be self-organizing (Sawyer 2000).

Despite the arguments of Sawyer (2000), many of these team-level effects may have more to do with the size of the organization than with the type of software being developed, for reasons that will be discussed in the research findings.

4 REQUIREMENTS MANAGEMENT

Requirements management literature is summarized using both software engineering and information systems perspectives, emphasizing the lack of interactions between the two streams. Requirements processes are described as iterative and parallel. Requirements are related to uncertainty.

Requirements and requirements management exist at an interesting intersection in the literature. The software engineering (SE) side provides a robust literature, with specialized conferences and journals examining specific aspects of requirements engineering, including elicitation, analysis, specification, and validation. However, it does so without typically examining organizational or systemic considerations. In contrast, the IS literature on requirements lacks the strong canonical foundation found in SE literature, and tends to address processes and approaches in a fragmented way (Hassan and Mathiassen Forthcoming)[1]. External reviews of the whole of requirements literature provide a basis for categorizing uncertainties revealed through requirements during the software development process.

Collectively, requirements engineering activities, along with integration of requirements engineering activities into project management are considered by this dissertation to constitute "requirements management."

[1] This is perhaps analogous to the state of agile development method in IS literature described by Baskerville, R., Pries-Heje, J., and Madsen, S. 2011. "Post-agility: What follows a decade of agility?," *Information and Software Technology* (53:5), pp. 543-555., that is, requirements management and requirements construction are something generally understood at a high level but it is difficult to describe current contributions with precision.

4.1 Requirements in the SE Literature

As with many bodies of knowledge, software practices may be usefully categorized as "generally accepted", part of specialized sub-fields, or those practices still being tested and researched.[2] The "Software Engineering Body of Knowledge" (SWEBOK) adopts this approach, and includes only generally accepted practices. With this in mind, any rigorous description of requirements practices beyond "generally accepted practice" is useful to the field as a whole, as descriptions of practice are used to refine the body of knowledge as the field progresses. SWEBOK has gone through several iterations, and its third version (dubbed SWEBOK V3) was released at the end of 2013, having completed its public review period. In contrast, SE review articles highlight the breadth of the field and suggest directions for future research in requirements engineering (Cheng and Atlee 2007) and requirements management.

A *requirement* is in SE defined as "a property that must be exhibited by software developed or adapted to solve a particular problem … An essential property of all software requirements is that they be verifiable" (SWEBOK 2013, p. 2-4). Working with software requirements is "not a discrete, front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle" (SWEBOK 2013, p. 2-4). Despite this espoused integrated view, SE researchers make the distinction between *requirements engineering* as a pre-

[2] This general taxonomy is used by IEEE in the development of SWEBOK, and is adapted from the "Project Management Body of Knowledge."

A Guide to the Project Management Body of Knowledge, 2000 ed., Project Management Institute, www.pmi.org.

development activities by which requirements are built from nascent ideas in to fully formed descriptions of architecture, function, and expectation, and *requirements management*, or the umbrella of activities involved in managing large numbers of requirements, such as ensuring traceability or analyzing trends (such as stability) in requirements over time (Cheng and Atlee 2007). However, in this dissertation we treat requirements engineering and management activities collectively as requirements management.

Broadly, SWEBOK separates requirements activities into elicitation, analysis, specification, and validation. SE literature tends to treat them as discrete and consecutive, but these need not be sequential; for example, analysis and specification may be alternative and iterative. Nor need these activities precede the beginning of development. Uncertainties may arise during development that require additional specification (and thus, potentially, analysis and validation). Of these activities, validation is perhaps the most motley, referring both to verification that requirements are understandable (implying uncertainties may arise) and meet company standards—a characteristic, it should be said, that is not fully knowable *a priori*—and also referring to the application acceptance tests on the developed software to ensure requirements have been met.

However, despite its strengths in developing a common vocabulary and basis for discussion, SWEBOK does not provide much discussion of management of requirements (beyond suggesting they be managed via a change control process). It also does not consider the effect of vertical information systems on requirements management (beyond suggesting ad hoc methods—such as using spreadsheets—may be less effective) nor does it discuss sources of uncertainty as requirements are communicated across vertical and horizontal boundaries in large organizations.

Implicit in the background of discussion of agile requirements is the fundamental nature of requirements: to transfer the desire of a stakeholder to the understanding of the developer, and express that desire in working software. Thus, agile methods serve to reduce degrees of separation and permit dialog between requirements holders (customers) and developers as a means to manage uncertainty, with the dual goals of 1) reducing interpretation errors, 2) overcoming uncertainty through speedy feedback (Cao and Ramesh 2008). In the agile SE literature, requirements engineering is formally recognized as parallel, iterative and incremental, in a way designed to separate it from "traditional" requirements engineering (Cao and Ramesh 2008), even though these activates exhibit the same traits in "traditional" settings (Hickey and Davis 2004).

4.2 Requirements in the IS Literature

Despite the desire of much SE literature to treat requirements as infallible directives (c.f., Sillitti et al. 2005), IS researchers know that requirements have inherent uncertainties, and reflect the culture, knowledge, and (possibly flawed) interpretations of those writing them (King 2013). Although software developers might wish to eschew the uncertainty inherent in requirements—as evidenced by the continued development of formal specification requirements languages (e.g., Heymans and Dubois 1998), which constitutes its own niche area within the requirements literature—uncertainties remain in requirements so long as they are interpreted by developers. Investigation of the human aspect of requirements construction seems to naturally fall within the domain of

IS development (ISD) research. Unfortunately, ISD literature has little to say about these facets of requirements construction. That is not to say these topic are untreated in the IS literature; the "classic" paper by Davidson (2002) serves as an excellent counter-example. Still, requirements construction as a body of knowledge remains unsettled and infrequent within IS research (Hassan and Mathiassen Forthcoming).

Iivari et al. (2004) argued that requirements construction—identification and specification of the needs of users—should be one of the knowledge areas for which IS researchers could provide "distinctive competence" (p. 322) that contributes to a settled body of knowledge, and further argued "requirements construction continues to be the major bottleneck in ISD" (p. 323). Indeed, in their analysis of articles in *MIS Quarterly* and *Information Systems Journal* between 1996 and 2000, requirements construction was a prominently featured topic, although spread across a number of development contexts (e.g., business process redesign, groupware, decision support systems, etc.), representing a fragmentation of knowledge and approaches.

More recently, Hassan and Mathiassen (Forthcoming) argued for a settled contribution and body of knowledge in the IS literature[3] through citation and *n*-gram analysis of classics[4]. They demonstrated requirements construction classics represented a tiny fraction (3%) of ISD classics, only thirteen articles, despite being one of the categories Iivari et al. (2004) and King (2013) claim IS researchers should be able to

[3] Their search was confined to the Senior Scholars Basket of Journals, a list of the top eight journals in the field: *MIS Quarterly*, *Information Systems Research*, *Journal of Management Information Systems*, *European Journal of Information Systems*, *Information Systems Journal*, *Journal of Strategic Information Systems*, *Journal of the AIS*, and *Journal of Information Technology*.

[4] To be designated a "classic," an article must have been cited at least forty times over a decade.

offer a distinctive contribution. Of the thirteen requirements construction classics (Appendix A), only a couple are modern considerations of requirements processes. Two are a mix of very specific context with an internal customer (executive information systems) and contain requirements advice that while valuable, represents generally accepted practices. Another article considers the systems analyst. The remainder use requirements processes as a context or example application for exploring broader theories of knowledge sharing, cognitive fit, project failure, modeling and boundary spanning.

The consequence of the IS tradition considering these broader theories is grander, more generalizable theories, which may be contributing to the neglect of ISD research focusing on requirements construction. This is, perhaps, an example of the trend Benbasat and Zmud (2003) identify when they claim IS researchers are "under-investigating phenomena intimately associated with IT-based systems and overestimating phenomena distantly associated with IT-based systems" (p.183). Maybe, similar to what Weber (2003) observed regarding the state of research on conceptual models and designing databases in the 1980s, requirements construction has been co-opted by related disciplines (in this case SE).[5] This explanation is supported by recent journal analyses (Lowry et al. Forthcoming), which take the position that most

[5] Perhaps the other reason Weber (2003) describes is also true: IS researchers as a whole may not have sufficient undergraduate, post-graduate, or professional experience to examine the details of systems development or requirements construction with confidence. Both of these arguments seem to imply research on requirements construction and management is, contrary to the claims of Iivari et al. (2004) and King (2013), best suited to sister disciplines than to IS. This is counter to my experience and expectation, but I will not delve further into that discussion here.

publications of the ACM and IEEE—perhaps the most likely outlets for requirements-related research—are not widely considered "IS journals."[6]

An alternative, if somewhat trite, explanation for the dearth of ISD requirements classics is that the ISD field is volatile, and although it has existed for some time, began to mature at the same time it was being disrupted by agile methods. Thus, while there may be excellent requirements construction papers in IS outlets, such papers may not yet be old enough to be considered classics.

Despite the lack of a canonical, classical foundation, when considering requirements and uncertainty via the traveling of ideas metaphor, this research aligns with the tradition in the IS literature of seeking broader theories of knowledge sharing and uncertainty management in the context of requirements construction. However, it also refocuses attention on the relevant IT artifact (Orlikowski and Iacono 2001): the requirement. In doing so, this research anticipates a deep understanding of requirements construction practices as they unfold within and between units as an important area of ISD research.

4.3 Contrasting IS and SE Requirements Literature

IS and SE approach the study of requirements differently (Table 4–1). Whereas SE literature tends to focus on individual steps of requirements engineering, with occasional perspectives on requirements management, IS literature tends to adopt a

[6] And if outlets are not IS journals, they would likely not be on tenure-quality publication lists for top IS researchers, meaning there is little incentive for non-tenured faculty to consider requirements research.

Table 4–1: Contrasting streams of requirements literature

Software Engineering	Information Systems
• Focused • Problem & solution are distinct spaces • Examines process steps • Requirements management distinct from development	• Contextual • Problem & solution spaces interact • Examines process flow • Requirements management part of development

more holistic and contextual approach. Additionally, SE literature seems to take as assumed that requirements management is extrinsic to development; conversely, IS researchers tend to take the view that requirements management is a development activity, even though use of a computer programming language may not be implicated.

IS researchers recognize—perhaps more explicitly than evidenced in SE—that although requirements management models typically show the steps of elicitation, analysis, specification, and validation as discrete and sequential, in practice they almost always occur iteratively and in parallel (Hickey and Davis 2004) (Figure 4-2), meaning requirements both advance and regress, and may be utilized in multiple stages simultaneously. This is particularly true in weaker forms of requirements management as practiced in agile methods (Ramesh et al. 2010). More than in related fields, IS are more likely to apply theories of reasoning, sense-making, and social interaction. IS researchers seem to consider contingent contextual factors and holistic system-wide consequences when selecting or recommending requirements methods, which may contribute to the lack of cohesion discussed by Hassan and Mathiassen (Forthcoming).

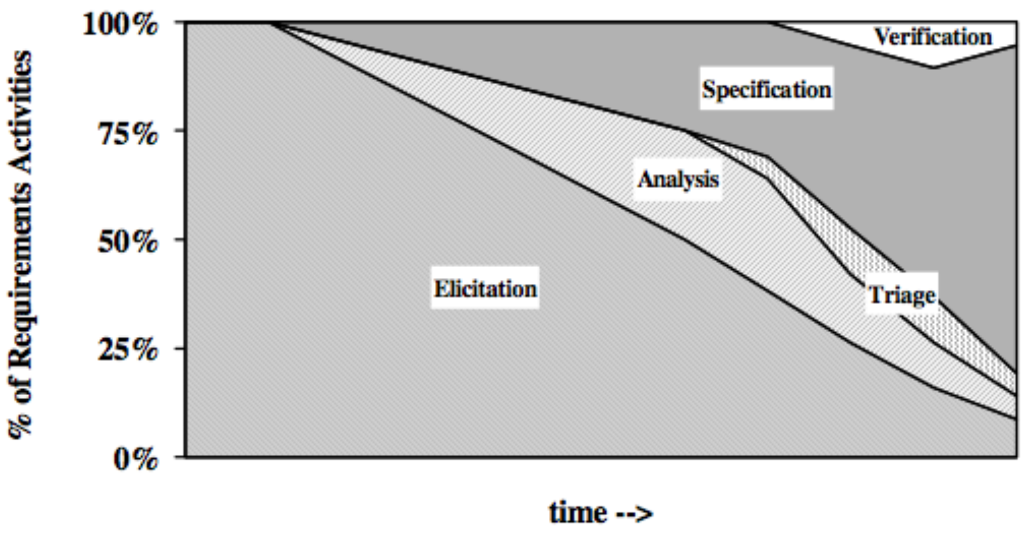

Figure 4–1: Parallel Model of the Requirements Process, per Hickey and Davis (2004)[7]

As Hickey and Davis (2004) noted, IS researchers, as with SE researchers, use a multitude of terms to describe the same requirements management activities, although this has perhaps settled somewhat since the publication of SWEBOK:

> "There is little uniformity in the industry concerning names given to these activities (Siddiqi and Shekaran 1996). For example, to paraphrase Hickey (1999), Davis (1993) defines two activities: problem analysis and product description. Graham (1998) defines two activities: requirements elicitation and requirements analysis. Zave (1997) defines three activities: elicitation, validation, and specification. Jarke and Pohl (1994) define three activities: elicitation, expression, and validation. Pohl (1996) defines four activities: elicitation, negotiation, specification/documentation, and validation/verification. Finally, Thayer and Dorfman (1994) define five activities: elicitation, analysis, specification, verification, and management." (footnote, p. 82; internal citations reformatted)

Additionally, (Hickey and Davis 2004) demonstrated, as previously mentioned, that the requirements activities occur iteratively and in parallel. (Figure 4-2) This is

[7] Hickey and Davis (2004) use "triage" to mean determining which groups or requirements will be addressed in a release. Some authors (e.g., Ramesh, et al., 2010) consider this part of analysis, offering yet another example of unsettled definitions.

consistent with research by Davidson (2002), who treats requirements as social constructions elucidated over time through social interaction as actors engage in resolving ambiguity. Davidson (2002) also found that because of the social nature of requirements, interactions were not consistently recorded in requirements documents. Thus, the requirements documents inconsistently addressed the assumptions underlying particular requirements (even if later uncovered), and tended to reduce the value of the documents. In both of these studies, however, uncertainty is best represented by identity concerns; complexity and volatility as aspects of uncertainty are not obviously considered.

On a more basic level, however, the different fields of requirements and IS consider the nature of "development" and where it exists in the organization in drastically different ways. Cheng and Atlee (2007) describe the difference this way:

> "In general, the research challenges faced by requirements-engineering community are distinct from those faced by the general software-engineering community, because requirements reside primarily in the problem space, whereas other software artifacts reside primarily in the solution space. That is, *requirements* descriptions, ideally, are written entirely in terms of the environment, describing how the environment is to be affected by the proposed system. In contrast, other software artifacts focus on the behavior of the proposed system, and are written in terms of internal software entities and properties. Stated another way, requirements engineering is about defining precisely the *problem* that the software is supposed to solve (i.e., defining *what* the software is to do), whereas other SE activities are about defining and refining a proposed software *solution*." (emphasis original)

The distinction between problem and solution spaces exists in academic requirements research, despite the apparent incongruity of portions of the requirements management domain such as analysis, modeling, and verification residing in, or at the very least bridging, the problem and solution spaces.

Although requirements may flow into the firm from multiple sources, IS researchers tend to either observe the vendor–client dyad or scope their research to the boundaries of the firm. This dissertation adopts the latter approach, and is thus less concerned with discovery that often occurs via connections beyond firm boundaries (as discussed in the robust requirements elicitation literature), but focuses on those interactions that exist within the firm's processes, such as requirement exposition, as well as the traveling and translation of requirements into software. This narrower focus aligns with the accepted requirements engineering dimensions of specification, representation and agreement (Pohl 1994). Although creation of the formal requirement artifact ideally relies on interactions with customers, the actual artifact creation, that is, instantiating the idea as an artifact usable by stakeholders within the firm for the purpose of software development, is an activity that often occurs within firm boundaries.

IS researchers challenge the inherent assumptions of much of the requirements literature, particularly as it applies to "traditional" (or "plan-based") development (as opposed to "agile", "organic", "ad hoc", or "flexible" development).[8] These assumptions generally presume, contrary to what is asserted in this research, that most uncertainty

[8] For a discussion of the differences in software methodologies, see, e.g.:

Baskerville, R., Pries-Heje, J., and Madsen, S. 2011. "Post-agility: What follows a decade of agility?," *Information and Software Technology* (53:5), pp. 543-555.

Harris, M.L., Hevner, A.R., and Collins, R.W. 2009. "Controls in Flexible Software Development," *Communications of the Association for Information Systems* (24), pp. 757–776.

can be resolved prior to development. More specifically, these assumptions include (Ramesh et al. 2010; Sillitti et al. 2005):

- The customer is able to specify all needs up front, prior to development.
- One or more stakeholders are in charge of requirements gathering activity.
- The development team readily understands customer needs.

The first of these seems to assume a context of a single customer (or markedly few customers), contrary to the assumptions of packaged software. The second seems to imply requirements construction activities take place within a single function (although the phrasing "… or more" offers enough wiggle room to be so universally true as to be unhelpful). The last of these assumptions is challenged by the already cited literature on requirements uncertainty. None of these assumptions are wholly useful in the intended context of this research (discussed more fully in Chapter 7). This may be because development practices in modern software organizations blur the line between flexible and plan-based methods (Baskerville et al. 2011; Harris et al. 2009).

Lastly, the literature on both sides—SE and IS—is preoccupied with the *process* by which we manage requirements (whether through emphasis on steps or flow), with insufficient emphasis on the *product* for which requirements are managed. The differences between software products developed by organizations is so large that any claim to a generalized process is weakly grounded (Lee and Baskerville 2003; Thompson and Perry 2004). Processes and steps that work well for one organization and product may not work for another.

Attempts have been made to synthesize the whole of requirements literature. Mathiassen et al. (2007) reviewed 116 articles across both IS and SE streams. As part of their review, they classified requirements techniques as discovery, prioritization, experimentation, and specification techniques. Unsurprisingly, these activities reflect, but do not map directly to, the commonly held list of requirements activities discussed earlier: elicitation, analysis, specification and verification. Such deviation in language is understandable, even expected, as these classifications were derived from reviewing literature that labels these activities inconsistently, so new language is likely less ambiguous.

