A Dimension-Oriented Theory of Requirements Space in Business Information Systems

ARBI GHAZARIAN
Arizona State University
Department of Engineering and Computing Systems
7171 E. Sonoran Arroyo Mall, Mesa, AZ 85212
USA
Arbi.Ghazarian@asu.edu

Abstract: Low process predictability and, consequently, excessive rework are salient characteristics in many of today's commonly-used software development life cycle processes, making it exceedingly difficult for development organizations to deliver quality software systems within economically and technically reasonable frames. This paper advances the argument that, as a solution to this problem, building software engineering theories provides a fruitful avenue to increasing the predictability of the various software life cycle processes. Accordingly, we introduce a reusable process design methodology that relies on building software engineering theories with predictive power to inform the design of more predictive and, therefore, effective software processes. The usefulness of the proposed methodology is demonstrated through an expansive case study, which aims to design a more effective requirements engineering method for the domain of business information systems. We report results from several empirical studies to support the arguments put forward in this paper.

Key–Words: Requirements Engineering, Theory Building, Process Design, Dimension Orientation, Business Information System, Domain Engineering

1 Introduction

Compared to most other fields of engineering and manufacturing, software engineering life cycle processes are notorious for their low degree of *predictability* and consequently low degree of *repeatability*, making industrial software projects especially vulnerable to potential high risks in terms of the four fundamental project constraints of scope, cost, schedule, and quality. Even with the rapid and significant software engineering advancements of the few past decades, such as the object oriented paradigm, software patterns, programming language technology, as well as the emergence and wide adoption of modern software engineering and project management practices such as eXtreme Programming (XP) [3] and Scrum [43], major cost and schedule overruns in software projects are still the norm, rather than the exception. The quest for an economically viable process for the production of high-quality software systems remains relevant to this day. We advocate theory building as a scientific approach to address these issues. Accordingly, *this paper contributes a software engineering theory, called dimension-orientation, that provides an accurate map of the requirements space for the domain of business information systems*.

The traces of this low degree of process predictability can be found in all of the pre-release software life cycle phase activities, including requirements, design, implementation, and testing activities, leading to the suboptimal construction of software products through numerous cycles of trial and error, of guesswork and rework, and of approximating solutions, resulting in a production process that is lengthy and consequently uneconomical. Results from a 2009 published report of software development practices in the domain of enterprise information systems by Ghazarian [15], who tracked the complete history of source code changes made during the two year development period of an enterprise resource planning software system, are a case in point. The study found that about 78% of the total number of source code changes made during the two year construction period of the studied system were reworks in the form of either corrective or behavior-preserving source code changes; only about 22% of the code changes made before releasing the system involved new developments that

added functional value to the system. A significant amount of coding effort during the construction of software systems is expended on rectifying functional and design problems in code, rather than developing new code. This indicates that we do not yet know of a routine way to objectively move from a set of software requirements to a corresponding optimal implementation. Instead, relying on engineer's experience and through a creative process, we build an initial approximation and, continuously and in an ad hoc fashion, rework it towards satisfying the given requirements set. In other words, we do not have a conscientiously developed software engineering theory to predict and proactively plan for what we should expect in a software project and therefore avoid reworks as much as possible. In their 2001 list of top 10 software defect reduction factors [6], Boehm and Basili state that software projects spend about 40 to 50 percent of their effort on avoidable rework, consisting of effort spent on fixing software difficulties that could have been detected and fixed earlier and less expensively or avoided altogether. They identify hastily specified requirements as a major source of avoidable rework.

Ghazarian's study [15] further reported that at least 26% of added lines and 39% of deleted lines of code were caused by corrective changes alone. Moreover, on the average, every submission of new code to the source code repository had contained two and half new faults, which had to be detected, fixed, and verified at a later time, imposing further costs on the development process. The organization and the system reported in Ghazarian's study [15] were a common case in the domain of enterprise systems, sharing many similar characteristics with most software projects in the same domain in terms of the choice of programming language, development tools, frameworks, Application Programming Interfaces (APIs), team size and structure, and development practices. Therefore, it is very likely that many software projects and organizations within the domain of enterprise systems are challenged by the same problems. Studies of software systems in other domains, such as satellite ground control system [1] and real time system [36], have found similar results, indicating that excessive rework is not just a common condition in many software engineering endeavors, but a way of working and coping with uncertainty. Therefore, it is a highly plausible hypothesis, if not a fact, that many software projects are in a similar challenging situation as the one described in [15].

Unfortunately, the situation in post-release software life cycle phases, especially during the maintenance phase, is as challenging or even worse than the pre-release software engineering phases. When typically-late and over-budget projects eventually deliver their software products, transitioning into the operation and support stage, quality issues rear their heads, requiring numerous cycles of rework, plunging the software projects into further cost and schedule problems. The negative impacts of post-release quality issues can be external such as usability and reliability issues that directly impact both the end users as well as the organizations that depend on these software systems for their services and daily operations, as well as internal such as maintainability problems impacting the development organization and its maintenance developers. Studies have shown that maintenance costs, resulting from corrective, adaptive, perfective, and preventive changes, with a share of 40 to 90 percent of the total development costs, are the dominant cost of software systems [4, 5, 9, 11, 13].

Corrective changes alone account for a sizable portion of maintenance costs. Lientz and Swanson [33], in their study of 487 data processing organizations in 1980, reported that, on the average, about 21% of the maintenance effort is spent on corrective maintenance. Later studies in 1990 concluded that, in spite of a decade of advancements in software engineering, the maintenance problems had remained the same [38, 49]. Today, after over 20 more years of progress in software engineering, it is not difficult to observe that software maintenance is still an exceedingly major concern in software systems.

Recent empirical studies have demonstrated that mechanisms, such as design regularities [16], which increase a system's predictability are an effective means to address maintenance problems at a fundamental level [14]. Accordingly, recent effort [20] has attempted to develop quantitative models to measure and control system predictability. Degree of Regularity [20] and Regularity Density [20] are examples of such measures. The goal is to address post-release problems by producing software systems that are highly predictable. That is, systems that have a predictable design and, therefore, are easy to reason about and maintain. To build such systems necessarily requires a process that is objectively repeatable.

To sum up, to address both the pre-release as well as post-release software life cycle challenges at a fundamental level, we argue that the software engineering community needs to move towards devising objectively repeatable development processes, which can

only be accomplished through developing software engineering theories that have predictive power over aspects of software engineering interest. Such predictions can then be employed to inform software engineering processes, practices, techniques, tools, and technologies. Our observation of the status quo in the software industry is that too much emphasis is placed on software engineer's experience. Although experience can be a valuable source of insight, it should not substitute the development of theories that can afford us predictive power and therefore help us to increase the repeatability of various software engineering processes. In line with this philosophy, *this paper contributes a general research framework that takes experience and observations from real-world software engineering projects as input, subjects them to rigorous scientific evaluation, develops software engineering theories, and uses the resulting predictions to lay a solid theoretical foundation for the development of more effective and repeatable software processes*.

The scope of this current work is limited to the requirements phase of software engineering. However, this work is part of a large research program that aims to incrementally address the various areas of software engineering towards achieving its ultimate goal of developing a highly repeatable full software development life cycle process. In what follows, we will use the research framework presented in this paper to develop, evaluate, and refine a specialized software engineering theory, one that makes predictions about software requirements space. Three fundamental premises of the work reported here are as follows:

1. Shifting from current widely used opportunistic software engineering practices and processes to ones that are more predictive and repeatable will make it possible not only to eliminate a significant amount of wasted effort in reworking software artifacts, but also make reworks less difficult, when they are necessary. This premise has been supported in previous work including [19], [20], [18], and [16] as well.
2. To move from opportunistic to objectively repeatable software engineering processes, we need to develop software engineering theories with predictive power.
3. A third premise of our work is that limiting our scope to specific domains or application areas makes it possible to develop theories that, although less powerful in explaining software engineering phenomena in a wide range of domains, are more detailed and accurate within a particular domain and, thus, directly usable by software engineering practitioners within that domain. That is, from a pragmatic point of view, our goal in building specialized software engineering theories is to make accurate and detailed predictions within a domain of interest to inform the design of more effective software engineering processes within that domain. Although higher generality translates into a broader scope of applicability, but may demand more effort in operationalizing constructs and relationships of a theory to a given situation. In contrast, lesser generality might make a theory immediately applicable to a domain [44].