The review by Mathiassen et al. (2007) is useful because they identified the three flavors of uncertainty (identity, volatility, and complexity), discussed previously (Section 2.3), and successfully applied them in a summary of the requirements literature. We adopt these labels for this research, and approach the setting through the IS tradition, contextually and holistically.

5 WORK DESIGN AND UNCERTAINTY

Modern and seminal theory of work design is presented and related to uncertainty. Horizontal and vertical work design structures are explicated. The problems arising from particular work design structures are highlighted, and framed as uncertainties. Organizations are similar to software in the sense that they are the result of design as well as emergence.

5.1 Work Design

In framing uncertainty as an information processing problem, Galbraith (1973) suggests the management of uncertainty is one of the purposes of organizations, which can be responded to with differentiation or integration strategies.[1] The contingency theory of organization structure was a response to addressing uncertainty in organizations. The contingency view, that an organization's success depends on the match between the uncertainty an organization faces and its structural ability to process information in response to uncertainty, is commonly held in the organization (Sinha and Van de Ven 2005), information systems (Nidumolu 1995), and project management literature (Jiang et al. 2009). This is sometimes referred to as the information processing view, with the argument that organizations can adopt strategies and structure changes to process more information, although this approach loses some of the richness of early work (Galbraith 1973; Sinha and Van de Ven 2005).

[1] Similar concerns exist in the development of software, where questions of tight and loose coupling between modules are addressed.

Work design is the system of procedures for organizing work (Sinha and Van de Ven 2005). It goes beyond individual jobs and examines the broader view of the organization or system along with its attendant support services (Mintzberg 1980; Trist 1981). Work design is reflected in the study of organizations' internal structure, and can affect an organization's ability to access and utilize knowledge and allocate resources (Weigelt and Miller 2013).

Galbraith's (1973) original theory included response strategies to an organizations need to increase information processing (whether due to poor performance or additional information). Although presented as complementary mechanisms and responses to work design, organizations rarely appear fully formed (or fully designed), but rather tend to emerge over time. The common view is that uncertainty leads to responsive structural changes and eventual equilibrium, but an organization's structure is both designed and organic as it constantly reacts to uncertainty (Jauch and Kraft 1986). This is a classic question of design or emergence, and organizations are, as other artifacts (software!), the result of both.

Within the software development literature, there is discussion about the most appropriate work design for software organizations (Austin and Devin 2009). While often framed as a tension between plan-based and flexible software processes (Harris et al. 2009), software processes are adopted based on organization strategy and goals (Slaughter et al. 2006), which reinforces the notion that these discussions of the development process are, at their core, work design issues and attempts to mitigate the uncertainty inherent in software development.

Although the work design literature concerns itself with organizations, its underlying principles apply generally to both work systems and the products produced from these work systems. Parnas (1972), in his classical work on modularity for software, refers to a software module as “a piece of work”, hence directly relating the design of the software to the design of the related development work. Similarly, the general literature on industrial design and innovation emphasizes the duality between the structuring of the producing organization and the architecture of the product being produced and it points to both being nearly decomposable and reflective of each other (Sanchez and Mahoney 1996; Simon 1996). With this commonality, software organization design and software product design become analogies for each other.

5.2 Contingency Theory

In describing organization design strategies, Galbraith (1973) lists several alternatives, most of which can be characterized as facets of the “horizontal” and “vertical” labels used by Sinha and Van de Ven (2005). The first three strategies relate to vertical structures, and comprise a “mechanistic bureaucracy.” First, “rules or programs” are imposed on sub-units as a standardized way of coordinating work. Rules fill the same roles for organizations as habits do for individuals, and are particularly useful for repeated work (Galbraith 1973, p. 10). Such rules or programs, however, require attention and reinforcement by hierarchical authorities tasked with reinforcing processes (Mintzberg 1980). The resulting assumption is that procedures are directed rather than organically coordinated between units. The hierarchy (Galbraith’s second strategy), addresses situations not covered by rule or tradition, and is expected to respond in a way that considers all affected sub-tasks. Thus, hierarchy, which is the epitome of vertical work design, is used to coordinate "in addition to, not instead of, the use of rules" (Galbraith 1973, p. 12). Targeting or goal setting is a third method employed by vertical coordinators in work design, whereby outcome controls (such as goals, requirements, schedules, and design constraints) are set as boundaries for the task, and the organizational unit need not seek approval for work within those boundaries. Additionally, Galbraith (1973) lists four response strategies intended to address failings in the mechanistic model. Creating slack resources, through reducing the level of performance required of an organizational unit, and creating self-contained units (that cross functional boundaries) are two strategies designed to reduce the need for information processing and coordination between units. Similarly, the strategies of investment in vertical information systems and the creation of lateral relations are intended to increase the information processing capacity of units[2] (Galbraith 1973). The four response strategies are suggested as an exhaustive description of an organization's possible responses to uncertainty, with slack resources (reduced performance) occurring by default.

5.3 Horizontal and Vertical Work Design

Modern work design literature recognizes two primary types of boundaries within a work system. These boundaries are imposed with the intention of breaking work into independent pieces. *Vertical division of work*, or hierarchical division, exists within a

[2] Even though investment in vertical information systems is a "response", modern designers of work systems would do well to consider the impact of information systems when designing work. For example, it is not unusual for self-organizing agile development teams to be built around a central information system.

unit, and may include access to resources or knowledge within the strata of a unit. Vertical division may refer to an organization's administrative hierarchy viewed as a collection of subordination and authority relationships, of which there may be one or more, or it may refer to hierarchical decomposition of a work product. *Horizontal division of work*, or modular division (sometimes called "differentiation"), is the imposition of modular boundaries on tasks that may be split in sequence or parallel between organizations. Horizontal boundaries are often related to knowledge or function. Such splits may be within a firm or may cross boundaries of multiple firms in a network (Sinha and Van de Ven 2005).

The defining of internal structures, with consideration of responses to anticipated and actual hierarchical and modular problems, reflect "allocation of decision rights to subunits completing distinct jobs and the coordination among those subunits" (Weigelt and Miller 2013, p. 2). In other words, work design is the allocation (or withholding) of decision rights, and the modularization—and thus necessary coordination—or work across subunits. Moreover, division of work may be tightly or loosely coupled. In a vertical division, a subunit may be granted autonomy or constrained by structures, budget authority, and accountability. Lateral coordination represents the extent to which horizontally divided work units align to complete a task (Weigelt and Miller 2013).

In combination, horizontal and vertical divisions of work enable variegated configurations. In each case, knowledge boundaries exist between work divisions that must be crossed for successful coordination. However, as Sinha and Van de Ven (2005) explain, these divisions reveal problems of modularity and hierarchy, respectively.

Further, when horizontal and vertical divisions of work interact, as they do in practice, a third type, known as network problems, also becomes manifest (See Figure 5-1). Within the domain of software systems, problems of division of work are often solved by identification and application of repeating patterns (e.g., Vlissides et al. 1995). Patterns of organizational design exist as well, although they tend to be rougher and less detailed than software development patterns.

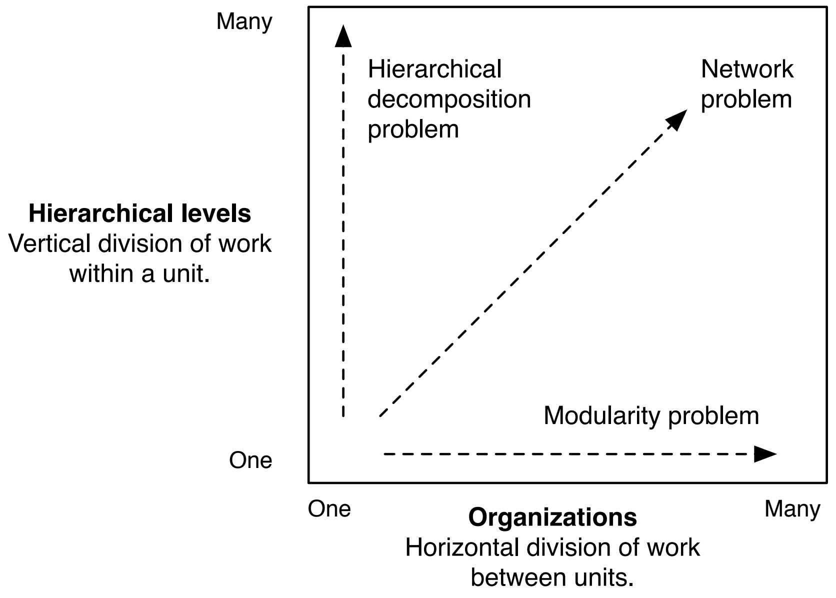
Many
One
Hierarchical levels
Vertical division of work within a unit.
Hierarchical decomposition problem
Network problem
Modularity problem
One
Many
Organizations
Horizontal division of work between units.

Figure 5–1: Conceptualizing Work Design Problems (Sinha and Van de Ven 2005)

A modularity problem considers the division of work and separation of responsibilities between units. Functional and cross-functional teams are examples of organizational solutions to modularity problems, as are functional and product-silo organization structures. Microsoft's recently announced reorganization from a product division to a functional division of responsibility (Balmer 2013) is an example of (primarily) horizontal work design. Outsourcing decisions are also examples of organizational modularity problems. Modular work coordinated between units, which may reside in multiple firms, combine to form modern value chains that comprise work systems. Within the domain of software systems, the model–view–controller pattern permits a separation of concerns within computer code. It has the additional advantage of applying structural rules to a software work system, such that a developer within that system has insight, based on the rules of the system, into where a particular work item should be. This lessens cognitive load, and improves efficiency, as the need to process information is reduced.

In contrast to the modular, loosely-coupled approach, integrated systems and organizations are better suited for tasks that are ill-structured, difficult to decompose, time constrained, or otherwise require a greater need for coordination (Weigelt and Miller 2013). Although it is the antithesis of the modular approach, the choice to adopt an integrated structure is also a horizontal work design decision. As with software, there are trade-offs in adopting the integrated approach to designing organization units. It does bounded tasks quickly and well, but as integrated structures grow in size internal coordination and maintenance also grows exponentially, in contrast to modularized processes which may be easier to coordinate and maintain.

A hierarchy problem considers the coordination and control of work, and allocation of decision rights and knowledge across hierarchical levels of a work system. For example, a hierarchical problem recently considered in IS literature is the ideal reporting structure of the CIO (Banker et al. 2011). Within the domain of software systems, and more specifically object-oriented systems, vertical structures may be expressed by the relationship between a base class (also called superclass) and a subclass. At a high level, the base class, or "senior" object in the code "hierarchy", defines general rules for the system; the subclass is granted decision authority for specific instances of the subclass, applicable to its more targeted needs. The overall structure is simplified and easier to manage when general rules defined at higher levels of the object hierarchy are applied across multiple subclasses. This sort of "embedded coordination," through the application and use of standardized interfaces, acts as hierarchical coordination "without the need to continually exercise authority—enabling effective coordination of processes without the tight coupling of organizational structures" (Sanchez and Mahoney 1996, p. 63). In organizations, such standardization of controls, an imposition of hierarchical authority, is another example of how vertical structures are put in place to reduce uncertainty, and thus project risk (Na et al. 2004; Nidumolu 1996).

Although not explored fully in this dissertation, network problems consider the aggregation of and interaction between horizontal and vertical work designs (Sinha and Van de Ven 2005).

5.4 Work Design and Information Systems

Galbraith would not have been able to predict the strong effect of technology and information systems on organizations. As Orlikowski (1996) demonstrated, technology facilitates modularization of processes and sharing of knowledge across horizontal boundaries. Additionally, some of the earliest uses of enterprise information systems were to enable views of information across vertical boundaries, as evidenced by the requirement construction classics dealing with executive information systems (Watson and Frolick 1993). Digitization of processes (although not technically a "vertical" information system, as Galbraith (1973) predicted), has also led to increased modularity as evidenced by a boom in outsourcing (Davis et al. 2006). Yet one of the biggest challenges organizations face in outsourcing is maintaining coordination across horizontal boundaries beyond the firm.

Perhaps the biggest weakness—and yet most prescient claim—of early contingency theory was an underestimation of the magnitude of the effects of information systems on organizations. Indeed, Im et al. (2013) offered empirical evidence that firms were processing more information with fewer people by investing in information systems. In their variance time-lagged study, they also found IT use is both an antecedent and a consequence of organizational change. Consistent with the predictions of Galbraith, as coordination activities increased, firms would invest in information systems, seemingly as a cost control measure. Such investment would then, over time, decrease coordination costs, and eventually the size of the firm. The evidence is clear that information systems are reducing coordination cost across both vertical and horizontal work boundaries.

Although interest in contingency theory and the structure of organizations waned in the late 1970s, there have been several recent calls in top journals (Sinha and Van de Ven 2005; Zammuto et al. 2007) to reapply contingency theory to modern, technology-enabled organizations. Governance and strategy research have also been criticized for ignoring the internal structure of organizations (Weigelt and Miller 2013). The core of contingency theory is that task uncertainty, as originally described by Galbraith (1973) and others, leads to contingent organization structures. Considering requirements as expressions of task uncertainty (Galbraith 1973) in a software development organization, and following them as they travel through the organization will reveal conditional uncertainties useful for the study of how software development work may be designed. In pursuing such efforts, Sinha and Van de Ven (2005, p. 389) highlight three types of categorical issues relevant to organizational researchers in their call to reopen the study of work design,: "(1) defining the boundaries of work systems, (2) examining how the system is nested in a hierarchy within and between organizations, and (3) determining interactions between the elements of a work system."

6 TRAVELING OF IDEAS

The "travel of ideas" literature is presented and adapted. The central concept of "traveling" is discussed and further dissected to provide greater clarity in a software development context.

6.1 The "Traveling" Metaphor

In explaining the travel of ideas metaphor, Czarniawska and Joerges (1996) argue that in order to become useful, management ideas are sent to places other than where they emerged. Along the way, these ideas are translated into new kind of objects, and this translation is a necessary step in their travel. Czarniawska (2009) summarized how ideas are changed as they move from place to place, arguing the sharing of an idea requires it be newly interpreted. Interpretation and reinterpretation occur every time an idea moves from one place to another or from one point in time to another. Even when captured in an information system, it is still (re)interpreted as the idea passes from the user to the system and from the system to the user. At each time or place, the idea is recreated differently. Although this concept of interpretation is broadly described in the traveling literature and organization studies as "translation," in practice, particularly in the context of requirements and software development, expressing the traveling of ideas simply in terms of translation (in its original meaning) is overly broad. To compensate, this research adapts the traveling framework utilizing concepts from the knowledge management and requirements literature.

Carlile (2004) describes how knowledge may have syntactic or semantic aspects. Syntactic boundaries may be represented by source code, formal specification languages, domain specific languages, or more generally, a common lexicon shared by a group. To *share* an idea is to transfer it across a social boundary while preserving the lexical context used to express it; sharing occurs with a common syntax. However, even with a common syntax, sematic differences arise (Carlile 2004); sharing of knowledge may lead to differing interpretations between the sender and the receiver. As Czarniawska (2009, p. 425) acknowledges, “a thing moved from one place to another cannot emerge unchanged: to set something in a new place or another point in time is to construct it anew” (p. 425). Because sharing may lead to negotiation and trade-offs between actors, it’s considered to exist at the semantic level (Carlile 2004), and such discussions are only possible with a shared syntax. The syntax is itself negotiated over time as actors make trade-offs and share understanding, but such negotiations are only successful when the syntax is settled.

An idea may be *translated* from one syntax to another. This can be as “simple” as documenting tacit knowledge, or storing knowledge in an information system. This definition of translation is much narrower than the one applied by Czarniawska (2009). Explicitly documenting tacit knowledge, as might occur during requirements processes, has been recognized as one of the most critical processes in organizations (Nonaka 1994). When a developer expresses a requirement in code, she expresses the idea anew using the syntax of a programming language. A systems analyst documenting his understanding of a requirement is also engaged in translation. Just as with sharing, translation necessarily results in change to the idea, as the nature of syntaxes causes ideas to be expressed differently due to idioms of a given syntax.[1] Translation may uncover ambiguous meanings (the semantic level); just as linguistic translation can introduce or mask connotations, so too can syntactic translation of ideas. It is for perhaps this reason that Carlile (2004) describe the documentation of tacit knowledge as a semantic, rather than syntactic endeavor, although he recognizes that semantic discussions occur when the idea being presented is novel, or dependencies make meanings ambiguous. It is because of translation between syntaxes (such as the language of external users and internal product managers) that parties attempting to communicate can begin to share knowledge. Because it involves different syntaxes, translation typically occurs across technology boundaries rather than social ones, although strong social boundaries (firms, cultures, countries) also adopt differing syntax. For example, the storage, retrieval and transformation cycle as described by Carlile and Rebentisch (2003) highlight the effects of translation. Using the vocabulary of traveling as adapted in this research, it might be rebranded as a construction–sharing–translation cycle.

As already noted, requirements, expressions of ideas, undergo change—unintended or not—as they are specified. Specification and other uncertainty reduction activities *construct*, or flesh out an idea. This occurs through investigation and elicitation (Hickey and Davis 2004), and is a consequence of work done within or across boundaries. Construction might include such activities as developing test cases for software, and in such cases would accompany either sharing or translation, as

[1] A possible exception to the transformation of an idea during translation might be translating a simple idea from one formal language (e.g., programming language; as opposed to natural language) to another, particularly where the two language have similar syntax.

construction might occur through negotiated understanding across social boundaries, or through the actions of a single person translating their understanding by expressing it in requirements or source code. In a software development setting, construction activities include documenting procedures to validate requirements such as test plans or test cases.

6.2 Conceptualizing Traveling

It is important to note that "translation" in the traveling of ideas literature is used differently than by Carlile (2004). A casual reader of Carlile (2004) might assume from the figures and descriptions that translation is inexorably tied to the semantic level. While this is true, in that successful translation requires consistent semantic understanding, the translation is necessarily occurring because knowledge is expressed in different domains, and thus in different syntaxes.