The research contributions of this paper are twofold: first, we propose and demonstrate the use of a general and reusable research and design methodology that, in practice, can be used to develop more effective software engineering processes, tools, and techniques. Second, although there are general guidelines and established best practices, such as the design science guidelines [25], for conducting information technology research, they require a relatively high degree of experience and both theoretical and empirical understanding and research maturity on the part of researchers, which can not be expected from beginning researchers. Therefore, we believe that a research and design process with step-by-step instructions can be invaluable to beginning researchers. In this paper, we present a research and design framework that while it conforms to established research guidelines and best practices, it is easy for inexperienced researchers to follow. Therefore, the work presented in this paper is making an educational contribution as well. Throughout this paper, we use the terms business information systems, business systems, enterprise systems, and enterprise information systems interchangeably.

The rest of this paper is organized as follows: in Section 2, we introduce our general research and design methodology. In Section 3, we follow the steps of this proposed research and design methodology to develop and evaluate a software engineering theory about the requirements space in the domain of enterprise systems. Conclusions and directions for future research follow in Section 4.

2 A Research and Design Methodology for Software Engineering

For a software engineering method to be effective to its intended use, it must be based on relevant facts

and solid evidence about the context in which the method is supposed to be applied, the range of problem types the method is supposed to address as well as the criteria for a successful outcome of the method. In this paper, we use the general term software engineering method as an umbrella term to include all software engineering processes, practices, and techniques. Software engineering methods, then, are not solutions to single well-defined existing problem instances, but rather a set of generic and possibly ordered steps, ideally with well-defined scopes of application, that aim to achieve desired outcomes for yet unobserved problems occurring within their application scope, where the outcomes of the methods conform to a corresponding set of desirable quality characteristics. This emphasis of a method in addressing yet unobserved future problems necessarily requires predictions about the characteristics of the future problems that fall within the application domain of the method. Therefore, when it comes to designing engineering methods, and in particular software engineering methods, it is of utmost importance to determine the range of problem types that such methods will need to deal with within their application scopes as a method cannot reasonably be expected to be effective in handling an unpredicted situation for which it was not originally designed.

This determination of the range and the nature of future problem types (i.e., their distinguishing characteristics) can be best accomplished through building theories with predictive power over the domain of concern for a method, thereby informing the design of the method. In the absence of a solid theoretical foundation that, within an acceptable accuracy, predicts future situations that need to be dealt with and justifies why, where, and how the method is effective, software engineering methods are prone to being developed based on unverified assumptions, opinions, and biases, rendering them ineffective or at best suboptimal. In line with this philosophy, below, we present a 12-step design methodology to develop a requirements method for the domain of enterprise systems that has a scientific software engineering foundation. The specific objective of this current work is to devise a requirements method that is capable of producing high quality requirements specifications for enterprise systems. However, the methodology can be reused to design methods for other areas of software engineering and in various application domains.

1. Formulation of a hypothesis, based on experience and expert knowledge within a domain, anecdotal evidence, accidental observations, recurring patterns, insights, ideas, and reasoning, which, if proved correct, can potentially improve some aspect of the software engineering life cycle process. The formulation of a hypothesis can lead to the formulation of a number of relevant research questions.
2. Study of the relevant literature to determine if previous work has any results that have direct or indirect bearing or can shed light upon the hypothesis and its relevant research questions.
3. Empirical verification of the hypothesis through numerous purposeful and planned observations of relevant real-world phenomenon, data collection, and data analysis.
4. Development of a theory, or one or more laws, or both, based on the stability in repeated observations and the accepted hypothesis, with explanatory and descriptive power, respectively, over the observed phenomenon, in case the hypothesis proved to be correct. Whereas laws describe the observable phenomenon (i.e., what happens), the theory explains the phenomenon (i.e., why it happens).
5. Evaluation of the predictive power of the formulated theory, or the laws of the formulated theory, in new cases within their intended domain beyond the original dataset used to build the theory or observe the laws.
6. Refinement of the theory towards more accurate predictions based on feedback from evaluation on new cases.
7. Formalization of the verified theory in a formal or semi-formal language such as an ontology, domain model, graphical notation such as Unified Modeling Language (UML), logic or other mathematical notation, or a any combination of these as appropriate.
8. Exploitation of the theory or law's predictions as a theoretical foundation to inform the development of the desired software process.
9. Controlled evaluation of the effectiveness of the new software engineering method through a proof-of-concept systems resembling the real-world problem features, a benchmark or standard problem, measurements and metrics, small-scale real-world problem, or evaluation in a simulated environment.
10. Using feedback from controlled evaluation to improve the design of the software engineering method.

11. Large-scale industrial evaluation of the effectiveness of the new process in real-life situations.
12. Using feedback from large-scale industrial evaluation to improve the design of the software engineering method as well as to refine the underlying theory.

In the above methodology, steps 1 through 6 build a solid theoretical foundation, step 7 captures and specifies this theoretical foundation in a precise language or formalism, and steps 8 through 12 put this theoretical foundation into practical use through developing a theory-backed software engineering method to be used in real-world software engineering endeavors. In this present paper, we will specifically focus on the first 6 steps of the process in order to develop a requirements engineering theory to explain and predict the requirements space in the domain of business information systems. The remaining steps will be the focus of a future work, which will aim to use the theory presented in this present work to design a specialized requirements method for the domain of business information systems.

The proposed design methodology, which combines theoretical aspects with practical concerns in software engineering, is an adaptation of scientific method to the software engineering practice. It should be noted that although we conveniently described the methodology as a sequence of steps, in practice, it will be used in an iterative fashion with feedback loops between steps, some steps occurring in parallel, or out of their presented order in the methodology. In the next section, we will follow the steps of the proposed research and design methodology to build, evaluate, and refine a theory about the requirements space in the domain of business information systems. In future work, we will continue by using this newly developed theory to design a requirements method for business information systems.

3 Design of A Requirements Method for Business Information Systems

Large business organizations today rely on enterprise systems for their daily operations. Enterprise systems are software systems that support business process of business organizations, and as such, they are often referred to as business systems, or more precisely integrated business systems. Enterprise systems provide a technological platform that enables organizations not only to integrate and coordinate their various business operations, but also facilitate the sharing of information across the various functional departments and management hierarchies within organizations. Enterprise systems can link an enterprise with its customers, suppliers, and business partners. The ultimate business goal in enterprise system implementation is to improve the effectiveness and efficiency of enterprise organizations.

A successful software project demands a correct and thorough requirements specification [45]. Enterprise systems are no exception. In fact, enterprise systems, as models of the business world, posses high information content as they embed large amounts of information about organizational policies, business entities and their relationships, business work flows, regulations, business transactions, as well as domain rules. Therefore, in the absence of a thorough requirements engineering approach that aims to systematically capture, validate, represent, and share this body of information with both technical and non-technical project stakeholders, it is unlikely to succeed in capturing a comprehensive view of an organization and its business. Studies have shown that incomplete specification of requirements is a major risk and failure factor for software projects. For instance, in an industrial case study of defect introduction mechanisms in enterprise systems [17], it was observed that specification-related defects, with a share of 42.5% of all reported defects, represented the largest category of the software defects in the studied enterprise system. Of this 42.5%, 34.7% were caused by incomplete requirements specifications, while the remaining 7.8% were a result of a lack of traceability between various requirements specifications. Among other results, the study concluded that improving the requirements process in terms of the completeness of the produced specification, including explicit documentation of all business rules and data validations can play a significant role in reducing defect rates in enterprise systems. Another study of pre-release software faults in enterprise systems [15], demonstrated that faults of omission with a share of 24% and spurious faults with a share of 15.7% of all the faults were among the dominant classes of faults and could be largely traced back to the requirements phase of the development life cycle. This study also, not surprisingly, concluded that more development effort should be directed toward the specification of requirements.

There is consensus in the software engineering literature that requirements methods that can help in the development of high quality system and software specifications, that is, correct, complete, and consis-

tent specifications, as well as methods that can help with the early detection and removal of problems from specifications of requirements can significantly lower software project costs, while at the same time increase the quality of the delivered software system. Boehm and Basili state that [6] finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

As we stated earlier in our third premise that underlies this work, it is our position that focusing on a particular domain, such as the domain of enterprise systems, will better allow us to possibly identify laws that govern the domain. Given the abundance of evidence from previous studies on the importance of high quality specifications of requirements in the success of software projects, in this current work, we aim to discover laws concerning the requirements space in enterprise systems. The argument being put forward here is that the predictions made by such laws of a domain, as well as their encompassing theory, will allow us to develop specialized domain-specific requirements methods that, compared to current domain-agnostic requirements methods, will be more effective in producing complete specifications of requirements, and consequently, help us to avoid unnecessary and excessive reworks. As described in our design methodology, the process begins with formulating a hypothesis, which we will discuss in the subsection that follows.