In contrast, the actor-network theory on which the traveling metaphor (Czarniawska 2009; Czarniawska and Joerges 1996) is based, uses "translation" to mean any reinterpretation or instantiation as an idea is expressed over time and space. Some IS research that adopts the traveling metaphor combines theorization (the building of ideas) with translation (the implementation of ideas) and therefore uses a more targeted view in which translation means "how IT ideas are reinterpreted and implemented in particular organizational settings" (Nielsen et al. 2013, p. 6). However, even this interpretation is too broad. An even narrower view, which treats translation as reinstating to a new syntax is more consistent with the common definition of translation and adopted for this dissertation.

In the same vein, Carlile (2004) expresses transformation as occurring at the pragmatic level of interaction, which is only possible when syntactic and semantic differences have been settled. Much the same way as construction is defined here, pragmatic action, such as determining whether to move a requirement forward through the development process or to discard or postpone it is the trigger for construction. Although Carlile (2004) uses "transform" to mean pragmatic interaction, Carlile (2002) clarifies the pragmatic approach is centered around localized knowledge that is invested and embedded in practice, and that boundary objects that cross pragmatic boundaries do so with the purpose of not only being used for representing and learning about an idea, but also for transforming an idea. Accordingly, in this dissertation, we use travelling of requirements to include constructing, sharing, and translating requirements combined with the understanding that each new instantiation of a requirements is newly interpreted, and may have new meaning for each actor.

In summary, the traveling metaphor is a good complement to the information processing perspective of organizations (Galbraith 1973). Further, complex sets of horizontal and vertical boundaries combine to form networks (Sinha and Van de Ven 2005) through which requirements travel. When combined with the study of work design, the concepts of constructing, sharing, and translating describe the journey of requirements as they travel through organizations.

7 RESEARCH METHODOLOGY

The setting and design of this research are described. We use a qualitative, case study method. GridCo has a structured new product development (NDP) method. Analysis is delineated, and the intended coding scheme is justified.

7.1 Qualitative Case Study

This research adopts a single-site, longitudinal qualitative case study, which is useful for studying contextual factors, particularly organizational structure. Moreover, case studies bring nuance and depth to complex data (Mason 2007), and are appropriate for addressing "how" and "why" questions, particularly in real life contexts (Yin 2009). The blend of technical and human-behavioral aspects of software development lends itself to qualitative study (Seaman 1999). Thus, qualitative methods are the best fit for the research objectives.

Further, as the research unfolds, qualitative methods permit a recursive cycle of inductive reasoning, data analysis, and comparison to extant literature (Eisenhardt and Graebner 2007). Although this may be most notable in the application of the packaged software domain, which is uncovered and explored more thoroughly as data collection progresses, this benefit of the case method applies to other included foundational theories as well. As an additional example, the traveling metaphor fit the goals of the research well, but was not, in its original form (Czarniawska 2009; Czarniawska and Joerges 1996) descriptive enough to inform rich coding of data. Recursive application of reasoning, comparison to observational data, and inductive theory building permitted development of a more descriptive framework as presented in the previous sections.

As with most qualitative research, this dissertation adopts an interpretive perspective (Klein and Myers 1999). Interpretive researchers consider reality to be socially constructed, and assume actors behave according to their respective subjective perceptions (Orlikowski and Baroudi 1991). Socially constructed artifacts may include language, shared meanings, information systems and documents (Klein and Myers 1999). Other researchers have treated requirements as social constructions (Davidson 2002). Interpretive research considers phenomena of interest from the contextual framework of its participants, in their natural setting. Thus it is important to engage in the research setting through observation and interaction (Orlikowski and Baroudi 1991). The interpretive perspective aligns well with longitudinal case studies.

Following initial meetings that occurred in December 2012, we prepared a memorandum of understanding (MoU), also called a researcher–client agreement (Davison et al. 2004). The MoU highlights the role of the researchers and confirms the willingness of GridCo to share particular kinds of data, provide access to employees, and permit observation. It also stipulates that the researchers are responsible for reporting key findings and recommendations to the company. The MoU was signed by both company officers and the researchers before research began in earnest. Additionally, the Institutional Review Board (IRB) for Georgia State University reviewed and approved this human-subjects research.

7.2 Research Setting

The research occurred at a medium-sized development arm of GridCo[1], a large multinational provider of power and smart-grid solutions. The company produces a product ecosystem of utility meters, network storage and routing components, and command-and-control software ("GridWare") that must operate not only on legacy systems, but interoperate with competitor systems and meters, and adhere to common standards using a variety of communication media (e.g., Internet, radio frequency, power-line carrier and cellular). The development arm of GridCo is composed of several hundreds of people. GridCo builds hardware, firmware, and administrative control software via a hybrid process of plan-based and flexible development, using more than 35 small software development teams at multiple locations in the U.S., an offshore captive in India, as well as outsourced development providers.

For several reasons, the industry is dominated by a handful of incumbents (including GridCo). Because customers tend to be large utility providers, the potential market is limited, and there is thus strong competition for a relatively small number of customers. Although GridWare meets the definition of package software as outlined in this dissertation, GridCo both does and does not exhibit the attributes Sawyer (2000) ascribes to organizations developing packaged software (See Chapter 9).

[1] This is, of course, a pseudonym.

Table 7–1: Overview of NPD Stages at GridCo

Stage	Milestone	Stage Description	Contextual Application
Discover	**NPD-0 Start**	Not technically part of the defined and gated NPD process, but listed in the company's documentation. Documentation states, "Ideas are captured and scored using a standardized, cross-functional metric."	No formal ranking or filter processes at this stage were observed. Methods of weighting and ranking other than contractual demands (sometimes with financial penalties) were not referenced by participants. Data imply customer meetings, contracts, and internal R&D are filtered through product area managers at this stage.
Scope	**NPD-1 Scope Ask** Sprint 1/18	The scope for the next cycle is considered, and sized (roughly estimated). High-level scope for the cycle is communicated to executives and project leaders.	Backlog list is almost always oversubscribed, and exists only for current cycle. Development work commenced before scope document complete. Formal scope ask delivered after cycle already started.
Commit	**NPD-2 Scope Commit** Sprint 3/18	Scope is determined feasible and approved. A particular scope is committed to for the cycle. Development formally begins. Change control implemented for further scope/budget changes.	Software development has already been occurring throughout the cycle. Scope commit actually occurred in sprint 9, mid-way through the cycle.
Develop	**NPD-5 Feature Complete** Sprint 14/18	Actual development should be completed by this stage. The product is ready for verification, testing.	Some few items remained, and were continued to be worked, when this stage was to begin.
Verify	**NPD-6 Testing Complete** Sprint 18/18	Testing is complete.	

This research is primarily concerned with a single release cycle of the GridWare command-and-control system during 2013 that was planned to last 36 weeks. We observed meetings from related projects as part of data collection in order to build a richer picture of the release cycle and to observe behaviors similar to those employed but which we could not observe directly for timing reasons. Development of the GridWare system for the observed release cycle depended on related hardware and firmware projects, which increased both the volatility and complexity of the observed requirements.

GridCo uses a series of stage gates that overlay a common technology NPD process. Ostensibly, "gates" describe "go/no-go" decision points; while this was allegedly true for some gates at GridCo, more than one participant intimated that inertia as well as market and contractual demands made continuation of the release cycle all but certain. Indeed, GridCo's NPD overview document read, "The stages are 'soft' meaning that work in a subsequent stage can start before all the deliverables of a prior stage are complete." So in practice, these gates functioned more like milestones. Table 7–1 lists the stages, the end-of-stage milestones, a brief description of each stage, and brief comments about how the stage was implemented at GridCo. These are marked with labels that correspond to stages of GridCo's NPD process.

Observant readers may note that stages in the above table seemingly skip over NPD-3 and NPD-4. Although GridCo utilizes these interim stages for hardware processes, they are not present in software processes. This NPD process is universally mandated at GridCo.

Perhaps the most interesting point regarding the application of the stated NPD process, was that not only did development work commence *before* scope was finalized, it commenced before the high-level scope for the release was trimmed to an accomplishable size, with nothing formally more concrete than knowing of some contractual obligations that would certainly be part of the final scope. The initial NPD-1 date was delayed from February 10th to March 10th (and eventually delivered March 12th), two sprints into the 18-sprint project.

Similarly, the deliverable scope for the release cycle was not committed to until the cycle was half over. This NPD-2 deliverable was a formal event that involved multiple layers of local review as well as presentation to an executive global approval board. The project manager faced internal pressure from his superiors to move the NPD-2 (scope commit) date earlier in the release cycle. Several participants indicated the NPD-2 date had a tendency to move later in the cycle than they would like, but provided no argument for having it earlier other than doing so would be less embarrassing to explain to the executive global approval board. (The NPD-2 review, per the global process, includes a budget and resources request to accomplish the described scope, although most of those personnel resources have already been utilized.)

Indeed, a “late” NPD-2 benefited the cycle as scope was flexible up to that time; any scope changes following NPD-2 approval required change control and executive oversight. This permitted multiple scope changes early in the cycle resulting not only from identity and complexity reasons, but also due to unpreparedness of bottle-necked preliminary work.

Thus, it is important to understand that while employees at GridCo considered this release cycle (and previous cycles) a success, the scope documents against which cycles were evaluated were not set until the middle of the cycle when many uncertainties have already been resolved.

7.3 Data Collection

Data collection occurred over a ten month period, and included interviews and clarifying conversations with fifteen key informants, observations of planning, estimation, review, and approval meetings at multiple hierarchical levels, as well as process documents, meeting and project status summary documents, organization charts, conversations with the company liaisons, and electronic records (Table 7-2). We typically captured data from interviews and researcher meetings with stakeholders as audio recordings, although some subjects requested certain comments not be recorded. Following most meetings and interviews where both researchers were present (most researcher–stakeholder meetings, and about half of the interviews), researchers met and reflected on interpretations of observed interactions, and engaged in dialectic reflection and investigator triangulation (Patton 2005; Yin 2009). These dialogues were also documented. In all cases, researcher notes and observations provided additional sources of data. Importantly, researchers were also given access to the central information system used to store requirements. Towards the end of the engagement, the key findings of the study were presented to the release cycle manager for feedback and discussions.

Table 7-2: Summary of data sources

Data source	Explanation
Employee Interviews	Structured or semi-structured interviews regarding perceptions of requirements, uncertainty, and project stumbling blocks.
Researcher–Stakeholder Meetings	Meetings to define scope of research, summarize practices, describe organization structure, and present key results from the study. Differ from interviews in that these meetings were driven by company stakeholders or collaboratively with researchers.
Meeting observations	Unobtrusive observations of regularly scheduled project meetings.
Project documentation	Project plans, status reports, meeting summaries, stack rankings, change control requests, approvals, and other decision documents produced during the course of the studied release cycles.
Requirements	Metadata, user stories, decomposition, acceptance criteria, as stored in the common information system.
Clarifying conversations	Personal conversation with key informants intended to clarify observations or validate interpretations.
Dialectic reflection	Post data-collection researcher meetings intended to challenge and align perspective to improve reliability. The timing of these meetings also served as an additional opportunity to document and flesh out observations and impressions that might not have been otherwise recorded.
Research notes	All other notes taken during the research. e.g., design of interview or survey instruments, literature reviews, theories investigated, and reviewer feedback.

Functional requirements, which are represented as “user stories”, along with testing requirements, are stored in a vertical information system at GridCo. Multiple views of this data were available. Requirement completion over time was available via project status documents, which are also stored in a central (but separate) information system. We had access to these documents as well.

Each interview lasted between 1 and 1.5 hours. We selected interview subjects to include a mix of positions across both horizontal and vertical boundaries. Interviewees came from across the breadth of the involved processes, and included vice presidents, project and product managers, business analysts, architects, software development managers, release managers, and stakeholders in research and development and quality assurance roles. The perspectives of multiple stakeholders are necessary to contrast interpretations of the internal boundaries of the organization and to enable rigorous analysis of conclusions. Together, the broad view of the company offered by multiple stakeholders and rich sets of artifact data across the complete release cycle permitted triangulation of findings to enhance reliability.

Specifically, data were collected from interviews with 15 stakeholders (two were interviewed twice), 6 discussion meetings, 27 meeting observations, and 12 clarifying conversations, and a final review meeting in which study results were presented and discussed, comprising dozens of hours of recorded audio and more than 200 pages of researcher notes. Official project status update documents, proposals, and presentation slides were also collected. Data from meeting observations was well saturated, and was complemented with official summary documents from many of the observed meetings.

Although this research was initially scoped at the boundary of the firm, this limit was examined over the course of data collection. Product managers were used as proxies for interaction with GridCo's customers, and as a means to validate the scoping decision. As expected, data decreased in relevance close to the periphery, thus the scope boundaries of the research was validated (Yin 2009).

In addition, GridCo permitted access to its central information systems, including not only the central requirements repository, but also a document repository with meeting summary and output documents. Participants relied heavily on the centralized information system as the source of knowledge. As a project manager said, the IS was used as the canonical version of "the truth."

7.4 Coding Structure

The most preferred strategy for analyzing case study data is reliance on theory (Yin 2009). The theories developed in the preceding chapters formed the basis for coding (Table 8-2). Some codes commonly appeared in concert with others. For example, sharing occurs across a boundary, so data on sharing are typically accompanied by indications of the observed work structure. The traveling constructs, sharing, translating, and constructing, represent times when uncertainty is likely to be manifest. In addition to the theory-based codes, the position(s) of the actor(s) involved were also captured.

It is important the list of codes is sufficiently descriptive, mutually exclusive, and (within the scope of the research), collectively exhaustive. The descriptive framework for categorizing requirements groups combined with the theoretically derived descriptions (Table 8-2) were designed to encompass the who, the what (requirement), the when (travel), the where (work design), and the why (uncertainty) so that we could properly address the "how" of the research questions. Although this framework is anticipated to be sufficient, coding is an iterative process that is informed by both theory and data (Miles and Huberman 1994). These codes, combined with the data displays, present a sufficiently rich picture to validate the findings presented in this dissertation.

Table 7–3: Framework for Analyzing Traveling of Requirements

Theory	Code	Summary
Uncertainty (Mathiassen et al. 2007)	Identity	Difficulty in the knowing of requirements caused by communications gaps.
	Volatility	The changing of requirements whether for internal or external reasons (e.g., time, budget, changing market or customer preferences).
	Complexity	Difficulty in specifying and communicating requirements; includes the cognitive load required to understand the effects of implementation.
Work Design (Nidumolu 1995; Sinha and Van de Ven 2005) **Contingency Theory** (Galbraith 1973) **Boundaries** (Carlile 2002)	Horizontal	Modular or serial work design. May involve mutual adjustment.
	Vertical	Formal coordination within a hierarchical structure. Decomposition. Encompasses Galbraith's conceptualizations of both hierarchy and targeting.
	Network	Complex combinations of horizontal and/or vertical boundaries. Included for completeness.
Travel of Ideas (Czarniawska 2009; Czarniawska and Joerges 1996)	Share	The movement of an idea across a boundary. Encompasses changes that occur due to interpretations.
	Translate	The (re)enacting or materializing of an idea in a different form, using a different syntax.
	Construct	The explication of an idea within a given syntactic/semantic context.

7.5 Data Analysis Strategy

Analyzing case study evidence is one of the most difficult aspects of case study research (Yin 2009, p. 127). Preliminary analysis validated the research frame and study scope. To begin, we placed evidence in a matrix of categories and created data displays to provide rich pictures of processes, events, and temporal ordering (Miles and Huberman 1994; Yin 2009). Data displays summarized the organizational context, and highlighted the work design relationships discussed by Sinha and Van de Ven (2005) (i.e., horizontal, vertical, and network relationships) and early contingency theorists. These displays were iterated against the data to present an accurate and informative synthesis.

The bulk of analysis centered on identifying indications of uncertainty within the data, and coding these utterances using the various research lenses described previously, particularly the work design strategies employed, as well as classifying the type of uncertainty represented. We used the types of uncertainty (identity, volatility and complexity) as indicators of the difficulty actors encounter in attempting to accomplish their tasks of sharing, translating, and constructing requirements within the organizational structure and the boundaries created by vertical and horizontal work design (Sinha and Van de Ven 2005).

To assist in selection of specific groups of requirements for in-depth analysis—and consequently, to support data reduction (Miles and Huberman 1994)—we articulated a diverse set of general requirements traveling behaviors. We developed these categorizations based on our professional and academic experience as likely representing interesting types of requirements. We initially categorized the requirements (or groups of requirements) we observed at GridCo into one or more of these categories of travelling:

1. Requirements that behaved as expected (few manifestations of uncertainty).[2]
2. Requirements that were added to the release cycle.

[2] Although requirements in this last group are something of an endangered species, they are also, with few exceptions, uninteresting. The only exception may be those requirements that originated within the development and architecture teams, which may be the reason they consequently seemed to exhibit little uncertainty in their construction.

3. Requirements that expanded in scope.
4. Requirements that contracted in scope.
5. Requirements that were removed from the release cycle.

Although these categories suitably described important traveling behavior within a release cycle, it eventually became evident they did not represent the complexity of observed traveling behavior. We therefore attempted multiple ways of collating and displaying data, which led to the creation and refinement of additional data displays that reflected the traveling we observed from multiple perspectives: the process, the organization structure, temporal structure, and the release cycles of the product itself.

Via categorization of behaviors identified through iterative refinement of these data displays, we identified select groups of requirements described in interviews and project documents that reveal the series of events that led to a resolution of task uncertainty over the life of each group of requirements, and selected those with enough data for narrative completeness. These strings of events reflect the process progression, the organizational structures involved, and the accomplished traveling activity at GridCo, which in turn led to identification of emergent patterns. Simplified displays of release cycle traveling (localized, cross-layer, and cross-cycle) summarized high level-views of how requirements are constructed, shared, and translated (RQ1) and how requirements travel (RQ2). These data displays provided a conceptual foundation for positioning richer detail within its natural context. Models and storylines developed by pattern-matching and explanation-building (Yin 2009) of aggregated and coded data were then subject to verification and attempted disconfirmation through triangulation of the multiple available sources of data.

8 TRAVELING OF REQUIREMENTS

At GridCo, requirements traveled in certain ways within and across release cycles as participants addressed uncertainties. We identified and analyzed three major categories of traveling behavior: local, cross-layer and cross-cycle.