3.1 Formulation of the Low-Cardinality Hypothesis

The idea of designing a domain-specific requirements method specifically for enterprise systems, one that would potentially be more effective in producing complete specifications of requirements for enterprise systems and would consequently yield higher quality requirements specifications than contemporary domain-agnostic requirements methods, emerged in the industry, from the observation of a recurring phenomenon, a pattern, in the requirements for several enterprise systems. The author of this paper, after many years of engagement in the development of enterprise software systems, had noticed a pattern that all enterprise system development projects that he had encountered thus far, regardless of their user organization, type of business supported, or application area, essentially had to implement similar types of requirements. The statements of functional requirements would naturally vary from one project to another to reflect the specific user and business needs in their respective business and organizational context. However, regardless of these variances in the business or enterprise area, the types of functional requirements would largely remain the same from one project to another. For instance, it was observed that diverse applications, such as accounting, inventory control and management, sales management, order processing, customer relationship management, supply chain management, and enterprise resource planning, just to name a few, all in their specifications of requirements had a large number of requirement statements that described a few frequently-occurring software problem dimensions - or enterprise dimensions - such as inputs to a business process, the content and the format of the outputs of a business process, rules for validating the inputs to the system, and business rules or rules of the enterprise domain. The observation was the same regardless of the fact that the projects belonged to different business organizations in different countries with different rules for conducting business, and were concerned with completely different business areas. What was common among all the observed projects was that they all belonged to the domain of enterprise systems. This simple observation led to the formulation of the following hypothesis:

The majority of functional software requirements in enterprise systems belong to a small core set of requirements classes, - or enterprise dimensions - each class representing a problem dimension in the domain of enterprise systems. In other words, the requirements space in the domain of enterprise systems, for the most part, deals with a limited set of problem dimensions.

In technical terms, this hypothesis states that the requirements space in the domain of enterprise systems can be characterized as a *densely-clustered low-cardinality multi-dimensional space*. We will refer to this hypothesis as the low-cardinality hypothesis throughout this paper.

This low-cardinality hypothesis necessitates answering the following research questions, if we aim to evaluate its validity and eventually arrive at an encompassing theory with predictive laws about the requirements space in the domain of enterprise systems:

1. What are the classes of functional requirements in the domain of enterprise systems? In other words, what are the enterprise dimensions? Or equivalently, what would a comprehensive taxonomy of problem dimensions in the domain of enterprise systems look like?
2. How frequently each identified requirements

class occur? In statistical terms, what is the frequency distribution of the various types of requirements? Which classes of requirements are frequently-occurring - or core - to the domain of enterprise systems?

3. Are there any relationships among the various identified enterprise problem dimensions? If so, what is the nature of these relationships?
4. Is it possible to use the knowledge of the enterprise requirements classes (i.e., dimensions), their frequency distributions, and the relationships among enterprise dimensions to improve the requirements process for enterprise systems? And if so, how?

The first three research questions are ontological questions about the domain of enterprise systems regarding what exists (i.e., a taxonomy of entities), which entities are most fundamental, and what the nature of the relationships are among the entities or classes of concepts, respectively. The fourth research question is pragmatic in nature as it aims to explore ways to exploit the theoretical knowledge gained from answering the first three questions to improve an aspect of real-world software engineering. To answer the first research question above requires the development of a taxonomy, which, as emphasized in [44], is needed to support theory building.

3.2 Related Work

Before attempting to evaluate the validity of the proposed hypothesis and answer the ontological questions surrounding it, we conducted an extensive search of published literature to see if previous work, to any degree, had addressed these questions. To our disappointment, there was not much work reported specifically on requirements for the domain of enterprise systems. Given the abundance of evidence in the software engineering literature on the importance of the requirements phase in improving software quality and reducing cost and schedule issues in software projects, this low number of publications on enterprise systems development, and in particular requirements for enterprise systems, is surprising because enterprise systems account for a sizable sector of the software development industry, and similar to other types of system development, suffer from quality issues as well as cost and schedule overruns. This observation has been supported in [15] as well, arguing that this lack of information about enterprise systems might be due to the commercial sensitivity of business systems, which compels business organizations not only to make it hard for researchers to obtain access to these systems for research purposes, but also restrict the sharing of knowledge gained from studying such systems.

We found that even in the broader software engineering literature, a detailed and accurate understanding of the nature of functional software requirements has not received the attention it deserves. Addressing the research challenges posed by non-functional requirements seems to have overshadowed the study of functional requirements to a degree where we were not even able to find detailed taxonomies of functional requirement types. Published literature on requirements classification, for the most part, merely separates functional requirements from non-functional requirement, organizing them into broad and generalized categories, leaving the details to be worked out by software engineers in an ad hoc fashion. What we seek is a pre-conceived and accurate model of functional requirements to guide the requirements engineers during the requirements phase.

To give a few examples of requirements taxonomies, the software quality models FURP [23] and its extensions by Rational Software [27, 30, 40] FURPS+ have one broad category for functional requirements (the F in FURPS) and four major categories for non-functional requirements, namely Usability, Reliability, Performance, and Supportability (URPS). As another example, the IEEE recommended practice for software requirements specification [26] does not provide a detailed classification scheme for functional requirements. It lists four categories of non-functional requirements: external interface requirements, performance requirements, design constraints, and software system attributes. Functional requirements, consisting of inputs, outputs, and processes, are merely separated from non-functional requirements and can be organized along broad dimensions such as mode, user class, objects, response, and functional hierarchy. This over-generalization of functional requirements was probably necessary to keep the standard applicable to a wide variety of software domains. However, the flip side of a generalized standard is that as we distance ourselves from the peculiarities of various software domain, we lose our capability to provide detailed guidelines for the requirements engineering process, relying on the requirement engineer's experience and knowledge of the domain to work out the details. In contrast to previous models of functional requirements, this current work aims at arriving at a detailed classification and characterization of functional requirements in enterprise systems.

In the subsections that follow, we will empirically verify the validity of the low-cardinality hypothesis. In the process, we will find answers to the ontological and practical questions raised in Subsection 3.1 including the development of an empirically derived requirements taxonomy for functional requirements that is more detailed and comprehensive compared to existing taxonomies. This will corresponds to the third step in the proposed design methodology.

3.3 Evaluation of the Low-Cardinality Hypothesis

To evaluate the validity of the proposed hypothesis, we collected, classified, and statistically analyzed data from software requirements for 15 software projects, including 14 private industrial and one public projects, all in the domain of enterprise systems. To facilitate the evaluation process, we created a research database to store and organize the collected data in the form of atomic statements of functional requirements. The atomicity [19] of requirements ensured that all the collected data points (statements of requirements) are at the same level of granularity. To increase the validity of our findings, we collected a large dataset, including 1217 statements of functional requirements from 171 system functions, representing a wide variety of applications in the domain of enterprise systems. We then added meta-data to each statement of requirement including the class of functional requirements to which the requirement statement belonged. Quality control for the data collection and classification phases included the independent repetition of data classification, group reviews of the dataset, and statistical random inspection of data and its classification. After ensuring the quality of the collected data and its classification, we generated reports about the studied projects, including the identified classes of functional requirements (enterprise problem dimensions), their frequency distributions, as well as descriptive statistics, such as average, standard deviation, and mean, about each class of requirements. A detailed account of this study can be found in [22]. For convenience, the results are summarized in Table 1 below. Table 2 provides brief descriptions for the requirements classes identified in Table 1.

Table 1 provides answers to the first two research questions raised in Subsection 3.1 concerning the emergent classes of functional requirements in enterprise systems and their frequencies. It suggests that the requirements space in the domain of enterprise systems is composed of 12 classes of requirements, or problem dimensions, as listed in the first column of Table 1, with the five classes of data outputs, data inputs, event triggers, business logic, and data persistence occurring more frequently than the other classes, making them core to the domain of enterprise systems. These five core requirements classes each contribute over 10% to the total number of requirements in the entire dataset as well as in their respective projects and together account for about 85% of the requirements space in the studied enterprise projects. This observed distribution of enterprise requirements supports the proposed hypothesis that the requirements space in enterprise systems is densely clustered around a few core problem dimensions thus a low-cardinality multi-dimensional space.

To answer our third research question regarding the relationships among the classes of requirements - or problem dimensions - we analyzed a large number of requirements to identify potential relationships among them. The observation was that the existence of one type of requirement makes the existence of another type of requirement very probable. In other words, in complete specifications of systems, the inclusion of a requirement into the specification of a system makes it necessary, in more cases than one would randomly expect, to add one or more other relevant requirements to the specification to maintain its completeness. That is, in a large number of cases, the classes of functional requirements in enterprise systems are existentially dependent upon each other. The problem dimension of the original requirement is what determines what additional types of requirements need to be specified.