8.1 Types of Traveling

Our original intention in examining traveling of requirements was to consider five types of observed behavior, which we label "localized traveling" that individually and collectively describe the fate of requirements over the course of release execution. However, while each requirement (or set of requirements) may be described using localized language, it became evident that such framing alone was insufficient to fully capture how requirements traveled. Cross-layer and cross-cycle traveling introduced changes to the project's scope across other parts of the company and over time, and reveal additional insights into how requirements travel. Cross-layer traveling describes the experience of requirements with dependencies across multiple hardware, firmware, and software layers. Requirements with dependencies across layers were subject to mid-stream modification (identity uncertainties) and whiplash effects (volatility). Requirements also traveled across cycles, moving between consecutive, overlapping release cycles. This cross-cycle traveling was a response to volatility, and additionally served to postpone or reduce future identity uncertainties. Each of these types of traveling occurred against the backdrop of the standardized NPD (new product development) procedures and a fixed organizational structure at GridCo.

8.2 The Expected Journey

An overview of the work design at GridCo for the observed cycle (Figure 8–1) outlines the expected journey of requirements. Requirements were expected to travel horizontally between functions as tasks were performed. As described in the NPD process, the product portfolio manager considered requirements for inclusion in the cycle based on input from product managers and (for large products) product area managers. These product managers interfaced directly with customers, and acted as customer representatives during the cycle. Planning the centralized C&C system required input from multiple product managers, as the system is a hub in a network of HW and SW products. In theory, developers or the in-house R&D group might present requirements to product (area) managers. However, in the observed cycle, the vast majority of requirements (both by count and by share of effort) in the backlog at the start of the cycle were contractual obligations to current and future customers, which we interpreted as vertical demands.[1]

There was not an overall, continuously maintained backlog of requirements. Rather, a backlog was (re)created for each cycle, based largely on imminent customer obligations, and made more volatile by the interventions of directors and executives through vertical lines.

[1] The included requirements were so weighted toward future contractual demands that one participant exclaimed, "It would be nice if we sold the stuff we did, rather than the stuff we're going to do!"

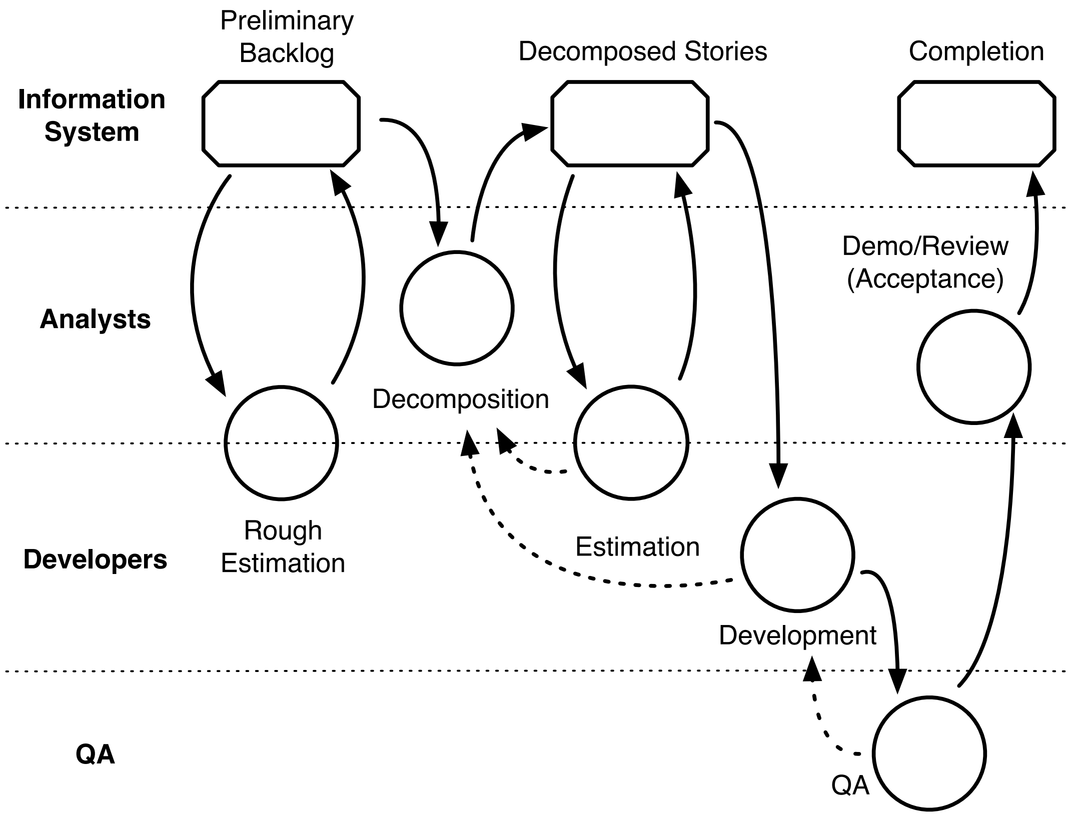

Figure 8–1: Work Design

Prior to the release cycle getting underway, the product portfolio manager constructed a list of items desired for completion in the cycle ("Backlog"). These items were assigned rough effort estimates in a plenary meeting of product managers, systems engineers, architects, senior developers, and quality assurance; essentially every senior person from product development with responsibilities in the cycle was there. These initial estimates (represented as "T-

Table 8–1: Initial Estimation Ranges

T-Shirt Size	Story Points
XS	1–5
S	6–20
M	21–50
L	51–100
XL	101–150
2X	151–250
3X	251–500

shirt sizes", as shown in Table 8–1) reflected a range of story points[2], and were later combined with risk estimates as part of a Monte Carlo simulation to estimate the size of the cycle's capacity and the risk of over-subscription. These tasks were in place to reduce uncertainty (by providing an abstraction layer to manage complexity and other uncertainties), and reflected additional construction of information about the requirements via horizontal coordination.

In a retrospective interview months later, a project manager indicated the backlog review meeting was perhaps too big a production for what it accomplished, and further indicated practice had since been modified so estimation was accomplished (in the following cycle) by a smaller group of people considering fewer items on more regular basis during an already existing meeting. However, at the time of the backlog review meeting, the portfolio manager indicated to those present that the plenary estimation meeting was necessary in order to have enough information to push back on superiors who were apparently demanding items be included that exceeded the capacity of the cycle. Some in the room, perhaps jokingly, indicated the "critical" items slated for the cycle represented four times the available capacity. The portfolio manager intimated following the meeting that the project was 25% over capacity before the meeting even started. The observed meeting was insufficient to size all of requirements on the backlog. In fact, several participants mentioned the over-subscription of work items (as compared with resources and time-to-completion) as a common occurrence at the start

[2] "Story points" are something of an abstract concept representing "nebulous units of time" Rasmusson, J. 2011. *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf., and are sometimes called "ideal days." The actual unit used is not important, the intent is to bring focus to the relative size of different tasks (ibid.). At GridCo, management estimated a story point was roughly equivalent to four or five hours of a developer's time.

of cycles. In the cycle following the one observed, a product manager estimated in a meeting that the over-subscription of the next cycle was an estimated 20%.

Following initial estimation, backlog items were re-ranked with executive input by the portfolio manager. The project cycle was still over-subscribed, but the work of further constructing requirements more fully commenced. Although we have described the development process here in a sequential manner, in practice it operated in multiple concurrent iterations such that decomposition of some requirements was concurrent with writing of computer code. Coding commenced with decomposed requirements from a previous cycle; decomposition could then occur in manageable chunks at a pace slightly ahead of the software developers.

A central group of architects and business analysts decomposed (constructed) backlog items that had been stack ranked and marked for inclusion in the cycle and shared via a central information system. Decomposition included translating high-level requirement descriptions into detailed user stories. As part of the decomposition process, requirements were broken into smaller chunks. As a director described, "[T]he story shouldn't be larger than about seven [story points] worth of work… In general that's the number we're using because of the sprint size that we have, and the amount of work that we believe the teams can complete within that [time frame]." When the story is complete, a senior developer and senior QA analyst, in concert with the business analyst responsible for constructing the story, gives the story a finer estimation of development story points and QA story points. If a particular requirement significantly exceeds seven points, it was usually split into multiple stories[3], although this outcome was negotiated between development, QA, and systems engineering during the decomposition process. Participants offered several examples from their information system of requirements—collections of stories—comprised of upwards of 70 stories.

Translating high-level requirements into stories and properly constructing stories includes not only detailed architectural explanations of the work to be done, but also steps to verify requirement completion and any acceptance criteria. Stories were consciously detailed in order to unravel complexity and guard against identity uncertainties. A systems engineer explained, "We don't know when we're decomposing these stories what team is going to get this, and where that team is going to be located. … We now have maybe 20-plus teams working on this product, with a wide variety of skill level." Consequently, stories were decomposed to be as specific as possible, so even the lowest-skilled teams could accomplish them.

Stories were shared with software developers using the same central information system that stored the requirements. When identity uncertainties arose in the interpretation (translation) of a story, developers tended to contact the person who authored the story, and, as appropriate, a member of the architecture team and product manager (representing horizontal and network connections). Responses that materially altered the design of the story or its test procedures were appended to the story in the

[3] Although each story is itself a requirement, in describing process at GridCo this paper will use "story" to indicate a small unit of work, and, in general, "requirement" to specify a collection of related stories. The phrase "requirement group" means a collection of requirements with a similar theme (e.g., security) or similar purpose (e.g., related to a specific piece of hard ware, or intended for a particular customer).

information system; although we observed this practice, it's not clear how common the practice was.

In addition to communication across horizontal boundaries, we observed travelling across vertical lines. In the above examples (identity and complexity resolution and suggestions), vertical stakeholders were copied on the email conversations, but did not participate. However, when schedule uncertainties arose related to third-party actors—external vendors or internal dependencies on other projects—the observed communication was almost exclusively vertical first. Those high enough on the vertical chain would then communicate across horizontal boundaries and with product and project managers to determine whether items should remain in scope.

Lastly, acceptance of each story required certification by the developer and QA personnel assigned to the story. The delivered software was demoed to the business analyst, who, as author of the story, held ultimate responsibility for acceptance.

This standard procedure of gross estimation and ranking, decomposition, development, and acceptance worked for many of the included requirements. Accordingly, minor uncertainties were easily resolved using expected horizontal and vertical communication lines.

Among the backlog items slated for the release, the set of requirements that most closely adhered to this ideal process were either small in size (fewer than 20 story points) or related to security. These requirements, even at a high level, were understandable to developers and relied on common industry practices. Consequently, there was low identity uncertainty; complexity was similarly low because developers tended to already possess relevant domain knowledge.

8.3 Localized Traveling

Figure 8-2 illustrates the observed localized traveling of requirements. Representing the requirements accepted into a release cycle's scope as a circle, we identified five different types of localized travelling: (A) requirements implemented as expected (the dot within the circle); (B) requirements added to scope (the incoming arrow); (C) requirements removed from scope (the outgoing arrow); (D) requirements discovered to be more complex than expected (the expanded circle); and, (E) requirements discovered to be less complex than expected (the contracted circle). In general, (A) represents low uncertainty, (B) and (C) represent high volatility uncertainty, while (D) and (E) represent high complexity or identity uncertainties. We accounted for our observations of requirements that travelled locally as expected (A) in the previous section, and this description serves as a baseline for the our accounts of the other forms of localized travelling provided below.

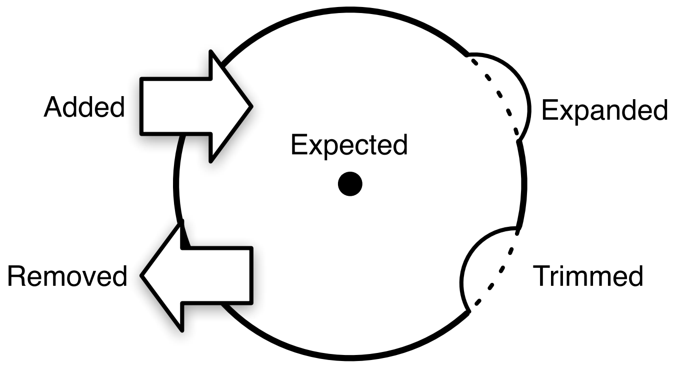

Figure 8–2: Local Traveling of Requirements

8.3.1 Added requirements

Often termed "scope creep," requirements are commonly added to projects after work has begun. Two general categories of requirements were added to scope as the cycle progressed. The first batch occurred early in the project, before the scope was partially or formally fixed (NPD-1 and NPD-2, respectively.) The second collection of added requirements occurred after scope had settled as a consequence of failed coordination between horizontal departments.

Scope was not finalized by the first sprint, even though development had begun. Not only was scope still changing, the rank order of requirements was also changing. A system engineer commented, "Up until two days ago, there were no security features in [this cycle], now, around five of the top ten [requirements] are security features. … This changing of priorities is common. We may see it change again."

Two sprints (four weeks) into the cycle, scope had still not been finalized. The NPD-1 date was initially targeted for the first sprint, but was very quickly pushed back to the middle of Sprint 3. The NPD-2 deliverable, originally scheduled for Sprint 3, was consequently pushed back to Sprint 5. Scope was still shifting early in the project, which was problematic, as decomposition and development occurred concurrently. A project report[4] from the middle of the first sprint indicated. “The current scope for [this cycle] is still not set, but the [development] teams are working from a list of six items which has recently changed from the original list of six.”

A development manager expressed frustration about the later NPD-2 date in a meeting: “We will be 5 sprints in, and not everything will be decomposed. [I’m] already assigning work to teams, but without guidance about the eventual task list, [we’re] assigning based on the current work, not on the best overall fit.” The over-subscription of requirements in the cycle meant that analysts had more requirements to decompose than could possibly be worked. Both the initial reduced list of scope and the eventually decomposed requirements are examples of goal-setting or targeting, indications of vertical control. The responsible functional groups retain decision rights, even though the responsible functions were treated as functionally and socially horizontal within the organization. From an organization design perspective, scope definition and requirements construction both constrain and determine outcome controls for later work, and are thus vertical connections. There was general discontent with the way the project was over-subscribed, and that scope definition was overlapping the development schedule by too much, and thus causing development resources to be slack.

Despite frustration with a moving scope target, development teams still had enough decomposed work. Due in part to delays with firmware dependencies in the

[4] Project cycle status was reported to executive and international oversight weekly. Due to the number of projects in progress at the company, only those projects self-reporting as “yellow” or “red” (as opposed to “green”) received attention in the executive review meetings.

previous cycle, some requirements were change controlled out of the previous cycle. Additionally, the NPD-5 date of the previous cycle was change-controlled from coinciding with the start date of the observed cycle to overlapping with the first two sprints of the observed cycle. The lateness of the previous cycle may have added to delays in finalizing scope. Although the previous development and the scope definition were handled by different groups within the product organization (software and product management, respectively), completing the scope definition required coordination.

A more interesting set of requirements was added during sprint 8, midway through the project, shortly before project scope had officially committed. Due to multiple competing demands on the business analysts and systems engineers, not enough stories had been fully decomposed to match the developers' capacity for the sprint. To prevent the unutilized capacity from being wasted, development began implementing architectural changes they had proposed in a previous release cycle, although these requirements were officially slated for a future release. Interestingly, while not fully decomposed, senior developers had enough familiarity with the intended requirement to deliver code for the sprint. Although discussed later (Section 8.5, Cross-Cycle Traveling), these architectural changes—which created a modular structure for more easily adding support for new meter types to the utility network—had been passed over multiple times for inclusion in the release cycle, in favor of contractual customer demands. However, to best utilize otherwise slack development resources, these architectural improvements were added into scope, not through the normal vertical channels (although they were later formally accepted as part of scope), but by the developers. This presents something of an anomaly from a work design perspective, as the slack was caused by inadequate vertical coordination, but it was filled by the reciprocal relationship through mutual coordination, a horizontal work design mechanism.

8.3.2 Removed requirements

As with added requirements, there were two primary categories of removed requirements: those removed while the scope was still churning, and those removed after the scope was fixed at NPD-2.

The backlog list was quite volatile through NPP-1, as items moved in and out for reasons including availability of hardware, support of third-party vendors, discovered defects, changing customer requirements, and executive support. Such volatility was essential to the release cycle: after NPD-1, a product manager estimated scope still exceeded capacity by between 123% and 137%. One manager expressed frustration with the constantly changing requirements list, saying , "We should make butter in this company, as much as we churn scope." Volatility in accepted requirements continued up to NPD-2.

"We should make butter in this company, as much as we churn scope."

Removal was an essential task in order to accomplish NPD-2. Not only were project resources insufficient for the requested scope, those resources were being used ("burned") by the passage of time, although not necessarily fully utilized. In sprint 6, for example, development managers reported they had assigned work "below the line" (likely to be outside of the cycle scope) to developers, because sufficient work "above the line" had not been properly decomposed. To the apparent exasperation of others present in the status meeting, a product manager asked, “If they’re working items below the line, does that mean capacity has changed above the line?” The answer from several present was a resounding, “Yes!” As development had started, any time not spent on items above the line was irrecoverable.

After scope had settled, other requirements were removed from scope, for reasons of volatility. Early in the cycle, a requirement to support a particular wireless communications protocol was added to the requirements list, as it had been recently change-controlled out of the previous release as the hardware and firmware necessary for development and testing had not arrived from the third-party vendor in time. This requirement remained in the observed cycle through NPD-2 without significant development progress, despite receiving regular attention from product and project management. Project and product managers were frustrated, and moved the problem up the hierarchy. The release manager eventually explained that despite multiple negotiations the vendor was unwilling to provide their newest hardware and firmware versions as the vendor suspected GridCo was developing their own internal versions of the same. Due to this and other schedule troubles with the vendor, GridCo felt it necessary to fully control development of the communications HW and FW in-house, and eventually ceased its relationship with that vendor. However, enough time had lapsed waiting on and negotiating with the vendor that the necessary HW and FW were not ready in time for inclusion in the observed cycle either. Because of the volatility uncertainty encountered, this requirement was removed from the project via change control after NPD-2. However, this uncertainty was managed throughout by constant follow-up and horizontal coordination. In addition, product and project managers regularly sought guidance from vertical authorities on how to address the third-party vendor, for information on the status of the internal replication project, and advice on strategic fit of the requirement.