As an example, it was observed that, in many cases, requirements that specify the data input dimension of enterprise systems need additional specifications of data validation rules to prevent erroneous data entry by end users into these systems. Without such additional statements of requirements, system specifications would be incomplete, leading to quality, cost, and schedule issues during the subsequent stages of system development life cycle [17, 15].

Table 3 presents a list of probabilistically strong relationships that we observed in our dataset. To get a sense for the strength of the relationships among the identified problem dimensions and system functions in enterprise systems, we calculated the probabilities of each of the identified relationships being true in 5 randomly selected enterprise projects from our data set. The results are shown in the second column of Table 3. For instance, the fifth row in Table 3 indicates that 94% of system functions in the studied en-

Table 1: Frequency Distribution of Identified Problem Dimensions in Business Information Systems

Enterprise Problem Dimension	% of Total Requirements N= 1217	Average (%) Over 15 Enterprise Projects Observed	Standard Deviation	Median (%)
Data Output	26.37	22.21	11.29	20.51
Data Input	19.88	19.58	5.42	18.47
Event Trigger	16.18	11.70	7.84	11.11
Business Logic	11.66	14.56	8.75	14.28
Data Persistence	10.84	14.53	11.11	11.76
UI Navigation	4.84	6.43	6.75	4.54
External Call	2.62	3.00	5.70	0.00
Communication	2.30	1.32	2.04	0.00
User Interface (UI)	1.97	2.04	3.80	0.00
UI Logic	1.64	2.26	3.16	0.49
Data Validation	0.98	1.65	2.43	0.00
External Behavior	0.65	0.65	1.70	0.00

Table 2: Functional Requirements Categories and Their Descriptions

Requirement Class	Description
Data Output	the intermediate or final results of the system operations outputted to an output device, including the contents of the outputs and the formatting rules for displaying those contents.
Data Input	description of the external data items that are to be inputted into the software system.
Event Trigger	description of the internal or external stimulating actions or conditions, such as clicking on a menu item, link, or button, or a variable's value reaching a threshold that trigger system operations.
Business Logic	description of the application or business rules including workflows and calculations that define and govern the operations in a particular application area.
Data Persistence	descriptions of all the database related operations including reading, updating, inserting and deleting from/to a database.
UI Navigation	description of the flow of the screens (i.e. the rules for transition between screens) that make up an application.
External Call	description of the function calls between two systems including the description of the parameters used to make such calls and their expected values or responses.
Communication	description of the rules and the contents for electronic communication, such as email communication, between a system and an outside party.
UI	description of the *static* layout of the pages and screens that make up a system's user interface.
UI Logic	description of the *dynamic* behavior of a system's user interface (i.e., how the user interface interacts with its users).
Data Validation	description of the validation rules required to ensure the correctness of the inputted data items in terms of the permissible domain of values, the value ranges, and their correct formats.
External Behavior	description of the behavior of an operation or function in an external component or system.

terprise systems had at least one statement of requirement specifying the data input dimension of the system functions; only 6% of system functions observed in these enterprise systems did not need an input. As another example, the last relationship in Table 3 indicates that 40% of input items in the studied systems had at least one related data validation. It was important to capture the concept of the strength of a relationship because the dependencies among the system functions and problem dimensions were not always guaranteed. For example, we observed, as expected, that not all inputs necessarily needed validations, or not all system functions needed inputs from end users.

Rules 2 through 7 in Table 3 collectively tell us a great deal about the make up of system functions in the observed enterprise systems. They suggest that system functions in enterprise systems are largely composed of inputs, data validations, business rules, outputs, and persistence requirements. The other rules (1, 8, and 9) identify dependencies among the problem dimensions that make up the system functions.

This measurement of the strength of the various possible relationships between classes of requirements can be helpful in distinguishing between accidental versus persistent relationships in enterprise systems. Later, when we develop our domain model or theory, such information can inform us on which relationships to include and which ones to exclude. In table 3, we only included strong relationships between entities, the ones that capture an inherent characteristic of the domain and therefore we wish to include them in our domain model. In building a domain model or theory, we are not concerned with capturing accidental relationships as they do not represent common patterns in the domain and therefore cannot give us any insights about yet unobserved systems in the domain, beyond the system in which they were observed. As an example of a weaker relationship, we considered a relationship, which was observed in a few systems, stating that for every system function, there exists at least one external call. The statistical probability of this relationship was calculated to be slightly below 8% so we decided not to include it among our strong relationships. Another example is the relationship stating that every function of the system has at least one communication requirement with a statistical probability of 0% in the 5 studied systems. This is a prime example of a weak relationship, indicating an accidental requirement that might be observed in an enterprise system, but the problem dimension it describes does not represent an essential characteristics of the domain of enterprise applications and therefore can not be persistently and frequently found in the domain of enterprise systems.

Having arrived at a first-cut in answering our first three research questions regarding the core and non-core problem dimensions that make up the requirements space as well as the nature and the strength of the relationships in the requirements space for enterprise systems, our next step should be to develop an initial theory about the requirements space in the domain of enterprise systems. It is only after developing a theory with strong predictive power that we can answer our forth and practical research question raised in subsection 3.1

3.4 Dimension-Orientation: A Theory of Requirements Space

Let us begin this subsection with a famous quote from Kurt Zadek Lewin, who emphasized the importance of theories and their practical applications: "There is nothing so practical as a good theory" [32]. Our position is that, there is a shortage of conscientiously developed theories in the field of software engineering to explain and predict software engineering phenomena of interest and therefore the software engineering research community needs to focus more on building software engineering theories to discover, accumulate, and communicate software engineering knowledge. This view has also been supported by several software engineering researchers including Basili [2], Tichy [46], Sauer et al. [42], Kitchenham et al. [29], Endres and Rombach [12], Herbsleb and Mockus [24], Land et al. [31], and Jorgensen and Sjoberg [28]. Sjoberg et al. [44], in their guidelines on building software engineering theories, encourage theorizing as early as possible in spite of the likelihood of failures, arguing that in the absence of a theory to guide data collection, blindly gathered information might turn out to be useless. Moreover, a large bulk of information may render the beginning of theorizing next to impossible [7].

Here, we attempt to summarize our findings thus far into a theory that accounts for our observations in the 15 studied systems. To describe our theory, we adopt the four-component theory structure, suggested by Sjoberg et al. [44], where the description for a theory is divided into the following four parts:

1. Constructs
2. Propositions
3. Explanations
4. Scope

Table 3: Probabilistic Existential Relationships among System Functions and Problem Dimensions

Existential Dependency Relationship	**Probability (%)**
For every Event Trigger, there exists at least one other corresponding system function or requirement.	100
For every system function, there exists at least one corresponding Data Validation.	100
For every Data Validation, there exists at least one corresponding Data Output.	100
For every system function, there exists at least one corresponding Data Persistence.	96
For every system function, there exists at least one corresponding Data Input.	94
For every system function, there exists at least one corresponding Data Output.	94
For every system function, there exists at least one corresponding Business Logic.	94
For every item of Data Input, there exists at least one Item of Data Persistence.	51
For every Item of Data Input, there exists at least one corresponding Data Validation.	40

In the above theory structure, constructs represent the basic elements of the theory. According to [44], a software engineering theory is defined as a theory that includes at least one construct that is software engineering-specific. In our study, the main constructs include (a) the functional requirements space (b) its constituent problem dimensions represented by requirements classes, (c) system functions, and (d) atomic software requirements that make up the system functions. Propositions, or more precisely, relational propositions, of the theory describe the interactions among the theory constructs. In our study, this corresponds to the ontological statements about the make up of the functional requirements space (i.e., taxonomic categories of functional requirements) and their properties (e.g., being core or non-core) as well as the existential dependency relationships among system functions and problem dimensions listed in Table 3. Relational propositions of a theory give rise to the theory's predictive power. They describe the laws of the domain.

Explanations, or more precisely, explanatory propositions of a theory, are further propositions that explain why the relational propositions hold (i.e, the notion of causality). Explanatory propositions are what give the theory explanatory power. While the propositions (i.e., laws) of a theory describe what happens, the explanations of a theory describe why they happen [48, 41]. The scope of a theory describes the universe of discourse in which the theory is applicable. That is, it describe the conditions under which the theory's propositions are supposed to be applicable [10]. In our study, the domain of business information systems (i.e., enterprise information systems) is the scope of our software engineering theory.

Tables 4 and 5 below explicitly present the constructs, propositions, explanations, and the scope of the dimension-oriented theory. Its 12 propositions express relationships among its 21 constructs. Each proposition is explained by a corresponding explanation. The first construct, Functional Requirements Space, refers to the set of all imaginable statements of atomic requirements in the domain enterprise systems. The construct System Function is a general term that refers to a set of logically related atomic requirements that together provide a piece of functionality. From a functional point of view, enterprise systems are composed of a set of system functions. We use the term system function throughout this paper as synonymous with other commonly-used terms such as use case, feature, and high-level requirement. Since each atomic statement of requirement can have only one type (i.e., belongs to exactly one class of requirement), system functions and their encompassing requirements space are partitioned into a set of functional requirements classes. The descriptions for the identified classes of functional requirements, denoted in Table 4 by symbols C5 through C16, was provided earlier in Table 2. Core functional requirement classes are those classes of functional requirements that describe inherent characteristics of the domain as evidence by their frequent occurrence in specifications of systems in the domain. Non-core functional requirements classes, in contrast, are accidental, and not inherent, to the domain. They occur infrequently and do not represent fixed characteristics of the domain.

Requirements classes of data input and data persistence often provide a list of data items that must be inputted or stored into the system. Constructs, C19 and C20, data input items and data persistence items, refer to these data items, respectively. Compound requirements are a combination of two or more atomic requirements. Such statements of requirements must be decomposed into their constituent atomic requirements before they can be classified under one of the requirements classes.

3.5 Evaluating the Predictive Power of the Dimension-Oriented Theory

In the proceeding subsection, we formulated a theory about the requirements space in the domain of enterprise systems. However, an important step in the theory-building process remains to be done, which is to evaluate the validity of the theory's predictions through empirical studies. In the case of our theory, we need to empirically verify whether the propositions - or laws - of the theory hold true for new (i.e., unobserved) systems from the domain of enterprise systems. To do so, we conducted case studies using three new industrial projects from the domain of enterprise systems. We refer to these three projects as Test Case 1, Test Case 2, and Test Case 3, respectively. Test Case 1, which was an online marketplace for audio content, had 46 pages of requirements, including 71 system functions (i.e., use cases) and a total of 577 atomic functional requirements. We included the entire requirements set for Test Case 1 in our study. Test Case 2 was a web-based investment management and trading system with a 71-page requirements specification. Test Case 3 was an online banking software system with a 94-page requirements specification. We randomly sampled a set of 50 atomic functional requirements from each of the Test Case 2 and Test Case 3. Overall, this gave us a test dataset including 677 atomic statements of requirements from three new enterprise systems. To evaluate the the first three propositions of the theory (P1, P2, and P3), as shown in Table 4, we replicated the data classification and analysis procedure described in subsection 3.3 with the new dataset. Table 6 shows the results for each of the three test cases.

As it can be calculated from Table 6, Proposition P1 of the theory presented in Table 4 correctly predicted 99.31% of problem dimensions in Test Case 1, 94% in Test Case 2, and 100% in Test Case 3, respectively. Another way to look at these results is that we analyzed 677 atomic functional software requirements in the domain of enterprise systems and we only found 7 statements of requirements that could not be classified under one of the categories provided by the domain model of the proposed dimension-oriented theory. The remaining 670 requirements, accounting for 98.96% of the total number of requirements in our three test data sets, were covered by the 12 functional requirements classes listed in Proposition P1 of the theory. We only need to add two new non-core requirements classes, namely post-condition and data source, to achieve 100% coverage in all the of the three new system.

It must be noted that the field of empirical software engineering shares many common methodological issues with behavioral and social sciences, including the notion of the falsification of a theory. In such fields of study, regarding a theory as false based on its predictions is rarely feasible [44, 34, 50]. If a prediction is not supported, or only partially supported, by empirical evidence, an alternative theory or a refinements of the existing theory is sought, rather than the complete rejection of the theory. In Subsection 3.6, we will use feedback from the results of the evaluation of the theory to refine it.

Proposition P2 of the theory identifies data outputs, data inputs, event triggers, business logic, and data persistence as the five dominating classes of functional requirements in the domain of enterprise systems. As indicated by Table 6, In Test Case 1, data inputs, data outputs, and event triggers are indeed among the most frequently-occurring functional requirements classes as predicted by Proposition P2. However, the two functional requirements classes of business logic and data persistence, contrary to the prediction, are not among the core requirements classes in Test Case 1; instead, the three requirements classes of user interface and user interface logic followed by user interface navigation are among the most frequently-occurring categories of requirements. This is an interesting observation because the three requirements classes that were not predicted by the theory as core requirement types are all user interface-related.

We found an explanation for this discrepancy. In practice, it is not uncommon for development organizations to capture their user interface-related requirements using wireframes, prototypes, screen mocks, and other similar techniques that are visual rather than textual. In such case, as a result, fewer user interface-related requirements end up in the requirements specification documents, which can introduce noise in theories of requirements that are built merely based on data from these textual specifications. As it

Table 4: Constructs, Propositions, Explanations, and the Scope of the Dimension-Oriented Theory

Constructs	
C1	Functional Requirements Space
C2	Functional Requirements Classes (Problem Dimensions)
C3	System Functions
C4	Atomic Functional Requirements
C5	Data Output Functional Requirement Class
C6	Data Input Functional Requirement Class
C7	Event Trigger Functional Requirement Class
C8	Business Logic Functional Requirement Class
C9	Data Persistence Functional Requirement Class
C10	UI Navigation Functional Requirement Class
C11	External Call Functional Requirement Class
C12	Communication Functional Requirement Class
C13	User Interface Functional Requirement Class
C14	UI Logic Functional Requirement Class
C15	Data Validation Functional Requirement Class
C16	External Behavior Functional Requirement Class
C17	Core Functional Requirements Class (Core Problem Dimension)
C18	Non-Core Functional Requirements Class (Non-core Problem Dimension)
C19	Data Input Item
C20	Data Persistence Item
C21	Compound Requirement
Propositions	
P1	The *Functional Requirements Space* is composed of 12 known *Functional Requirements Classes*, including *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, *Data Persistence*, *UI Navigation*, *External Call*, *Communication*, *User Interface*, *UI Logic*, *Data Validation*, and *External Behavior*.
P2	The five functional requirements classes of *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, and *Data Persistence* belong to *Core Functional Requirement Classes*.
P3	The seven functional requirements classes of *UI Navigation*, *External Call*, *Communication*, *User Interface*, *UI Logic*, *Data Validation*, and *External Behavior* belong to the *Non-core Functional Requirement Classes*.
P4	For every *Event Trigger*, there exists one corresponding *System Function*, *Compound Requirement*, or *Atomic Functional Requirement*.
P5	For every *System Function*, there is a high probability that at least one requirement of type *Data Validation* should also exist.
P6	For every requirement of type *Data Validation*, there is a high probability that at least one corresponding requirement of type *Data Output* should also exist.
P7	For every *System Function*, there is a high probability that at least one corresponding requirement of type *Data Persistence* should also exist.
P8	For every *System Function*, there is a high probability that at least one corresponding requirement of type *Data Input* should also exist.
P9	For every *System Function*, there is a high probability that at least one corresponding requirement of type *Data Output* should also exist.
P10	For every *System Function*, there is a high probability that at least one corresponding requirement of type *Business Logic* should also exist.
P11	For every *Data Input Item*, there is a high probability that at least one corresponding *Data Persistence Item* should also exist.
P12	For every *Data Input Item*, there is a high probability that at least one corresponding requirement of type *Data Validation* exist.