8.3.3 Expanded requirements

Requirements expand as uncertainty is revealed during the development process, often during translation or construction. Expansion differs from addition (scope creep) in that it is not the addition of new requirements (volatility), but rather the result of a deeper understanding of existing requirements (e.g., identity or complexity uncertainties).

In one example of expansion growing from identity uncertainty, senior developers asked questions about the scope of a decomposed requirement during an estimation meeting. The requirement introduced a software process that might, in certain circumstances, lead to a failure condition. However, steps to recover from the failed state were not specified in the requirement, and developers questioned how that should occur. This led to the creation and inclusion of a new story as part of the requested feature.

An engineer related another instance of expanded requirements due to identity uncertainties. Requirements intended to satisfy one of GridCo's large, strategic customers ("Customer B") were prominent in the pool of requirements for the observed and previous cycles. The engineer explained that initial rough estimation (T-shirt sizing) was typically accurate, but also described an experience where that did not happen: "As we looked at the requirement, we made some assumptions in putting together a T-shirt size, and when they later went back to [Customer B], and said, alright, here's what we think this is, here's the assumption we made, then they shot that down pretty quickly, and said ‘no, you can't make that assumption.’ [We] brought it back, and that doubled the particular requirement scope size. That's the only one that's really been off in its estimation.”

A different type of expansion occurs when developers chose to re-architect software “under the hood” (as one systems engineer described it) in the hope of facilitating a future over-all reduction in work. Although this sort of refactoring was not officially sanctioned, it sometimes occurred and caused small increases in initial development time. In one notable instance, the refactoring work, which had been advocated by development for several cycles but never accepted by product management came in “through the back door” during a time when not enough work had been decomposed. Developers, hopeful that their estimate of a nearly 50% reduction of a particular kind of recurring future work would pay off, began working on the refactor. This requirements was later officially added to the release.

A final type of expansion was observed in the data. During the final four sprints of the release cycle, development is ideally complete, and the quality assurance, or “hardening,” process begins. Defects found in the cycle are sent back to developers for correction. Referring to previous cycles, one manager said, “the thing that kills us every thing is the high number of defects we find during hardening.” When pressed for clarification, this manager indicated the volume of defects were a problem, and largely stemmed from the great number of dependencies in the code. Thus, this type of expansion is a consequence of complexity uncertainty.

One other noteworthy requirement set was expanded during the cycle. Due to expanding international markets, GridCo wished to improve the localizations of its user interface. After an initial framework was in place, requirements related to globalization presented low identity and complexity uncertainties. As other requirements were delayed or faced great uncertainty, addressing globalization requirements, work that was initially intended to be accomplished in future cycles, grew to be a larger portion of the release as an easy way to continue utilizing development resources.

8.3.4 Trimmed requirements

The count of requirements accepted into scope may expand; the inverse is also true: requirements may be removed, or trimmed, from scope.

Requirements intended to satisfy contractual obligations to Customer B also experienced trimming. Some of these obligations aligned with requirements already slated for the observed cycle, but one manager estimated that nearly a quarter of the capacity of the release was devoted to requirements contractually agreed to with Customer B. By two months after NPD-1 (development sprint 5 of 14), these requirements still had not settled sufficiently to be decomposed. GridCo scheduled multiple daylong workshop sessions with Customer B to resolve these identity uncertainties.[5] The unresolved identity uncertainties of the work necessary to satisfy Customer B were evident in multiple status meetings during sprints 5 and 6. During sprint 6, NPD-2 had not yet occurred, and there was continuing concern voiced by

[5] Amusingly, one of these identity uncertainties was resolving what was meant by “etc.” in some of the contracted items.

developers that the cycle was still over-subscribed. Product managers delayed removing items from scope until the Customer B identity uncertainties were resolved. In the end, forced removal was unnecessary because there was an overall reduction in the capacity demands for the requirements particular to Customer B, which relieved much of the capacity pressure in the release cycle. However, some of the requirements were moved to the FW product group, which led to some indirect volatility.

The difference between removal and trimming, as with expansion and scope creep, is centered on resolving identity and complexity uncertainties rather than the volatility of demands. That is, demands do not change, but the understanding of them does.

8.4 Cross-Layer Traveling

Cross layer travelling is important, because it highlights complexity uncertainties arising from inter-dependent layers: requirements traveled between hardware and software in virtuous (or vicious) cycles (Figure 8–3). At GridCo, this complexity uncertainty manifested as schedule volatility for dependent components. Adding support for new utility meters to the C&C software system was a great deal of work. For example, adding support for only a few new meters caused the single largest portion of work and uncertainty in the release cycle. Coordinating the timing of completion between layers was problematic. In some cases, support for new HW was contractually obligated, but the HW itself was still being developed. First releasing the HW and then later releasing updated C&C SW to support it was an untenable option. As a development manger explained, "software and hardware releases coincide because of customer certification requirements."

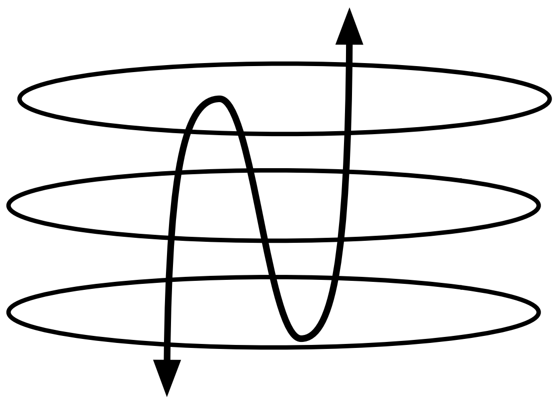

Figure 8–3: Cross-Layer Traveling of Requirements

Coordinating development between new HW, FW, and the supporting SW was a major challenge; the resulting complex interdependencies affected requirements at each of these layers. A change in a processor, or the data stored at the HW level necessitated a change in FW that would almost certainly affect the SW layer. However, these dependencies were cyclical: one manager explained, "for FW to complete, [it's] dependent on SW; for SW to complete [it's] dependent on FW." No matter which was completed first, the additional rework was frequently assigned to different teams, possibly in different sprints or even different release cycles. Given this cross-layer inter-dependence, one manager speculated GridCo might "be better off with cross-functional teams."

For the product development group, reliance on the central information system as well as coordinating documents and standards was essential, but insufficient. For example, at one point a requirements change was made in the C&C SW that required assumptions underlying SW at an intermediate network device to be challenged. The original intermediate SW had been written as an application, and after the requirements changed at a different level, it needed to be rewritten as a daemon.[6]

Dependencies were tracked in the central IS, but there was insufficient assignment of responsibility for cross-layer coordination. One participant indicated—in one of the rare times any participant was openly critical of the company—a major frustration with the lack of coordinating project management across hardware, firmware, and software layers, in that there was no orchestration of the critical path between them. This participant requested special care when making these statements, so as not to be seen as attacking any particular individual, indicating, "I've said enough to get me into trouble."

The coordination of HW, FW, and SW was a constant frustration. Development of the SW layer required access to HW. Due to the mismatch between HW, FW, and SW development schedules, the necessary HW was not scheduled to be delivered to the SW team until late in the cycle. The SW managers' preferred approach was to have a small number of teams work for a longer period building domain knowledge, but volatility in the HW and FW schedules necessitated utilizing more teams over fewer sprints. This created an additional whiplash effect, as the addition of development teams meant not only less productive teams (due to lack of focus and domain knowledge), but also that either teams shared HW prototypes, resulting in slowdowns, or teams being delayed

[6] In this case, "application" means a windowed program or executable with the potential for user interaction. Conversely, a "daemon", also called a "service" on some operating systems, is a program that runs in the background without interaction from the user.

further waiting for their own hardware. The situation was further complicated by multiple HW iterations during the SW development cycle: to properly certify software for use by customers, it needed to be written and tested against production versions of hardware identical to what customers would receive. In another instance, requirements were changed after they were decomposed to accommodate the unavailability of HW.

Even with the use of documented standards as a vertical coordination mechanism, one firmware manager indicated that when the supporting SW is written before the FW is official completed, it is typically necessary to go back and redo the FW to compensate for instances when the documented interfaces were unclear.

“We have this issue every release.”

During sprint 6 (about one month after NPD-1), development fell sharply behind schedule. During a heated status meeting, a manager indicated that dependencies in the FW (complexity uncertainty) necessitated additional resources be allocated to development. Hardware resources that were expected to be available were announced as delayed until at least 60% through the release cycle. A development manager explained in the meeting that if the necessary HW was incrementally delivered from sprints 9 through 12, as anticipated, the necessary SW development work could be accomplished, but there would be no opportunity to find or correct potential critical defects. He continued, “I estimate about 80% confidence of full [development completion] by [NPD-7] if firmware and [hardware] is complete by sprint 9. If that slips one sprint, it’s closer to 50% confidence.” (These resources were not actually delivered until after development on the cycle was complete; work on the dependent requirement sets was done in a minor release, out of cycle.) A different manager indicated that the release date of a hardware unit that interacted with the C&C software was delayed, and the release date was unpredictable, as the HW design had undergone a change from one processor family to another, and that consequently the necessary work was "unknowable" and "impossible to estimate." At this point in the project, all managers reported the release as high risk of falling behind, due to schedule volatility of dependent HW and FW components. Another manager commented, "We have this issue every release."

To further complicate matters, the volatility from cross-layer dependencies occurred at the same time as some significant identity concerns in an unrelated requirements group. A product manager exclaimed, "if we didn't have hardware pressure, [the other requirements] wouldn't be risky," indicating the cycle could absorb some uncertainty, but was struggling to handle multiple uncertainties with big potential schedule impacts simultaneously.

8.5 Cross-Cycle Traveling

Perhaps the biggest advantage of developing recurrent software is cross-cycle traveling of requirements (Figure 8–4). Future requirements thought to be highly uncertain were initially introduced as preliminary investigative requirements. The purpose of these investigative requirements was to identify which parts of the future implementation were uncertain, in order to resolve as much uncertainty as possible in future releases.

The requirements list for the release cycle was initially oversubscribed. However, the promise of future releases allowed a low-impact way of delaying implementation of requirements for something of a higher immediate priority. When implementation of HW-dependent requirements was delayed beyond the release date, other requirements slated for a future cycle were moved forward, with little loss in efficiency.

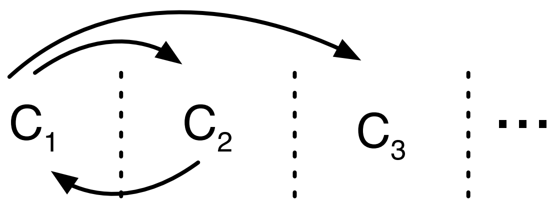

Figure 8–4: Cross-Cycle Traveling of Requirements

Recurrent release cycles also allowed for some (not customer-facing) development to be only partially finished in a release, with the promise of completion in a future release.

In sprint 6, due to a confluence of factors—major requirements groups from a particular customer were delayed due to identity uncertainty, and development to support new FW and HW were delayed for volatility reasons—development managers indicated during a status meeting that not enough requirements had been decomposed to provide sufficient work for all development teams. As a consequence, some development teams were tasked with complementing work outside the cycle. “I have four [vendor] teams with nothing to do that are currently working defects,” one manager noted. Capacity was not being fully utilized, and was consequently being lost as developers were idle or doing low-priority work. The problem was exacerbated a few sprints later when the major identity uncertainties with regard to a strategic customer were clarified, and scope of included items decreased.

To address this unutilized capacity, requirements related to software localization that had been investigated and prepared in a previous cycle—but had been withheld from the observed release for capacity reasons—were added to the observed cycle. The localizations were necessary for future strategic goals of the company, but had not initially been included in the observed cycle, ranking below the line of available capacity. However, due to the unexpected increase in immediate capacity, the localization requirements were added. Had these requirements not been investigated and decomposed in a previous cycle; and, had managers not had familiarity with the status of localization work from previous cycles, these requirements would not have been added to the release. A further benefit to adopting the localization requirements to the current cycle was that its scope was variable: localization work had low uncertainty and could be partially completed as capacity allowed with no noticeable effects to the user, and then fully completed in a future cycle. This flexibility permitted managers to use this requirement as a buffer to fill in work as space was available. In this manner, much of the localization effort slated for a future release was accomplished in the observed release.

As an additional but minor example, a developer reading a story concluded the described end state was unintuitive for the user (as it relied on the user remembering a number rather than a name, and search functionality was not available). The developer reached out to the architecture team suggesting that intuitive naming and search functionality be included. While waiting for a response, another team implemented the story, however, the developer's suggestion was included as a requirement for a future release by the product manager.

The ability to move requirements across cycles was an important tool for the release cycle manager, not just in managing the scope of the release, but in adapting to volatility from revealed uncertainties in development. As one participant described, “Some features [related to a particular meter] were growing too much … to support some functionality we didn’t need until next release. … As the teams were working [they] kept learning more.” Consequently, a portion of the requirements (comprising an estimated 600 person-days of work when HW, FW and SW layers were considered) was moved to the next release. The manager continued, “It was moved because more firmware resources were needed and firmware was strapped. Everything is strapped [for time].”

Lastly, in addition to the forward traveling of requirements through time, the recurrent nature of development permitted anticipatory work on requirements, even before the requirements had been accepted and specified. The analysts understood their time was a bottleneck to the development organization (as made starkly clear in the incident discussed previously). Consequently, analysts and engineers relied on their experience and knowledge-centric position within the organization to anticipate and pre-work selected requirements. As one engineer said, “I knew there wouldn’t be enough time in a release for design … [so] I’ll do up-front design for the most difficult things.”

9 DISCUSSION

Using GridCo as a reference, we relate our empirical analyses to theories of traveling, work design, and recurrent packaged software. To conclude our engaged scholarship, we discuss strengths and weaknesses of GridCo's development to explicate contributions and limitations of the study.

9.1 Traveling

Understanding how requirements travel in a particular organization permits researchers a way to understand the strengths and weaknesses of the organization structure surrounding product development, and how the structure may be best adapted to address uncertainty. The three types of traveling revealed in the analysis (local, cross-layer, and cross-cycle) gave insights into the workings of a complex software organization, as its members worked to resolve task uncertainty in the recurrent release of packaged software for electric grid management.

Figure 8–1 (Work Design) maps the typical travel of requirements across the organization. The traveling constructs described in the analysis framework are embedded in the display: each activity constructs or translates the requirement, and requirements are shared between organizations and with the common information system. At GridCo, the theorized differences between construction and translation did not appear significant. It may be that translation, as defined in this dissertation, is simply another form of construction, and that a broader definition of translation would be more useful, such as the one by Nielsen et al. (2013), which describe translation as transformation and movement as actors apply their knowledge to practical use. Or, it may be that the management-centric level of analysis of this dissertation was at too high of an organizational level to capture sufficient data about translation.

The strong use of an IS as requirements repository as a canonical source of truth may have also limited the “translation” of requirements, in its original meaning by Czarniawska and Joerges (1996). Except for individual construction work (decomposition and writing computer code), most of the sharing of requirements and construction effort (e.g., estimating and status updates) occurred in groups that crossed organizational boundaries, so a group’s consistent understanding of requirements may have also overshadowed possible changes due to translation. Certainly, the prolific use of a common IS served to minimize variation in individuals’ understanding of requirements.

As requirements traveled through the organization, uncertainties were resolved at each step. Identity uncertainties tended to be resolved earlier in the requirement’s lifecycle; complexity uncertainties were, by their nature, encountered later. Some identity uncertainties began with customer interaction or negotiated contracts items (for example, the requirement to support a particular meter, with no definition of “support”). Other uncertainties resulted from misinterpretation of customer intent. In both cases, these identity uncertainties were resolved by backtracking through the development process, sometimes resulting in consultation or negotiation with customers.

In our finding of localized traveling, the distinction between creep and expansion (or removal and trimming) is centered on the type of uncertainty being resolved; the former concerns volatility, while the latter addresses identity. This distinction may at times be murky for a couple of reasons. First, uncertainty due to complexity may reveal identity uncertainty and lead to volatility as well. Second, the observation may change with the unit of analysis: requirements at the smallest level of work may indicate creep or removal, but when considered as a full requirement or requirement group, this may present as expansion or trimming.

Temporal considerations also have a part in how GridCo managed uncertainty in requirements. Early in the project, prior to NPD-2, uncertainty was embraced with rough estimates (T-shirt sizes) as the requirement was shared horizontally across the organization to focus a shared vision, and requirements were added and removed from scope, causing a great deal of volatility in the early part of the cycle. During the middle of the project, horizontal coordination was insufficient to constrain the oversubscription of cycle capacity, and guidance was sought along vertical lines. Following NPD-2, process structures applied vertical reinforcement to travel paths as formal change control (and consequently, hierarchical approval) became required. Based on observation, this progression from “loose horizontal” to “strict vertical” over the life of the project was consistent with other releases. As one manager said, “early [we] embrace uncertainty, after, we want to restrict uncertainty and control it.”

Although not wholly related to the traveling of requirements, the structure of development on maintenance—use of a separate “sustaining team” external to the release—issues may limit the effects of double-loop learning (Argyris and Schon 1978; Nerur and Balijepally 2007) predicted by theory. GridCo separates a cadre of four development teams on a six-month rotation focused on maintenance issues. This has the advantage of providing a more predictive level of staffing for project issues, and creates a buffer of resources that in extreme circumstances were sometimes reallocated between maintenance and work on the release, but at the potential cost of more real-time learning at the team level.

9.2 Work Design

Our analyses of how requirements travelled at GridCo revealed interesting insights into how structures, processes, systems and knowledge impacted the ability to manage uncertainties in observed release cycle.

9.2.1 Structures, processes and systems

The discussion of work design (Chapter 5) focused primarily on horizontal (modular) and vertical (hierarchical) boundaries in work design, and, following our understanding of Sinha and Van de Ven (2005), tip-toed around addressing network problems, simply noting they were the complex, entangled interaction of modular and hierarchical concerns. Yet, in our observation, applying only horizontal and vertical descriptions was insufficient to capture the richness of interaction: nearly every interaction could be described as network.

Overall, GridCo used a primarily modular, functional organization design. Each of the functions involved in the release cycle also had responsibilities for other products, projects, and releases. Yet, the organization worked, and worked well. Against this basic empirical finding, we may relate our analyses at GridCo to the core literature. Galbraith (1973) assumes coordination occurs across vertical boundaries. Instead, at GridCo work moved back and forth across functional boundaries at the same horizontal level, requiring coordination of all parties. Sinha and Van de Ven (2005) describe network boundaries in a way that at first blush implies the quantity of connections is of primary importance, whereas this case exemplifies it is rather the necessity of coordination between a multiplicity of participants (each, potentially, with a cross-functional role) that exemplifies a network.

Mintzberg (1993) summarized Galbraith (1973) and other organizational researchers, and explained the continuum of "liaison devices" organizations adopt to overcome the deficiencies of purely functional or purely hierarchical organizational designs. In particular, GirdCo demonstrated the middle two types of devices on the continuum from simplest to most elaborate: standing committees and integrating managers. (Liaison positions and a full matrix structure begin and end the list, respectively.)

Perhaps the most continuously effective coordination mechanism at GridCo was the "project status meetings," attended by managers and directors across related functional silos. Although the status meetings were technically a task force for the release cycle, cycles for the C&C software overlapped, so the meeting would transition from status of a soon-to-finish release to status of a just-beginning release quickly, with no change in personnel. Status meetings were the primary mechanisms for cross-cycle traveling; decisions regarding whether to move requirements forward or backward were made and negotiated among this group of people.

Other meetings, including architecture and design meetings, evolved to included a subset of status meeting attendees. These design meetings negotiated some of the complex architectural uncertainties arising from dependencies across layers. The design meetings also adapted to take over the rough estimation function that in the observed cycle was accomplished through a plenary meeting, and thus accomplishing the same task piecemeal and as needed over a period of weeks, rather than interrupting many workers for an extended meeting.

The second organizational device, integrating managers, is “a liaison position with formal authority” (Mintzberg 1993, p. 83). At Gridco, both the release cycle manager and product portfolio manager had weak positional authority, and as Mintzberg (1993) predicts, primarily exerted influence by negotiation and persuasion of those over whom there was no formal authority. Some were more effective than others in this role. As was the case at GridCo, “The effective integrating manager appears to require a high need for affiliation and an ability to stand between conflicting groups and gain the acceptance of both without being absorbed into either” (Mintzberg 1993, p. 84).

In combination, integrating managers and standing committees created relationships that presented as a “local hub assembly” (Figure 9–1)[7], a complete network of connections of all stakeholders involved in the C&C release at the manager and director levels. The assembly, a group gathered for a common purpose, much like a standing committee or task force, was local to the release as well as to the parent product development organization. Additionally, the assembly acted like a hub to the greater product organization, in fashion similar to hub firms as described by Dhanaraj

[7] Figure 9–1, while illustrative, is limited by two dimensions. Imagine instead five different functions, each with a couple of specific specialists with a relevant stake in the release converging in a fully connected graph.

and Parkhe (2006). Orchestration of the release cycle was accomplished by the assembly as it “pulls together and leverages the dispersed resources and capabilities of network members” (Levéna et al. 2014, p. 158), under the direction of a process orchestrator. The process orchestrator leveraged an organizational reliance on a strong, central IS, as well as a regular structure of meetings and formal interaction opportunities to regularly align the focus of the assembly. Occasionally, vertical directives were received by one stakeholder, and had to be processed by the group. Conversely, there were occasional problems that were passed up some—but usually not all—of the vertical reporting lines of stakeholders. In this way, the “local hub assembly” formed the heart of the coordination of the release cycle.

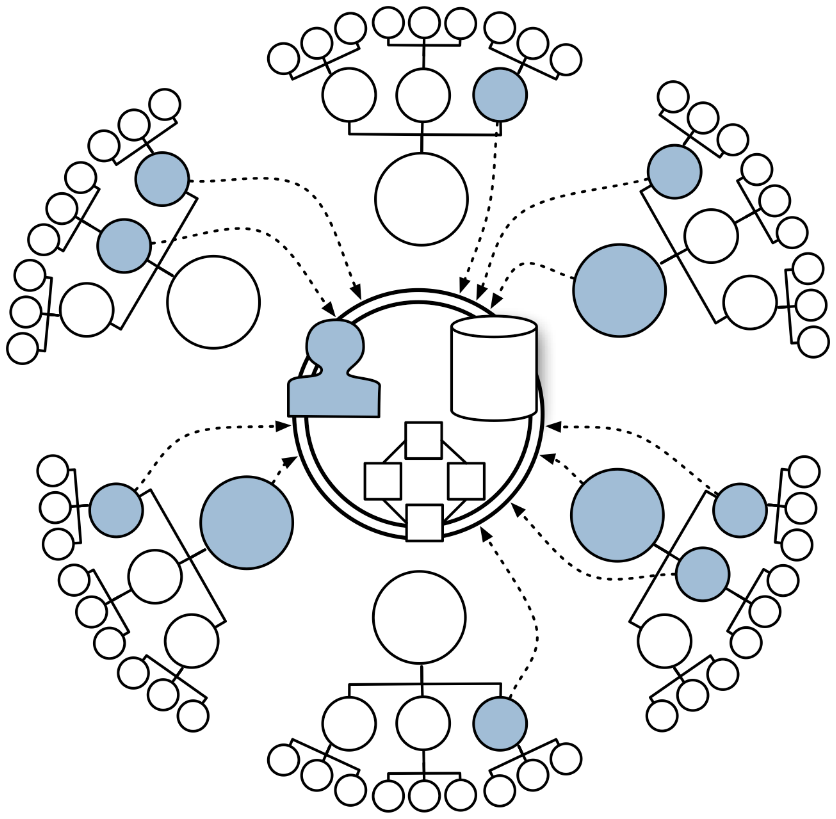

Figure 9–1: Coordination Through the Local Hub Assembly at GridCo

Here the duality between software and organization is again manifest. Uncertainty is a function of work design while at the same time also conditions and informs work design. The lower uncertainty (conditions of high programmability), such as what existed at the lower, operational levels of the organization due to the detailed deconstructions accomplished early in the process, permitted a more modular structure. Situations of higher task uncertainty (low programmability) required, more organic, tightly coupled structure in order to be responsive.

Other patterns described by Mintzberg (1993) were facially evident, including the way GridCo used specific roles as knowledge leaders, and as a consequence created slack resources (Galbraith 1973) while seeking standardized skills in a professional core that worked independently from colleagues (Mintzberg 1993). Reinforcement of processes and process orchestrators, what Mintzberg (1993) refers to as “technostructure” were also strongly evident, as GridCo is a very process-driven organization. The release cycle manager adopted this role.

Lastly, as has been described, there was a clear investment in vertical information systems, which is predicted by Galbraith (1973) to reduce uncertainty by increasing information processing capacity of the organization, a notion validated by multiple information systems researchers. However, despite a vertical information system, much of the data it provided was duplicated in multiple ways during coordination activities such as status meetings. For example, status summaries stored in the system were often read aloud during coordination meetings. Hence, a key function of the IS at GridCo was to support sharing of existing documentation and as a constant reminder to all participants what had been achieved and what plans had been committed to. The shared requirements IS was used collaboratively in requirements estimation, as a meeting guide for acceptance and review of completed requirements. Output from the vertical IS also formed the bulk of status summary documents in a manner similar to a balanced scorecard. In short, shared information systems were core to nearly every activity in product development at GridCo.

Overall, it was the combination of and interactions between the release assembly the process orchestrator that took center stage in dynamically organizing the release cycle so the participants could successfully manage uncertainties. Backstage, these mechanisms were enabled by Gridco's established organizational structure, its vast repertoire of processes, templates and standards, and, its extensive use of a comprehensive and extensively shared IS.

9.2.2 Knowledge Centers

Requirements construction and architectural decisions at GridCo relied on analysts and systems engineers, who worked in the same functional silo. As a whole, these were experienced employees with extensive domain and product knowledge. As they had primary responsibility for the deconstruction of all requirements in the queue, the oversubscription of the release placed a great deal of strain on this function, and eventually caused a bottleneck in the release.

This was a structural response to uncertainty, and had the benefit of permitting development resources to be more fungible and scale more readily. Requirements were deconstructed to the extent that even the newest development teams could accomplish them; a necessary feature, for one development manager estimated two-to-three-year ramp-up time for new developers to acquire enough domain knowledge to have full productivity. Unfortunately, the centralized group of analysts and engineers with the requisite domain knowledge could not scale as quickly. As one manager said, “We need more systems engineers.”

However, as the primary knowledge center for product development, this group unintentionally acted as a bottleneck to other decisions. Participants explained that any architectural or design decision made in the development organization required a second meeting to get the buy-in of these experts.

9.3 Recurrent Development of Packaged Software

Perhaps the most interesting finding specific to packaged software is the temporal traveling of requirements across cycles, a key tool used by GridCo. This suggests some of the observations of Sawyer (2000) are due not solely to the type of software being developed, but also due to the recurrent nature of development. This may help bring clarity to the muddle of definitions and distinctions between classifications of product software (Xu and Brinkkemper 2007). Cross-cycle traveling also enabled reconsideration in the following release of whether the pushed requirements were as important as initially indicated, and thus provided an additional filter useful in identifying the most important requirements. Requirements were not the only thing to travel, however. In multiple instances, resources were temporarily shifted between overlapping cycles.

One major benefit of cross-cycle traveling, was the minimal disruption caused by introducing a requirement from a previous cycle. Typically, much of the work of uncertainty management, through activities such as deconstruction, had already been accomplished; previously expended resources were not wasted, and there was no noticeable loss in momentum.

A second benefit of embracing cross-cycle traveling was the shifting of requirements with low-uncertainty forward and backward between cycles as a buffer or hedge against uncertainty. When important but highly volatile work became available, some low uncertainty requirements were shifted to the next cycle. Conversely, when high volatility unexpectedly opened capacity, low uncertainty requirements, in this case, international localizations prepared in a previous release and scheduled for a future release, were shifted to the current cycle so available capacity was not wasted.

Benefits of recurrent development extended beyond cross-cycle traveling. During a particular observation, senior developers were estimating story points for software features necessary for support of a new utility meter in the command and control software. Estimation of these stories occurred with surprisingly little discussion. Additionally, some stories were seemingly duplicated. On investigating further, it became clear that the stories being estimated were similar for all meters supported by the software, and that support for new meters was added regularly. Thus, iterating cycles not only increase domain knowledge, but may expose repeated patterns of functionality to be implemented in similar ways, improving productivity not only in the development, but also in the planning and coordination of development.

The shared experience of multiple iterations led, as expected, to a shared vocabulary at GridCo that took some time for researchers to understand. This increased organizational learning (Lyytinen and Rose 2006) and consequent clearer communication, likely, over time, resulted in reduced uncertainty due to translation as requirements traveled.

As predicted, cross-cycle traveling provides benefits beyond managing the uncertainty of a particular requirement or release. Some requirements occurred in multiple release cycles. Some requirements persisted "below the line" in multiple releases, but not prioritized high enough to be developed in a particular release. GridCo utilized investigative requirements as a tool to uncover uncertainties in developing a future requirement. Such investigative requirements did not necessarily introduce functionality in the then current release, but were fully realized in an initial release, but is fully realized in a future one. These are examples of traveling that would not be possible if development were recurrent, as it is in packaged software.

9.4 Engaged Scholarship

In return for site and data access, we agreed (Section 7.1) as part of our engaged scholarship effort (Van de Ven 2007) to return to GridCo with a practical evaluation of their organization and processes. The evaluation, summarized below, highlighted strengths and difficulties observed during data collection, and presented options for possible improvement.

9.4.1 Strengths

We saw GridCo as a mature organization that successfully coordinated across multiple sites thanks to a strong, mature process culture. The firm as a whole regularly managed hundreds, if not thousands of HW and SW projects. They had a history of successful releases, which spoke to their ability to repeat successes. This repeatability (Humphrey 1989; Lyytinen and Rose 2006) was reinforced by adherence to firm-mandated NPD-gates, combined with adaptive agile-like developments processes. This structure improved coordination with the rest of the firm, beyond the stakeholders in the release cycle as well as providing a structural short-term vision for the release (Parnas and Clements 1986). Yet, despite the rigid structure, the organization remained adaptive, and modified its processes to improve flow of information processing, as evidenced by the change in how rough estimation was accomplished. This ambidextrous balance of discipline and adaptability also provided performance management and social support to actors within the organization (Gibson and Birkinshaw 2004; Napier et al. 2011). Regular status meetings held functions accountable for their work. There was a communal and sometimes negotiated understanding of expectations, and feedback when expectations were not met, but also a willingness for parts of the organization to compensate when work by another stakeholder was insufficient, as it occurred when development utilized capacity that would otherwise have been wasted when insufficient requirements had been decomposed.

The role of the process orchestrator and the local hub assembly in collectively

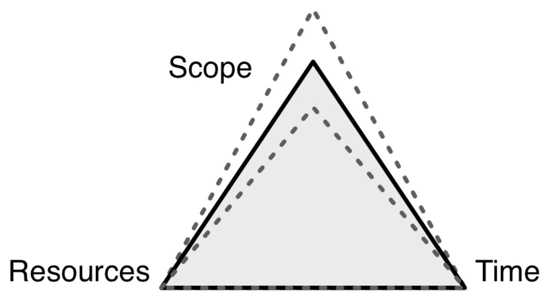

Figure 9–2: Manipulating the Triple Constraints of Project Management

coordinating the project was instrumental to the organization's success. There was regular effort in status meetings to agree on both unified internal and external messaging to maintain an aligned vision of the release. The organization embraced coordination cost as a necessity, and continuously reinforced and rewarded process behavior that was productive, not just formal. For example, although status documents were available via a shared information system, weekly external messaging to the firm hierarchy was read aloud. At first, we wondered whether this behavior was superfluous, but came to recognize that this over-communication was essential to building the almost consistently unanimous consensus of the assembly.

One of the greatest strengths of GridCo was its acceptance of the constraints of project management. The organization understood the trade-offs inherent in the triple-constraints of the "Iron Triangle" (scope, resources and time) (Kapur 2004), and stakeholders had support of their hierarchical leadership when making adjustments of scope and resources to meet vertical demands. The C&C release typically manipulated scope, as resources (e.g., personnel) and time were generally fixed (Figure 9–2).

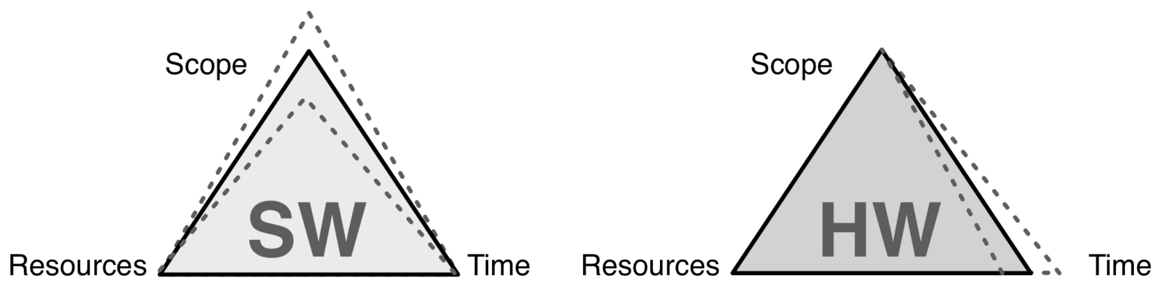

Figure 9–3: Differences in Constraint Management in Software and Hardware

9.4.2 Difficulties

By far the most pressing challenge GridCo faced was the management of cross-layer dependencies. HW and SW projects were supposedly timed to publicly release simultaneously, but internal schedules prevented this during the observed period. In development of SW, recurrent releases and the (comparatively) low cost of deployments and upgrades permitted release managers to manipulate scope by easily moving requirements to future cycles. HW development was not so lucky; their scope and resources were static by comparison, and consequently HW projects adjusted their release date as a response to encountered uncertainties. SW, on the other hand, was under a strict schedule by vertical fiat, and adjusted scope as necessary to meet the schedule (Figure 9–3) As a development manager indicated, “We’re agile in requirements, but not agile in schedules.” In short, the scope flexibility of SW was incompatible with time flexibility of HW. Managers responsible for the C&C release attempted repeatedly in previous cycles to impose hard limits on completion status of HW components included in the cycle’s scope, but were regularly overruled by executives determined to keep contractual commitments. Thus, cross-layer traveling of requirements dependent on HW and FW components introduced a great deal of schedule uncertainty in the observed release. Major HW components were eventually delivered so late that SW support, a major component of the release, was delayed and released as a separate update for a specific customer two months after the cycle was to have concluded.

Incomplete management of backlog requirements between release cycles also posed a difficulty for GridCo. Both strategic planning and drafts of backlog lists for each cycle were made more difficult by the lack of a unified backlog that persisted across cycles. Product and product area managers maintained separate backlogs that were combined at the start of each cycle, as one product manager described it, through “Darwinian” negotiation. Consequently, a manager noted, “There's very little of what I would call a true portfolio review in product management, except on a very ad hoc basis, per release, as things are just about in front of us, to say, okay, here's how these things are going to interrelate to produce a more cohesive product backlog.” Consequently, rather than a reserve backlog providing a organization-wide roadmap of future development, the backlog was made more volatile by frequent executive escalation of priorities in the early stage of the observed cycle. Cross-layer dependencies compounded the difficulties that arose from lack of a backlog. Multiple participants shared the view that a lack a portfolio view of requirements to examine relationships between HW and SW releases contributed to increased executive escalation of requirements early in cycles. Further, this lack of a backlog limited the ability of the product development group to communicate its plans to external entities within the company (e.g., sales). Each SW cycle began significantly over-subscribed; a backlog of uncompleted requirements is a sign of a healthy product, but an over-full backlog might indicate a need for additional resources.

These difficulties combine to manifest a third area of trouble for GridCo: product strategizing. Without a between-cycle backlog, there was no default way to communicate a long-term product roadmap with the rest of the firm. Instead, releases tended to experience increased volatility due to contractual demands from new customers through sales. GridCo was very effective in sales, and entered several significant strategic contracts before and during the observation period, but members of the product development organization expressed concern that they lacked a long-term vision, such as might be expressed in a shared, unified backlog. Consequently, strategic resource planning was also hampered because it is difficult to plan for future resources without understanding future scope. As one example, GridCo used an outsourced software development vendor. The ramp-up time and quality of delivered code were somewhat below expectations, and development managers wanted to move that capacity to in-house and overseas captives. As one manager summarized, "They haven't worked out like we wanted". Use of the vendor's development resources was initially slated to ramp down over a twelve-month period in a previous cycle, but demands of scope necessitated those resources be fully utilized. Without a backlog to inform the discussion, product development seemed to avoid addressing strategic ways of adjusting capacity (up or down) to properly fulfill demands of scope.