Table 5: Table 4 Continued

Explanations	
E1	The functional requirements space in the domain of enterprise applications can be logically divided into two partitions: (a) core functional requirements classes and (b) non-core functional requirements classes. As of yet, there are 5 known core and 7 known non-core classes of functional requirements. The sets of core and non-core classes of functional requirements will be updated with future observations and discoveries. See E2 and E3 as to why core and non-core classes exist in the domain.
E2	Enterprise software systems, in order to support their corresponding business processes in the real-world, frequently need to run system functions (through Event Triggers), which collect information (Data Input), execute the prescribed business rules pertaining to the supported business processes (Business Rules), store information as a result of executing the system functions (Data Persistence), and interact with users through displaying the results of executing the system functions, or messages indicating success or failure of the processes (Data Output). The sequence of event trigger, input, business rule execution, output, and persistence, or a variation of this sequence, is very common in business systems, giving rise to the 5 core classes of functional requirements.
E3	In addition to the core functional requirements classes, enterprise systems occasionally need to support features that are not common in the domain, but specific to particular applications within the domain. Although compiling an exhaustive list of non-core requirements classes would require surveying every imaginable application in the domain and therefore an impossible task to carry out, in our studies thus far, we have empirically discovered 7 classes of non-core requirements.
E4	Event triggers, by definition, force the execution of either a single atomic requirement, a compound requirement, or a system function (e.g., a collection of atomic and compound requirements).
E5	Most business services, and system functions as their implementations in enterprise systems, need to collect information, typically from end users, who can potentially input erroneous data into the system, necessitating, as a best practice, the validation of the inputted data for correctness of value and format. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing such validation rules.
E6	When the execution of a data validation rule results in the detection of a violation of the corresponding correctness rules (i.e., detection of erroneous input), as a best practice, the user of the system needs to be notified, through appropriate error messages and possibly be given instructional messages, describing a course of action that can be taken to rectify the problem. Therefore, the existence of a requirement of type data validation, with a high probability, implies the existence of one or more related requirements of type data output.
E7	Most business services, and system functions as their implementations in enterprise systems, need to store information, typically collected from end users or generated as a result of a business transaction with the end user. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data persistence aspects of the system function.
E8	Most business services, and system functions as their implementations in enterprise systems, need to collect information, typically from end users. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data input aspects of the system function.
E9	According to E5, it is very likely that a system function will have at least one requirement of type data validation. According to E6, the existence of data validation requirements implies a high probability for the existence of corresponding data output requirements. From E5 and E6, we can conclude P9. Moreover, System functions often need to notify their end users of the successful completion or failure of the system function through requirements of type data output, giving rise to further opportunities for the existence of requirements of type data output in system functions.
E10	Most business services, and system functions as their implementations in enterprise systems, involve the execution of business rules. Therefore, it is very likely for system functions in the domain of enterprise systems to have one or more statements of requirements, describing the related business rules.
E11	Business services, and system functions as their implementations in enterprise systems, often need to store information collected from end users to conduct business and process business transactions. Therefore, it is very likely for input data items to be stored in the system. From a specification point of view, input data items reappear in statements of data persistence requirements.
E12	To prevent data entry errors, when such errors are possible, as a best practice, system functions need to validate inputted data items before consumption and storage in the system. Therefore, data input items may need one or more relevant statement of requirements describing the data validation rules.
Scope	
S1	The Domain of Enterprise/Business Information Systems

Table 6: Frequency Distribution of Identified Problem Dimensions in Business Information Systems

Problem Dimension	(%) Test Case 1 N = 577	(%) Test Case 2 N = 50	(%) Test Case 3 N = 50
Data Output	9.53	36.00	24.00
Data Input	16.81	0.00	10.00
Event Trigger	25.12	14.00	10.00
Business Logic	1.90	16.00	12.00
Data Persistence	0.17	0.00	0.00
UI Navigation	8.31	10.00	6.00
External Call	0.00	0.00	0.00
Communication	0.34	0.00	0.00
User Interface (UI)	25.99	10.00	4.00
UI Logic	10.39	8.00	2.00
Data Validation	0.69	0.00	22.00
External Behavior	0.00	0.00	0.00
Post Condition	0.69	0.00	0.00
Data Source	0.00	6.00	0.00

can be seen in Table 1, this phenomenon can be observed in the dataset that was used to build our original theory, where the median values for the user interface navigation, user interface, and user interface logic class of requirements are 4.57, 0.00, and 0.49, respectively. These low median values are an indication that in many of these enterprise systems user interface-related requirements were not thoroughly specified textually within their corresponding requirements specifications. In other words, in building our theory about the requirements space in the domain of enterprise systems, we assumed that textual specifications of requirements for enterprise systems are an accurate representations of requirements that actually get implemented in these systems. This assumption is not accurate in cases where a statement of requirement is missing from the textual specification of requirements, but actually gets implemented in the software system (i.e., discrepancy between textual specification and implementation). This assumption, we believe, was a reasonable assumption to begin with. However, we realize that in Level I information system theories [44, 35, 51, 8], which describe working relationships that are concrete and based directly on observations, such discrepancies can lead to inaccuracies in the propositions of the theory. Therefore, we need to reflect all reasonably valid explanations of discrepancies between theory predictions and empirical observations to refine and therefore increase the accuracy of a theory's predictions. In the case of our theory, this means that we will need to refine it to regard user interface related requirements as core requirements in enterprise systems.

In Test Case 2, event triggers, business logic, and data outputs were the three core classes of functional requirements that were correctly predicted. The theory correctly predicted four core classes of functional requirements in Test Case 3.

Two further observations deserve attention in Test Case 2 and Test Case 3. First, we noticed that there are no requirements of type data input in Test Case 2. Second, data validations are among the most frequently-occurring requirements in Test Case 3. Both of these empirical observations are inconsistent with our theory predictions. As reflected in the theory propositions, in the majority of systems we have looked at in the past, data inputs have typically been among the core requirement types whereas data validations have been among the least-frequently occurring requirement types, accounting for a small share of requirements in this domain.

To understand the reason for these contrasting observations, we inspected the requirements specification documents for Test Case 2 and Test Case 3 and compared them to all the previous systems we had studied. We noticed a striking difference in terms of the specification style and format; whereas all of the previous cases we had studied had their requirements specified in the form of use case documents, the requirements for Test Case 2 and Test Case 3 were specified using proprietary templates. In Test Case 2, the template for the requirements document included a table for each application screen, with rows for each item on screen and columns for the format and data validation rules for each item. It was precisely this imposed documentation structure that had obliged the requirements engineers to capture a large number of data validation rules. In the absence of such an structure, many of these data validation requirements would have remained implicit and undoc-

umented. In light of this finding, we inspected the requirements specifications for the enterprise systems that were used to build the theory and found that the low number of data validations in these systems and their consequent identification as a non-core class of functional requirements were in fact a result of under-specification of data validations in these systems; had these systems specified all their missing data validations, data validations would become a core requirements class. As a result, we should update out theory to regard data validations a core requirement class.

In Test Case 3, the specification of requirements was driven by screen mock-ups. The screen mock-ups were not meant to serve as the final design for the system's user interface; they were only employed as a means to facilitate the capturing of requirements and for illustrative purposes only. Editable user interface components on screen mock-ups were meant to implicitly suggest the data input requirements for each application screen and as such they were not explicitly specified. This implicit and visual style of specification is in contrast to the use case format for requirements specification, where, typically, the textual description of the use case includes one or more steps to capture data input requirements. This explained why there were no statements describing data inputs in Test Case 3. The absence of data inputs in Test Case 3 was a byproduct of a stylistic choice in the specification of requirements rather than any indication of features that do not require data inputs. If we were to convert the requirements documents for Test Case 3 into textual use case format, data inputs could well be among the core requirements types. This means that data inputs should remain a core requirement class and therefore no changes are needed to be made to the theory in spite of the fact that we do not explicitly see requirements of type data input in the test system.

In the light of these explanations as to why there are a few inconsistencies between theory predictions and empirical observations, we conclude that requirements belonging to the core classes, whether implicit or stated, tend to exist in almost all systems in this domain and occur more frequently, though due to specification style and other factors, some core requirement classes might remain unstated and implicit. Non-core requirements classes, in contrast, are not commonly observed in systems in a domain and even when they occur, they account for a relatively small share of the total number of requirements.

3.6 Theory Refinement - Dimension-Orientation Version 2

Theory building is an iterative process, involving continuous development and refinement. In this subsection, we will use feedback from the theory evaluation process, described in the previous subsection, to refine the theory. Based on observations and analyses of the 15 industrial cases in the domain of enterprise systems that were used to build the initial version of the theory as well as the three test cases that were used to evaluate the original theory, we have identified a domain model, consisting of 9 core requirements classes. These core requirements classes include data output, data input, event trigger, business logic, data persistence, data validation, user interface, user interface logic, and user interface navigation. Non-core requirements classes that we have observed so far include the requirement classes of external behavior, external call, communication, post-condition, and data source. Accordingly, we will need to make the first three of the following changes to the propositions of the original theory described in Table 4. Further speculations lead to the last two necessary changes below:

1. Update Proposition p1 to include the new functional requirements types of post-condition and data source.
2. Update Proposition P2 to add data validation, user interface, user interface logic, and user interface navigation to the list of core functional requirements types.
3. Update Proposition P3 to remove data validation, user interface, user interface logic, and user interface navigation from the list of non-core functional requirement types. Also, the two new non-core functional requirements classes of post-condition and data source should be added to the list of non-core requirements classes.
4. A new proposition to complement Proposition P4, taking into account the reverse case, where the existence of a system function necessarily requires the existence of a corresponding event trigger.
5. Proposition P12 subsumes Proposition P5 and therefore Proposition P5 is redundant and can be removed from the theory, making the theory more parsimonious. As we will discuss later in this paper, parsimony is a goodness criterion for empirically-based theories like ours [44].