9.4.3 Options

Elucidating the difficulties encountered by product development at GridCo makes some avenues for potential improvement very clear. Strategic resource and scope management would be improved through a unified product backlog that is shared beyond product development and used as a basis for strategic resource planning. In addition, the release cycle was quite long. Consequently, some participants had a tendency to want to push development of requirements later in the cycle. In contrast, shorter cycles would beneficially narrow the solution space for managing uncertainty by constraining the schedule; and, requirements would be addressed or moved to a future cycle for later consideration, rather than being reconsidered multiple times (and causing thrashing in the project). Customers would likely not react well to shorter release cycles (their independent certification requirements were typically about several months), but shorter, interim releases could be considered only internally.

One method of gaining the benefits of shorter release cycles is a concept discussed by (McConnell 1998, p. 38), called “Two-Phase Funding.” Although McConnell (1998) describes two-phase funding from a financial perspective, the same principles apply when resources are fixed, and scope is being manipulated to match capacity. As a way to reduce variation, the first portion of the cycle is dedicated to the most uncertain requirements with the primary intent of reducing their uncertainty in the latter part of the cycle. This structural change might involve separate change control windows for the early and late parts of the cycle, and could include formal deadlines and completion standards for consideration of inclusion of requirements dependent on new HW, thus also reducing the need to manage the volatile interactions of cross-layer traveling. Early and late cycle windows would mean more frequent, but shorter ranking discussions, and these discussions would be simpler due to the shorter time frames involved. Similarly, such a change rewards discipline on low-uncertainty requirements, and provides for adaptability on high-uncertainty requirements. Discussions of volatility would be less common, leading to less thrashing during coordination. Lastly, the additional deadlines inherent in a double window cycle might lead to small productivity boosts due to deadline effects.

The final option presented to GridCo for consideration was allowing for a flexible release window. The high volatility of HW-dependent requirements almost guaranteed delays that were not apparent until well into the cycle. Time frames were fixed because they had been communicated outside the firm; protecting those releases date and instead providing release windows until schedules were less uncertain (for example, at NPD-2) would provide an additional option for managing uncertainty.

9.5 Contributions

It is evident from even casual observation that GridCo is a very process-mature organization. Actors at multiple levels rigorously document, adhere to, and, reinforce its engineering and management processes. The release manager of the studied release cycle claims a strong track record of successful releases (on time, full scope, within budget). In addition, the market seems to be responding to the success of the organization, as evidenced by GridCo winning contracts from increasingly large customers (and correspondingly increasing revenue) over the past year. This has lead to a very rapid growth in their development organization, maybe due to a successful handling of uncertainty in software development; consequently, such rapid growth may also reveal uncertainties. In any case, rapid growth affords a future opportunity to investigate structural responses to uncertainty.

The primary findings of this dissertation speak directly to the initial research questions. Local, cross-layer, and cross-cycle traveling of requirements are organizational responses to managing uncertainty. As such, this dissertation contributes to the under-represented requirements research in IS (Hassan and Mathiassen Forthcoming) by exploring requirements practices in a complex software development organization. Further, the notion of traveling (Czarniawska and Joerges 1996; Nielsen et al. 2013) is further validated as an analysis tool for researchers in IS.

Software development research as a whole benefits from research that considers both uncertainty and work design (King 2013), and this dissertation answers recent calls for modern work design research (Sinha and Van de Ven 2005). Software development is accomplished in a wide range of organizational structures, and the discussion of network hub assemblies contributes to field understanding of variations present in firms. The traveling metaphor (Czarniawska and Joerges 1996) and the three types of traveling observed at GridCo reveal new insights into management of uncertainties in development practices.

Software organizations that must manage multiple product layers benefit from understanding cross-layer traveling and the resulting whiplash effect of requirement dependencies and volatility. Even though GridCo employs risk models to provide management some confidence in cost–benefit and risk–rewards analyses, their models may require revision. Complex multi-layer software projects, such as GridCo's centralized C&C system, rely on HW (and its associated FW) being completed to a sufficient level before software development can begin. This means the work on later software requirements—which may reside in a different release cycle than the HW component—is subject to not only to its standard risk variance, but also to the sum of all risks of the dependent projects. At GridCo, project and functional boundaries within the organization were reinforced by release-focused processes, which have led some within the organization to call for a more holistic management of the project portfolio, and exploration of ways to span these boundaries with a more pragmatic approach

Development of packaged software (Xu and Brinkkemper 2007) is validated as distinct from development of other software, in that enables additional methods of managing uncertainty, as explained in cross-cycle traveling. The review of packaged software literature (Chapter 3) highlighted some inconsistencies in how packaged software is viewed and classified. The GridCo narrative is a useful data point in bringing order to this emerging domain.

Uncertainties with regard to requirements, as exemplified by identity, volatility and complexity uncertainties (Mathiassen et al. 2007), were evident throughout the release cycle. Implicated uncertainties will change between identity, volatility and complexity as requirements travel. The types of local traveling, as well as the different perspectives introduced by cross-layer and cross-cycle traveling, allow insight into what sorts of uncertainty might be expected, and a description of how those uncertainties were handled at GridCo. The organization's processes serve to enable and reinforce coordination through structural boundary spanning both through the local hub assembly of primary stakeholders and their respective functional hierarchies. However, at times it also impedes the success of the organization as some of these processes are ill-adapted to the uncertain nature of software, resulting in anomalies such as *ex post facto* approval of changes to project scope, schedule and budget, or the beginning of software development before requirements are accepted into a release. However, these events where structure is ill-fitting may represent acceptable costs when compared with the added complexity of utilizing different processes for different project types within the same organization. This question is echoed by Child (1977, p. 175), who asks whether an organization should "set a limit on its internal formalization in order to remain adaptable, or should it allow this to rise as a means of coping administratively with the internal complexity that tends to accompany large scale?".

Grounded in findings such as these, this research contributes to the IS requirements management literature and the packaged software literature. As described in Chapter 4, there is not a strong tradition of requirements-related research within the IS discipline (Hassan and Mathiassen Forthcoming). Treating requirements as expressions of uncertainty provides a connection to related fields of research in IS. Examining requirements within the context of an organization led to uncovering new knowledge about the sources of and responses to uncertainty in development of software, contributing to both the IS and software development literature. Further, by examining requirements i*n situ*, this dissertation provided insight into software development and contributes a modern narrative to existing knowledge of general practices.

GridCo is a very different type of organization than the cases considered in Sawyer (2000). Thus, this case may be useful in extricating industry and organization effects from effects contingent on whether development is of packaged software or custom development. When considered point-by-point, practices at GridCo may be analyzed and presented as evidence or contradiction of Sawyer's (2000) speculations. Many of the descriptions and effects of packaged software (Sawyer 2000; Xu and Brinkkemper 2007) have not yet been subjected to empirical analysis (Light and Sawyer 2007), so this dissertation is a novel entry in that regard.

Finally, consistent with the responsibilities laid out in the MoU, researchers contributed to practice at GridCo by providing theory-informed summaries of recommendations to key stakeholders at the research site (as described in Section 9.4).

9.6 Limitations

Any research is subject to limitations of scope and method. This research draws on a single case (Miles and Huberman 1994; Yin 2009), which limits the viability of cross-case comparison or generalization of findings to other contexts (Lee and Baskerville 2003). Researchers and practitioners in software development research, seem particularly likely to overreach in claims of applicability to other contexts, without consideration for differences between the contexts in the development method, organizational dynamics, or type of product (Jackson 1995). However, these disadvantages are weighed against the strengths of single-case research: attention to contextual dynamics and integration of multiple perspectives resulting in rich description (Mason 2007). Detailed description and rigorous analysis may enable future researchers to confirm and expand these findings in other contexts. To ensure rigor and validity, standard practices of empirical qualitative research were adopted (Miles and Huberman 1994; Yin 2009).

Single-case studies do not provide as strong a basis for theory building as multiple cases might (Yin 2009), yet single cases permit a richer description of observed phenomena, which can, in turn, lead to strong theory regarding the research setting. Although organization-level effects are difficult to generalize to other contexts, this is not the intent of single case research (Siggelkow 2007). Conversely, observations of multiple actors or artifacts within a consistent context permits investigation of the behavior and attributes of these subjects with a stronger claim that contrasting effects are not context-dependent.

Generalization to populations or other contexts is not the intent or purpose of interpretive research. Instead, interpretive research seeks to generalize descriptions within a setting, and from there, generalize to theory (Lee and Baskerville 2003). This is not a weakness of case study research, but rather a strength (Eisenhardt and Graebner 2007; Lee and Baskerville 2003).

Coding by a single researcher, as was done for this dissertation, is common in interpretive studies (Cousins and Robey 2005; Schultze 2000), although it sometimes raises concerns of researcher bias. However, as Eisenhardt and Graebner (2007, p. 25) noted, "Although sometimes seen as 'subjective,' well-done theory building from cases is surprisingly "objective," because its close adherence to the data keeps researchers 'honest.'" To mitigate researcher bias, the coding scheme was first dialectically iterated to be as objective and clear as possible. Both researchers participated in challenging interpretations in data collection, and conclusions reached through data reduction and data displays. Analyses were iterated to confirm fit between data, theory, and the coding framework. In all, these steps improved reliability of the interpretation and analysis (Miles and Huberman 1994).

Quality interpretive research further protects against claims of bias by triangulating data, using multiple sources and types of data, seeking feedback from key informant on researcher interpretations, and, by iteratively refining their understanding by rigorous immersion in the data (Miles and Huberman 1994; Yin 2009). The multiple sources of data employed in analysis give strength our conclusions Importantly, these findings were presented to key stakeholders at GridCo who concurred with the interpretations of data and key findings.

There is a danger that the findings regarding the traveling of requirements, and the types of traveling present at GridCo, will be applied to other contexts without appropriate verification in those contexts, but that is a problem for future researchers.

9.7 Conclusion

We began by questioning how requirements travel, both socially and structurally within an organization. RQ1 focused on construction, sharing, and translation of requirements, while RQ2 examined traveling from the perspective of organizational structure. At our level of analysis, it was sharing, more than construction or translation, that moved to the forefront. Although the requirements construction life-cycle was detailed in the analysis (e.g., Figure 8–1) and was useful in elucidating local traveling and confirming that requirements do indeed travel and change, the interaction of horizontal and vertical boundaries that informed cross-layer and cross-cycle traveling was of even more interest. Additionally, the organizational structure—the local hub assembly—mitigated the effect of these boundaries.

Both research questions constrained focus to "recurrent software development" which is addressed by cross-cycle traveling. In addition, the recurrent nature of development at GridCo enabled both extensive distributed domain knowledge and an ease of coordination between actors that might have been less likely in other contexts.

The three types of requirements traveling revealed in this dissertation—local, cross-layer, and cross-cycle—form the basis for discussion of organization structure, uncertainty resolution, and packaged software development, and provide real benefit to the field of information systems development. Using requirements as a lens, we have examined novel organizational structures in practice, and compared them to seminal work on organizational contingency theory. We have also validated a modified vocabulary of the traveling metaphor and applied it in IS research. These findings are novel contributions to practice and theory.

10 REFERENCES

Argyris, C., and Schon, D. 1978. *Organizational Learning: A Theory of Action Perspective*. Reading, MA: Addison-Wesley.

Austin, R.D., and Devin, L. 2009. "Research Commentary—Weighing the Benefits and Costs of Flexibility in Making Software: Toward a Contingency Theory of the Determinants of Development Process Design," *Information Systems Research* (20:3), pp. 462-477.

Balmer, S. 2013. "One Microsoft: Company realigns to enable innovation at greater speed, efficiency [Press release]." Retrieved July 12, 2013, from https://http://www.microsoft.com/en-us/news/Press/2013/Jul13/07-11OneMicrosoft.aspx

Banker, R.D., Hu, N., Pavlou, P.A., and Luftman, J. 2011. "CIO reporting structure, strategic positioning, and firm performance," *MIS Quarterly* (35:2), pp. 487-504.

Basili, V.R., and Boehm, B. 2001. "COTS-based systems top 10 list," *Computer* (34:5), pp. 91-95.

Baskerville, R., Pries-Heje, J., and Madsen, S. 2011. "Post-agility: What follows a decade of agility?," *Information and Software Technology* (53:5), pp. 543-555.

Benbasat, I., and Zmud, R.W. 2003. "The Identity Crisis within the IS Discipline: Defining and Communicating the Discipline's Core Properties," *MIS Quarterly* (27:2), pp. 183-194.

Benslimane, Y., Yang, Z., and Bahli, B. 2010. "Requirements uncertainty and standardization in IS development projects: A survey of the IT sector in China," *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*, pp. 1097-1101.

Boehm, B.W. 1991. "Software Risk Management: Principles and Practices," *IEEE Software* (8:1), pp. 32-41.

Brooks, F.P. 1987. "No Silver bullet: Essence and accidents of software engineering," *IEEE Computer* (20:4), pp. 10-19.

Cao, L., and Ramesh, B. 2008. "Agile requirements engineering practices: An empirical study," *IEEE Software* (25:1), pp. 60-67.

Carlile, P.R. 2002. "A Pragmatic View of Knowledge and Boundaries: Boundary Objects in New Product Development," *Organization Science* (13:4), pp. 442–455.

Carlile, P.R. 2004. "Transferring, Translating, and Transforming: An Integrative Framework for Managing Knowledge Across Boundaries," *Organization Science* (15:5), September/October 2004, pp. 555–568.

Carlile, P.R., and Rebentisch, E.S. 2003. "Into the Black Box: The Knowledge Transformation Cycle," *Management Science* (49:9), September 1, 2003, pp. 1180-1195.

Carmel, E., and Becker, S. 1995. "A Process Model for Packaged Software Development," *IEEE Transactions on Engineering Management* (42:1), pp. 50–61.

Carson, S.J., Madhok, A., and Wu, T. 2006. "Uncertainty, Opportunism, and Governance: The effects of volatility and ambiguity on formal and relational contracting," *Academy of Management Journal* (49:5).

Cheng, B.H.C., and Atlee, J.M. 2007. "Research Directions in Requirements Engineering," in: *Future of Software Engineering*. IEEE.

Child, J. 1977. *Organizations: A Guide to Problems and Practice*. New York: Harper & Rowe.

Choudhury, V. 1997. "Strategic Choices in the Development of Interorganizational Information Systems," *Information Systems Research* (8:1).

Cousins, K., and Robey, D. 2005. "Human Agency in a Wireless World: Patterns of Technology Use in Nomadic Computing Environments," *Information and Organization* (15:2), pp. 151–180.

Czarniawska, B. 2009. "Emerging institutions: pyramids or anthills?," *Organization Studies* (30:4), pp. 423-441.

Czarniawska, B., and Joerges, B. 1996. "Travels of Ideas," in *Translating Organizational Change,* B. Czarniawska and G. Sevón (eds.). New York: Walter De Gruyter.

Davidson, E.J. 2002. "Technology Frames and Framing: A Socio-Cognitive Investigation of Requirements Determination," *MIS Quarterly* (26:4), pp. 329-358.

Davis, A. 1993. *Software Requirements: Objects, Functions and States*. Upper Saddle River, NJ: Prentice Hall.

Davis, G.B., Ein-Dor, P., King, W.R., and Torkzadeh, R. 2006. "IT Offshoring: History, prospects and challenges," *Journal of the Association for Information Systems* (7:11), pp. 770–795.

Davison, R.M., Martinsons, M.G., and Kock, N. 2004. "Principles of canonical action research," *Information Systems Journal* (14), pp. 65–86.

Dhanaraj, C., and Parkhe, A. 2006. "Orchestrating Innovation Networks," *Academy of Management Review* (31:3), pp. 659–669.

Downey, H.K., Don, H., and Slocum, J.W., Jr. 1975. "Environmental Uncertainty: The Construct and Its Application," *Administrative Science Quarterly* (20:4), pp. 613-629.

Downey, H.K., and Slocum, J.W. 1975. "Uncertainty: Measures, Research, and Sources of Variation," *The Academy of Management Journal* (18:3), pp. 562-578.

Drazin, R., and Van de Ven, A.H. 1985. "Alternative Forms of Fit in Contingency Theory," *Administrative Science Quarterly* (30:4), pp. 514-539.

Dyreson, C. 1997. "A Bibliography on Uncertainty Management in Information Systems," in *Uncertainty Management in Information Systems,* A. Motro and P. Smets (eds.). Springer US, pp. 413-458.

Eisenhardt, K.M., and Graebner, M.E. 2007. "Theory Building from Cases: Opportunities and Challenges," *Academy of Management Journal* (50:1), pp. 25–32.

Galbraith, J. 1973. *Designing Complex Organizations*. Assidon-Wesley.

Gibson, C.B., and Birkinshaw, J. 2004. "The Antecedents, Consequences, and Mediating Role of Oranizational Ambidexterity," *Academy of Management Journal* (47:2), pp. 209–226.

Graham, I. 1998. *Requirements Engineering and Rapid Development*. Harlow, UK: Addison-Wesley.

Harris, M.L., Hevner, A.R., and Collins, R.W. 2009. "Controls in Flexible Software Development," *Communications of the Association for Information Systems* (24), pp. 757–776.

Hassan, N.R., and Mathiassen, L. Forthcoming. "Distilling Information Systems Knowledge: Canonical Knowledge Areas for Information Systems Development," *Submitted for publication*).

Henderson, R.M., and Clark, K.G. 1990. "Architectural Innovation: Reconfiguration of existing product technologies and the failure of established firms," *Administrative Science Quarterly* (35:1), pp. 61–82.

Heymans, P., and Dubois, E. 1998. "Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements," *Requirements Engineering* (3:3-4), pp. 202–218.

Hickey, A.M. 1999. "Integrated scenario and process modeling support for collaborative requirements elicitation." (Ph.D. dissertation), University of Arizona, Tucson.