We will accordingly need to update explanations E1, E2, and E3 to account for the changes in propositions P1, P2, and P3, We also need to add a new

Table 7: Version 2 - Constructs, Propositions, Explanations, and the Scope of the Dimension-Oriented Theory

Constructs	
C1	Functional Requirements Space
C2	Functional Requirements Classes (Problem Dimensions)
C3	System Functions
C4	Atomic Functional Requirements
C5	Data Output Functional Requirement Class
C6	Data Input Functional Requirement Class
C7	Event Trigger Functional Requirement Class
C8	Business Logic Functional Requirement Class
C9	Data Persistence Functional Requirement Class
C10	UI Navigation Functional Requirement Class
C11	External Call Functional Requirement Class
C12	Communication Functional Requirement Class
C13	User Interface Functional Requirement Class
C14	UI Logic Functional Requirement Class
C15	Data Validation Functional Requirement Class
C16	External Behavior Functional Requirement Class
C17	Core Functional Requirement Class (Core Problem Dimension)
C18	Non-Core Functional Requirement Class (Non-core Problem Dimension)
C19	Data Input Item
C20	Data Persistence Item
C21	Compound Requirement
Propositions	
P1	The *Functional Requirements Space* is composed of 14 known *Functional Requirements Classes*, including *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, *Data Persistence*, *UI Navigation*, *External Call*, *Communication*, *User Interface*, *UI Logic*, *Data Validation*, *External Behavior*, *Post-Condition*, and *Data Source*.
P2	The Nine functional requirements classes of *Data Output*, *Data Input*, *Event Trigger*, *Business Logic*, *Data Persistence*, *Data Validation*, *User Interface*, *User Interface (UI) Logic*, and *User Interface (UI) Navigation* belong to *Core Functional Requirement Classes*.
P3	The five functional requirements classes of *External Call*, *Communication*, *External Behavior*, *Post-Condition*, and *Data Source* belong to the *Non-core Functional Requirement Classes*.
P4	For every *Event Trigger*, there exists one corresponding *System Function*, *Compound Requirement*, or *Atomic Functional Requirement*.
P5	For every requirement of type *Data Validation*, there is a high probability that at least one corresponding requirement of type *Data Output* should also exist.
P6	For every *System Function*, there is a high probability that at least one requirement of type *Data Persistence* should also exist.
P7	For every *System Function*, there is a high probability that at least one requirement of type *Data Input* should also exist.
P8	For every *System Function*, there is a high probability that at least one requirement of type *Data Output* should also exist.
P9	For every *System Function*, there is a high probability that at least one requirement of type *Business Logic* should also exist.
P10	For every *Data Input Item*, there is a high probability that at least one corresponding *Data Persistence Item* should also exist.
P11	For every *Data Input Item*, there is a high probability that at least one corresponding requirement of type *Data Validation* exist.
P12	For every *System Function*, there exists at least one corresponding *Event Trigger*.
P13	For every violation of a business rule, there exists a system reaction as a corresponding compound requirement, consisting of a set of atomic requirements.

Table 8: Table 7 Continued

Explanations	
E1	The functional requirements space in the domain of enterprise applications can be logically divided into two partitions: (a) core functional requirements classes and (b) non-core functional requirements classes. As of yet, there are 9 known core and 5 known non-core classes of functional requirements. The sets of of core and non-core classes of functional requirements will be updated with future observations and discoveries. See E2 and E3 as to why core and non-core classes exist in the domain.
E2	Enterprise software systems, in order to support their corresponding business processes in the real-world, frequently need to run system functions (through Event Triggers), which collect information (Data Input), check the validity of inputted data (Data Validation), execute the prescribed business rules pertaining to the supported business process (Business Rules), store information as a result of executing the system functions (Data Persistence), and interact with users through displaying the results of executing the system functions, error messages, or success messages (Data Output). In GUI-based enterprise systems, Data Inputs and Data Outputs are tied to user interfaces; Inputs are collected through a user interface and system outputs are displayed on the user interface. These give rise to the three user interface-related classes of requirements including User Interface, User Interface Logic, and User Interface Navigation. The sequence of event trigger, input, data validation, business rule execution, output, and persistence, or a variation of this sequence, is very common in business systems, giving rise to the 9 core classes of functional requirements.
E3	In addition to the core functional requirements classes, enterprise systems occasionally need to support features that are not common in the domain, but specific to particular applications within the domain. Although compiling an exhaustive list of non-core requirements classes would require surveying every imaginable application in the domain and therefore an impossible task to carry on, in our studies so far, we have discovered 5 classes of non-core requirements.
E4	Event triggers, by definition, force the execution of either a single atomic requirement, a compound requirement, or a system function (e.g., a collection of atomic and compound requirements).
E5	When the execution of a data validation rule results in the detection of a violation of the corresponding correctness rules (i.e., detection of erroneous input), as a best practice, the user of the system needs to be notified, through appropriate error messages and possibly be given instructional messages, describing a course of action that can be taken to rectify the problem. Therefore, the existence of a requirement of type data validation, with a high probability, implies the existence of one or more related requirements of type data output.
E6	Most business services, and system functions as their implementations in enterprise systems, need to store information, typically collected from end users or generated as a result of a transaction with the end user. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data persistence aspects of the system function.
E7	Most business services, and system functions as their implementations in enterprise systems, need to collect information, typically from end users. Therefore, it is very likely that most system functions will have one or more statements of requirements, describing the data input aspects of the system function.
E8	From P7 and P11, we can deduce that it is very likely that a system function will have at least one requirement of type data validation. According to P5, the existence of data validation requirements creates a high probability for the existence of corresponding data output requirements. Therefore, from P7, P11, and P5, we can conclude P8. Moreover, System functions often need to notify their end users of the successful completion of the system function through requirements of type data output, giving rise to further opportunities for the existence of requirements of type data output in system functions.
E9	Most business services, and system functions as their implementations in enterprise systems, involve the execution of business rules. Therefore, it is very likely for system functions to have one or more statements of requirements, describing the related business rules.
E10	Business services, and system functions as their implementations in enterprise systems, are very likely to need to store information collected from end users to conduct business and complete transactions. Therefore, it is very likely for input data items to be stored in the system. From a specification point of view, input data items reappear in statements of data persistence requirements.
E11	To prevent data entry errors, when such errors are possible, as a best practice, system functions need to validate inputted data items before consumption and storage in the system. Therefore, data input items may need one or more relevant statement of requirements describing the data validation rules.
E12	All System Functions should be accessible through at least one Event Trigger.
E13	Systems need to respond to violations of the business rules. This response is specified in the form a set of requirements.
Scope	
S1	The Domain of Enterprise/Business Information Systems

explanation to explain the newly added proposition. We further need to remove Explanation E5 as a result of removing the Proposition P5 from the theory. The updated theory, which is shown in Tables 7 and 8, will serve as input to the remaining steps of the research and design methodology presented earlier in this paper (Steps 6 through 12), which aims to produce a fully validated and theory-backed software engineering process. These remaining steps will be the focus of a future work.

4 Conclusions and Directions for Future Work

The development of large industrial-strength software systems within acceptable economic and technical frames has remained a great challenge to this day as both pre-release and post-release software processes are, to a large degree, unpredictable, resulting in excessive rework, incurring not only cost and schedule overruns, but also quality issues. We put forward the argument that a fruitful avenue to address these challenges is to shift from current opportunistic software processes to ones that are more predictive and repeatable, that this shift can best be accomplished through developing software engineering theories with predictive power, and that focus on domain or application specific theories will make their resulting processes readily usable by software engineering practitioners. In line with these premises, we introduced a reusable step-by-step research and design methodology to develop more predictable software processes. We demonstrated the first half of the proposed methodology (Steps 1 through 6) through an expansive case study that aimed to develop a theory of the requirements space. In future work, we will demonstrate the second half of the process (Steps 7 through 12), producing a more effective requirements engineering method for the domain of enterprise systems. We reported results from several empirical studies to support the arguments put forward in this paper.

The ideas put forward in this paper can be continued in several directions. A great avenue to continue this research is to apply the proposed design methodology to various software engineering areas and to different software domains, such as scientific computations [21], agent-based systems [37], Geographical Information Systems (GIS) [39], Intelligent systems [47], and embedded systems, to devise more effective software processes.