Hickey, A.M., and Davis, A.M. 2004. "A Unified Model of Requirements Elicitation," *Journal of Management Information Systems* (20:4), pp. 65-84.

Humphrey, W. 1989. *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley.

Iivari, J., Hirschheim, R., and Klein, H.K. 2004. "Towards a distinctive body of knowledge for Information Systems experts: coding ISD process knowledge in two IS journals," *Information Systems Journal* (14:4), pp. 313-342.

Im, K.S., Grover, V., and Teng, J.T.C. 2013. "Research Note—Do Large Firms Become Smaller by Using Information Technology?," *Information Systems Research* (24:2), June 1, 2013, pp. 470-491.

Jackson, M. 1995. *Software Requirements & Specifications: A lexicon of practice, principles and prejudices*. ACM Press.

Jarke, M., and Pohl, K. 1994. "Requirements engineering in 2001: (Virtually) managing a changing reality," *Software Engineering Journal* (9:6), pp. 257–266.

Jauch, L.R., and Kraft, K.L. 1986. "Strategic management of uncertainty," *The Academy of Management Review* (11:4), pp. 777-790.

Jiang, J.J., Klein, G., Wu, S.P.J., and Liang, T.P. 2009. "The relation of requirements uncertainty and stakeholder perception gaps to project management performance," *Journal of Systems and Software* (82:5), pp. 801-808.

Jingyue, L., Conradi, R., Bunse, C., Torchiano, M., Slyngstad, O., and Morisio, M. 2009. "Development with Off-the-Shelf Components: 10 Facts," *IEEE Software* (26:2), pp. 80–87.

Kapur, G.K. 2004. *Project Management for Information, Technology, Business and Certification*. Prentice Hall.

Karlsson, L., Dahlstedt, Å.G., Regnell, B., Natt och Dag, J., and Persson, A. 2007. "Requirements engineering challenges in market-driven software development – An interview study with practitioners," *Information and Software Technology* (49:6), pp. 588-604.

King, J.L. 2013. "Balance of Trade in the Marketplace of Ideas," *Journal of the Association for Information Systems* (14:4), pp. 192–197.

Klein, H.K., and Myers, M.D. 1999. "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems," *MIS Quarterly* (23:1), pp. 67–94.

Kraut, R.E., and Streeter, L.A. 1995. "Coordination in software development," *Communications of the ACM* (38:3), pp. 69–81.

Lee, A.S., and Baskerville, R.L. 2003. "Generalizing Generalizability in Information Systems Research," *Information Systems Research* (14:3), pp. 221–243.

Levéna, P., Holmström, J., and Mathiassen, L. 2014. "Managing research and innovation networks: Evidence from a government sponsored cross-industry program," *Research Policy* (43), pp. 156–168.

Light, B., and Sawyer, S. 2007. "Locating packaged software in information systems research," *European Journal of Information Systems* (16:5), Oct 2007, pp. 527-530.

Liu, J.Y.-C., Chen, H.-G., Chen, C.C., and Sheu, T.S. 2011. "Relationships among interpersonal conflict, requirements uncertainty, and software project performance," *International Journal of Project Management* (29:5), 7//, pp. 547-556.

Lowry, P.B., Moody, G.D., Gaskin, J., Galletta, D.F., Humpherys, S., Barlow, J.B., and Wilson, D.W. Forthcoming. "Evaluating journal quality and the Association for Information Systems (AIS) Senior Scholars' journal basket via bibliometric measures: Do expert journal assessments add value?," *MIS Quarterly*).

Lyytinen, K., and Rose, G.M. 2006. "Information system development agility as organizational learning," *European Journal of Information Systems* (15:2), pp. 183–199.

Mason, J. 2007. *Qualitative Researching*, (2nd ed.). London: Sage Publications.

Mathiassen, L., and Pedersen, K. 2008. "Managing uncertainty in organic development projects," *Communications of the Association for Information Systems* (23:1), pp. 484–500.

Mathiassen, L., Saarinen, T., Tuunanen, T., and Rossi, M. 2007. "A Contigency Model for Requirements Development," *Journal of the AIS* (8:11), pp. 569–597.

McConnell, S. 1998. *Software Project Survival Guide*. Redmond, Washington: Microsoft Press.
McFarlan, F.W. 1981. "Portfolio approach to information systems," *Harvard Business Review* (59), pp. 142–150.
Miles, M.B., and Huberman, A.M. 1994. *Qualitative Data Analysis*, (2nd ed.). Sage.
Miller, D., and Friesen, P.H. 1983. "Strategy-Making and Environment: The Third Link," *Strategic Management Journal* (4:3), pp. 221-235.
Milliken, F.J. 1987. "Three types of perceived uncertainty about the environment: State, effect, and response uncertainty," *The Academy of Management Review* (12:1), pp. 133-143.
Mintzberg, H. 1980. "Structure in 5's: A Synthesis of the Research on Organization Design," *Management Science* (26:3), pp. 322-341.
Mintzberg, H. 1993. *Structure in Fives: Desiging Effective Organizations*. New Jersey: Prentice-Hall, Inc.
Mitchell, R.J., Shepherd, D.A., and Sharfman, M.P. 2011. "Erratic strategic decisions: when and why managers are inconsistent in strategic decision making," *Strategic Management Journal* (32:7), pp. 683-704.
Morrissey, J.M. 1990. "Imprecise information and uncertainty in information systems," *ACM Transansactions on Information Systems* (8:2), pp. 159–180.
Na, K.-S., Li, X., Simpson, J.T., and Kim, K.-Y. 2004. "Uncertainty profile and software project performance: A cross-national comparison," *Journal of Systems and Software* (70:1–2), pp. 155-163.
Napier, N.P., Mathiassen, L., and Robey, D. 2011. "Building contextual ambidexterity in a software company to improve firm-level coordination," *European Journal of Information Systems* (20:6), pp. 674-690.
Nerur, S., and Balijepally, V. 2007. "Theoretical Reflections on Agile Development Methodologies," *Communications of the ACM* (50:3), pp. 79–83.
Newkirk, H.E., and Lederer, A.L. 2006. "The effectiveness of strategic information systems planning under environmental uncertainty," *Information & Management* (43:4), pp. 481-501.
Nicolini, D. 2009. "Zooming in and out: studying practices by switching theoretical lenses and trailing connections," *Organization Studies* (30:12), pp. 1391-1418.
Nidumolu, S. 1995. "The effect of coordination and uncertainty on software project performance: residual performance risk as an intervening variable," *Information Systems Research* (6:3), pp. 191-219.
Nidumolu, S.R. 1996. "Standardization, requirements uncertainty and software project performance," *Information and Management* (31), pp. 135–150.
Nielsen, J.A., Mathiassen, L., and Newell, S. 2013. "Theorization and Translation in Information Technology Institutionalization: Evidence from Danish Home Health Care," *MIS Quarterly* (Forthcoming).
Nonaka, I. 1994. "A Dynamic Theory of Organizational Knowledge Creation," *Organization Science* (5:1), pp. 14-37.
Orlikowski, W.J. 1996. "Improvising Organizational Transformation Over Time: A situated change perspective," *Information systems Research* (7:1), pp. 63–92.
Orlikowski, W.J., and Baroudi, J.J. 1991. "Studying information technology in organizations: Research approaches and assumptions," *Information Systems Research* (2:1), pp. 1-28.
Orlikowski, W.J., and Iacono, C.S. 2001. "Research Commentary: Desperately Seeking the 'IT' in IT Research—A Call to Theorizing the IT Artifact," *Information Systems Research* (12:2), pp. 121–134.
Parnas, D.L. 1972. "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM* (15:12), pp. 1053-1058.
Parnas, D.L., and Clements, P.C. 1986. "A rational design process: How and why to fake it," *Software Engineering, IEEE Transactions on* (SE-12:2), pp. 251-257.

Patton, M.Q. 2005. "Qualitative Research," in *Encyclopedia of Statistics in Behavioral Science*. John Wiley & Sons, Ltd.

Pawson, R., Wong, G., and Owen, L. 2011. "Known Knowns, Known Unknowns, Unknown Unknowns The Predicament of Evidence-Based Policy," *American Journal of Evaluation* (32:4), pp. 518-546.

Pohl, K. 1994. "The three dimensions of requirements engineering: A framework and its applications," *Information Systems* (19:3), pp. 243-258.

Pohl, K. 1996. *Process-Centered Requirements Engineering*. New York: Wiley.

Ramesh, B., Cao, L., and Baskerville, R. 2010. "Agile requirements engineering practices and challenges: an empirical study," *Information Systems Journal* (20:5), pp. 449–480.

Rasmusson, J. 2011. *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf.

Regnell, B., Höst, M., och Dag, J.N., Beremark, P., and Hjelm, T. 2001. "An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software," *Requirements Engineering* (6:1), pp. 51-62.

Robey, D., Smith, L.A., and Vijayasarathy, L.R. 1993. "Perceptions of conflict and success in information system development projects," *Journal of Management Information Systems* (10), pp. 123–139.

Sanchez, R., and Mahoney, J.T. 1996. "Modularity, Flexibility, and Knowledge Management in Product and Organization Design," *Strategic Management Journal* (17), pp. 63–76.

Sawyer, S. 2000. "Packaged software: implications of the differences from custom approaches to software development," *European Journal of Information Systems* (9:1), pp. 47-58.

Schultze, U. 2000. "A Confessional Account of an Ethnography About Knowledge Work," *MIS Quarterly* (24:1), pp. 3–41.

Seaman, C.B. 1999. "Qualitative methods in empirical studies of software engineering," *Software Engineering, IEEE Transactions on* (25:4), pp. 557-572.

Siddiqi, J., and Shekaran, M.C. 1996. "Requirements Engineering: The Emerging Wisdom," *IEEE Softw*. (13:2), pp. 15-19.

Siggelkow, N. 2007. "Persuasion with case studies," *Academy of Management Journal* (50:1), pp. 20-24.

Sillitti, A., Ceschi, M., Russo, B., and Succi, G. 2005. "Managing uncertainty in requirements: a survey in documentation-driven and agile companies," *Software Metrics, 2005. 11th IEEE International Symposium*: IEEE, pp. 10 pp.-17.

Simon, H.A. 1979. "Rational Decision Making in Business Organizations," *The American Economic Review* (69:4), pp. 493-513.

Simon, H.A. 1996. *The Sciences of the Artificial*. MIT press.

Sinha, K.K., and Van de Ven, A.H. 2005. "Designing Work Within and Between Organizations," *Organization Science* (16:4), pp. 389-408.

Slaughter, S.A., Levine, L., Ramesh, B., Pries-Heje, J., and Baskerville, R. 2006. "Aligning Software Processes with Strategy," *MIS Quarterly*), pp. 891–918.

Snowden , D.J., and Boone, M.E. 2007. "A Leader's Framework for Decision Making," *Harvard Business Review*), November, 2007.

SWEBOK. 2013. "Guide to the Software Engineering Body of Knowledge (SWEBOK V3)." IEEE.

Thayer, R., and Dorfman, M. 1994. *Standards, Guidelines, and Examples on System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press.

Thompson, F., and Perry, C. 2004. "Generalising results of an action research project in one work place to other situations: Princip..." *European Journal of Marketing* (38:3/4).

Thompson, J.D. 1967. *Organizations in Action*. New York: McGraw-Hill.

Torchiano, M., and Morisio, M. 2004. "Overlooked aspects of COTS-based development," *Software, IEEE* (21:2), pp. 88-93.

Trist, E.L. 1981. "The sociotechnical perspective," in *Perspectives on Organizational Behavior,* A.H. Van de Ven and W.F. Joyce (eds.). New York: John Wiley and Sons.

Van de Ven, A.H. 2007. *Enaged Scholarship: A guide for organizational and social research.* Oxford: Oxford University Press.

Vlissides, J., Helm, R., Johnson, R., and Gamma, E. 1995. *Design Patterns: Elements of reusable object-oriented software*.

Watson, H.J., and Frolick, M.N. 1993. "Determining Information Requirements for an EIS," *MIS Quarterly* (17:3), pp. 255–269.

Weber, R. 2003. "Still Desperately Seeking the IT Artifact," *MIS Quarterly* (27:2), 06//, pp. 183-183.

Weigelt, C., and Miller, D.J. 2013. "Implications of internal organization structure for firm boundaries," *Strategic Management Journal* (Forthcoming).

Williamson, O.E. 1991. "Comparative Economic Organization: The analysis of discrete structural alternatives," *Administrative Science Quarterly* (36:2), pp. 269-296.

Xu, L., and Brinkkemper, S. 2007. "Concepts of product software," *European Journal of Information Systems* (16:5), Oct 2007, pp. 531-541.

Yin, R.K. 2009. *Case Study Research: Design and Methods*, (4th ed.). SAGE Publications.

Zammuto, R.F., Griffith, T.L., Majchrzak, A., Dougherty, D.J., and Faraj, S. 2007. "Information Technology and the Changing Fabric of Organization," *Organization Science* (18:5), pp. 749–762.

Zave, P. 1997. "Classification of research efforts in requirements engineering," *ACM Computing Surveys* (29:4), pp. 315–321.

Zmud, R.W. 1980. "Management of large software development efforts," *MIS Quarterly* (4:2), pp. 45-55.

APPENDIX A: ISD REQUIREMENTS CONSTRUCTION CLASSICS

1. Agarwal, R., Sinha, A., and Tanniru, M. 1996. "Cognitive Fit in Requirements Modeling: A Study of Object and Process Methodologies," *Journal of Management Information Systems* (13:2), pp. 137-162.

   Applies cognitive fit theory to requirements modeling. Experimental group showed better performance in process-oriented modeling tasks when using a process modeling tool.

2. Byrd, T.A., Cossick, K.L., and Zmud, R.W. 1992. "A Synthesis of Research on Requirements Analysis and Knowledge Acquisition Techniques.," *MIS Quarterly* (16:1), pp. 117-138.

   Synthesizes "knowledge acquisition" and "requirements analysis" literature. Categorizes elicitation techniques.

3. Davidson, E.J. 2002. "Technology Frames and Framing: A Socio-Cognitive Investigation of Requirements Determination," *MIS Quarterly* (26:4), pp. 328-358.

   Using an example of project failure, represents how changes in framing (of both the focus of the organization and the focus of the project) affect requirements priority. Concludes requirements are social constructions, fleshed out by often undocumented social interactions.

4. Guinan, P.J., Cooperider, J.G., and Faraj, S. 1998. "Enabling Software Development Team Performance During Requirements Definition: A Behavioral Versus Technical Approach," *Information Systems Research* (9:2), pp. 101-125.

   Team skill, management involvement, and little variation in team experience led to more effective team processes during requirements development. Team members engaged in positive boundary-spanning behavior (e.g., championing) and negative boundary-spanning behavior (e.g., guarding). Guarding behavior, *that is, limiting information requested or released by a group, is shown to negatively affect performance, a result contrary to some earlier research involving different activities.*

5. Hickey, A.M., and Davis, A.M. 2004. "A Unified Model of Requirements Elicitation," *Journal of Management Information Systems* (20:4), pp. 65-84.

   Presents a unified model of requirements elicitation, synthesizing a great deal of elicitation research. Provides guidance on comparing/contrasting elicitation models.

6. Houdeshel, G., and Watson, H.J. 1987. "The Management Information and Decision Support (MIDS) System at Lockheed-Georgia," *MIS Quarterly* (11:1), pp. 127-140.

   Carefully defined requirements are one of many factors, such as strong executive sponsorship, team approach to development, and evolutionary development, that lead to the success of a specifically studied system. Although only cursorily related to requirements, this paper suggests a complete set of requirements up front would be "difficult or impossible" (p. 136), and successful development occurred due to an evolutionary approach.

7. Majchrzak, A., Beath, C.M., Lim, R.A., and Chin, W.W. 2005. "Managing Client Dialogues During Information Systems Design to Facilitate Client Learning," *MIS Quarterly* (29:4), pp. 653-672.

   Discusses "collaborative elaboration" as an elicitation technique and a way to facilitate "client learning." Dialoguing with clients produces superior design phase outcomes.

8. Markus, M.L., Majchrzak, A., and Gasser, L. 2002. "A Design Theory for Systems That Support Emergent Knowledge Processes," *MIS Quarterly* (26:3), pp. 179-212.

   Design theory for systems with ambiguously defined users, unstructured requirements, unpredictable work contexts, and tacit knowledge distributed across experts and non-experts. (Example contexts are new product development, strategic planning, organizational design.)

9. Montazemi, A.R., and Conrath, D.W. 1986. "The Use of Cognitive Mapping for Information Requirements Analysis," *MIS Quarterly* (10:1), pp. 45-56.

   Cognitive mapping is used to improve understanding of complex cause–effect relationships. In the context of requirements analysis, this provides better understanding of relationships between requirements.

10. Schenk, K.D., Vitalari, N.P., and Davis, S.K. 1998. "Differences between Novice and Expert Systems Analysts: What Do We Know and What Do We Do?," *Journal of Management Information Systems* (15:1), pp. 9-50.

    Determines individual analyst's problem-solving skills are key to defining good systems requirements, and identifies specific weaknesses that separate novice and experienced analysts. Ability to identify and define problems, greater willingness to make and discard hypotheses, and consideration of a greater number of alternatives are some characteristics that distinguish novice and expert analysts.

11. Wand, Y., and Weber, R. 1995. "On the Deep-Structure of Information-Systems," *Information Systems Journal* (5:3), pp. 203-223.

    Not about requirements construction processes, per se. Authors propose models useful for examining the sufficiency of representational grammars.

12. Watson, H.J., and Frolick, M.N. 1993. "Determining Information Requirements for an EIS," *MIS Quarterly* (17:3), pp. 255-269.

    A mixture of methods—planning meetings, informal discussions with executive users, and observation of usage context—were useful in elicitation and (pre-development) validation of system requirements.

13. Wetherbe, J.C. 1991. "Executive Information Requirements - Getting It Right," *MIS Quarterly* (15:1), pp. 51-65.

    Information overload is given as a reason for lack of fit between systems and users. Post-delivery revisions are costly, and can be prevented with up-front requirements elicitation. Lack of information sharing between functions, use of interviews for elicitation instead of group collaborative processes, questioning user needs instead of use cases, and lack of prototyping are identified as hindering development of useful systems.