References:

[1] V. R. Basili, S. Chang, J. Gannon, E. Katz, N. M. Panlilio-Yap, C. L. Ramsey, M. Zelkowitz, J. Bailey, E. Kruesi, and S. Sheppard, Monitoring an ada software development, *ACM SIGAda Ada Letters*, July-August 1984, 4(1):3239.

[2] V.R. Basili, Editorial, Empirical Software Engineering, 1(2), 1996.

[3] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional, 1999.

[4] E. Bersoff, V. Henderson, and S. Siegel, Software configuration management, Prentice Hall, 1980.

[5] B. Boehm, Software and its impacts: A quantitative assessment, *Datamation*, May 1973, 9:4859.

[6] B. Boehm and V.R. Basili, Software Defect Reduction Top 10 list, IEEE Computer, vol. 34, no. 1, pp. 135-137, 2001.

[7] M. Bunge, Scientific Research: The Search for a System, Springer-Verlag, New York, 1967.

[8] J. Carroll and P.A. Swatman, Structured Case: A Methodological Framework for Building Theory in Information Systems Research, *European Journal of Information Systems*, 9:235-242, 2000.

[9] J. Cleland-Huand, C. Chang, and M. Christensen, Event-based traceability for managing evolutionary change, in *IEEE Transactions on Software Engineering*, Sep. 2003, 29(9):12261242.

[10] B. Cohen, Developing Sociological Knowledge: Theory and Method, 2nd Edition, Belmont, CA, Wadsworth Publishing, 1989.

[11] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard, Lassie: A knowledge-based software information system, in *Communications of the ACM*, 1991, 34(5):3449.

[12] A. Endres and D. Rombach, A Handbook of Software and Systems Engineering, Empirical Observations, Laws, and Theories, Fraunhofer IESE Series on Software Engineering, Pearson Education Limited, 2003.

[13] R. Fjelstad and W. Hamlen, Application program maintenance study - report to our respondents, Technical Report, IBM Corporation, DP Marketing Group, 1986.

[14] A. Ghazarian, A Design-Rule-Based Constructive Approach to Building Traceable Software, Ph.D. Thesis, University of Toronto, 2009.

[15] A. Ghazarian, A Case Study of Source Code Evolution, in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, Kaiserslautern, Germany, IEEE Computer Society, March 2009, pp. 159-168.

[16] A. Ghazarian, Coordinated Software Development: A Framework for Reasoning about Trace Links in Software Systems, in *Proceedings of the IEEE's 13th International Conference on Intelligent Engineering Systems (INES 2009)*, IEEE Computer Society, Barbados, April 2009, pp 236-241.

[17] A. Ghazarian, A Case Study of Defect Introduction Mechanisms, in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE 2009)*, Springer, Lecture Notes in Computer Science (LNCS), Amsterdam, The Netherlands, June 2009, pp. 156-170.

[18] A. Ghazarian, Effects of Source Code Regularity on Software Maintainability: An Empirical Study, in *Proceedings of the IASTED International Conference on Software Engineering and Applications (SEA 2010)*, Marina del Rey, USA, November 2010.

[19] A. Ghazarian, M. Sagheb-Tehrani, A. Ghazarian, A Software Requirements Specification Framework for Objective Pattern Recognition: A Set-Theoretic Classification Approach, in *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (CECCS 2011)*, Las Vegas, USA, April 2011, pp. 211-220.

[20] A. Ghazarian, A Probabilistic Mathematical Model to Measure Software Regularity, in *Proceedings of the 15th IASTED International Conference on Software Engineering and Applications (SEA 2011)*, Dallas, USA, 2011.

[21] A. Ghazarian, A Domain-Specific Architectural Foundation for Engineering of Numerical Software Systems, *WSEAS Transactions on Systems*, No. 7, Vol. 10, pp. 193208, July 2011.

[22] A. Ghazarian, Characterization of Functional Software Requirements Space: The Law of Requirements Taxonomic Growth, in *Proceedings of 20th IEEE International Requirements Engineering Conference (RE'2012)*, Chicago, USA, September 2012.

[23] R. B. Grady, Practical software metrics for project management and process improvement, Prentice Hall, 1992.

[24] D.J. Herbsleb and A. Mockus, Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering, ACM SIGSOFT Software Engineering Notes, 28(5):138-147, 2003.

[25] A.R. Hevner, and S.T. March, and J. Park, and S. Ram, Design Science in Information Systems Research, MIS Quarterly, Vol. 28, No. 1, March 2004, pp. 75-105.

[26] IEEE Recommended Practice for Software Requirements Specification, IEEE Std 830, 1993.

[27] I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999.

[28] M. Jorgensen and D.I.K. Sjoberg, Generalization and Theory Building in SOftware Engineering Research, in *Empirical Assessment in Software Engineering (EASE2004)*, IEE, 2004, pp. 29-36.

[29] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg, Preliminary Guidelines for empirical Research in Software Engineering, IEEE Transactions on Software Engineering, 28(8):721-734, 2000.

[30] P. Kruchten, The Rational Unified Process: An Introduction, 2nd Edition, Addison Wesley Longman Inc., 2000.

[31] L.P.W. Land, B. Wong, and R. Jeffery, An Extension of the Behavioral Theory of Group Performance in Software Development Technical Reviews, in *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, 2003, pp. 520-530.

[32] K. Lewin, The Research Center for Group Dynamics at Massachusetts Institute of Technology. Sociometry, 8:126-135, 1945.

[33] B.P. Lientz, and E.B. Swanson, Software Maintenance Management, Addison-Wesley, 1980.

[34] C.E. Lindblom, Alternatives to Validity. Some Thoughts by Campbell's Guidelines: Creation, Diffusion, Utilization, 8:509-520, 1987.

[35] R.K. Merton, Social Theory and Social Structure, 3rd Edition, The Free Press, Ney York, 1978.

[36] A. Mockus and L. G. Votta, Identifying reasons for software changes using historic databases, in *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, 2000, pp. 120130.

[37] F. Neri, Empirical investigation of word-of-mouth phenomena in markets: a software agent approach, *WSEAS Transactions on Computers*,WSEAS Press (Wisconsin, USA), issue 8, vol. 4, pp. 987-994, 2005.

[38] J. T. Nozek, and P. Palvia, Software Maintenance Management: Changes in the Last Decade, *Journal of Software Maintenance: Research and Practice*, 1990, Vol. 2, No. 3, 157-174.

[39] R. Pulavarthi and A. Ghazarian, An Interactive Network of Events with Geographic Perspective, *WSEAS Transactions on Information Science and Applications*, No. 12, Vol. 9, pp. 369-378,World Scientific and Engineering Academy and Society, December 2012.

[40] Rational Software Inc., RUP - www.rational.com.

[41] D. Sandborg, Mathematical Explanation and Theory of Why-Questions, The British Journal for the Philosophy of Science, 49(4):603-624, 1998.

[42] C. Sauer, D.R. Jeffery, L. Land, and P. Yetton, The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research, IEEETransactions on Software engineering, 26(1), 2000, pp. 1-14.

[43] K. Schwaber and M. Beedle, Agile Software Development with Scrum, Prentice Hall, 2001.

[44] D.I.K. Sjoberg, T. Dyba, B.C.D. Anda, and J.E. Hannay,Building Theories in SOftware Engineering, Guide to Advanced Empirical Software engineering, F. Shull et al. (eds.), Springer, 2008, pp. 312-336.

[45] M.S. Tehrani and A. Ghazarian, Software Development Process: Strategies for Handling Business Rules and Requirements, Journal of ACMSIGSOFT, Software Engineering Notes, Volume 27, Issue 2, pp. 58-62, March 2002.

[46] W.F. Tichy, Should Computer Scientists Experiment More? 16 Excuses to Avoid Experimentation, IEEE Computer, 31(5):1998, pp. 32-40.

[47] T-S Tsay, Intelligent Guidance and Control Laws for an Autonomous Underwater Vehicle, *WSEAS Transactions on Systems*, Issue 5, Volume 9, pp. 463-475, May 2010.

[48] B. Van Fraassen, The Scientific Image, Oxford University Press, New York, 1980.

[49] H. V. Vliet, Software Engineering: Principles and Practices, 2000, John Wiley & Sons.

[50] K.E. Weick, Theory Construction as Disciplined Imagination, Academy of Management Review, 14(4):490-531, 1989.

[51] R.K. Yin, Case Study Research: Design and Methods, Sage Publications, Thousand Oaks, CA, 1984.