8-7-2012

A Requirements-Based Exploration of Open-Source Software Development Projects – Towards a Natural Language Processing Software Analysis Framework

Radu Vlas
Georgia State University

PERMISSION TO BORROW

In presenting this dissertation as a partial fulfillment of the requirements for an advanced degree from Georgia State University, I agree that the Library of the University shall make it available for inspection and circulation in accordance with its regulations governing materials of this type. I agree that permission to quote from, to copy from, or publish this dissertation may be granted by the author or, in his/her absence, the professor under whose direction it was written or, in his absence, by the Dean of the Robinson College of Business. Such quoting, copying, or publishing must be solely for the scholarly purposes and does not involve potential financial gain. It is understood that any copying from or publication of this dissertation which involves potential gain will not be allowed without written permission of the author.

RADU E. VLAS

NOTICE TO BORROWERS

All dissertations deposited in the Georgia State University Library must be used only in accordance with the stipulations prescribed by the author in the preceding statement.

The author of this dissertation is:

RADU E. VLAS
Computer Information Systems Department
35 Broad St., NW
Georgia State University
P.O. Box 4015
Atlanta, Georgia 30302-4015

The director of this dissertation is:

WILLIAM N. ROBINSON
Computer Information Systems Department
35 Broad St., NW
Georgia State University
P.O. Box 4015
Atlanta, GA 30302-4015

A REQUIREMENTS-BASED EXPLORATION OF OPEN-SOURCE SOFTWARE DEVELOPMENT PROJECTS – TOWARDS A NATURAL LANGUAGE PROCESSING SOFTWARE ANALYSIS FRAMEWORK

BY

RADU EDUARD VLAS

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

Of

Doctor of Philosophy

In the Robinson College of Business

Of

Georgia State University

GEORGIA STATE UNIVERSITY
ROBINSON COLLEGE OF BUSINESS
2012

ACCEPTANCE

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

H. Fenwick Huss, Dean

DISSERTATION COMMITTEE

Dr. William Robinson
Dr. Balasubramaniam Ramesh
Dr. Duane Truex
Dr. Walt Scacchi

ABSTRACT

A REQUIREMENTS-BASED EXPLORATION OF OPEN-SOURCE SOFTWARE DEVELOPMENT PROJECTS – TOWARDS A NATURAL LANGUAGE PROCESSING SOFTWARE ANALYSIS FRAMEWORK

BY

RADU EDUARD VLAS

JULY 2012

Committee Chair: Dr. William N. Robinson

Major Academic Unit: Computer Information Systems

Open source projects do have requirements; they are, however, mostly informal, text descriptions found in requests, forums, and other correspondence. Understanding such requirements provides insight into the nature of open source projects. Unfortunately, manual analysis of natural language requirements is time-consuming, and for large projects, error-prone. Automated analysis of natural language requirements, even partial, will be of great benefit. Towards that end, I describe the design and validation of an automated natural language requirements classifier for open source software development projects. I compare two strategies for recognizing requirements in open forums of software features. The results suggest that classifying text at the forum post aggregation and sentence aggregation levels may be effective. Initial results suggest that it can reduce the effort required to analyze requirements of open source software development projects.

Software development organizations and communities currently employ a large number of software development techniques and methodologies. This implied complexity is also enhanced by a wide range of software project types and development environments. The resulting lack of consistency in the software development domain leads to one important challenge that researchers encounter while exploring this area: specificity. This results in an increased difficulty of maintaining a consistent unit of measure or analysis approach while exploring a wide variety of software development projects and environments. The problem of specificity is more prominently exhibited in an area of software development characterized by a dynamic evolution, a unique development environment, and a relatively young history of research when compared to traditional software development: the open-source domain. While performing research on open source and the associated communities of developers, one can notice the same challenge of specificity being present in requirements engineering research as in the case of closed-source software development. Whether research is aimed at performing longitudinal or cross-sectional analyses, or attempts to link requirements to other aspects of software development projects and their management, specificity calls for a flexible analysis tool capable of adapting to the needs and specifics of the explored context. This dissertation covers the design, implementation, and evaluation of a model, a method, and a software tool comprising a flexible software development analysis framework. These design artifacts use a rule-based natural language processing approach and are built to meet the specifics of a requirements-based analysis of software development projects in the open-source domain. This research follows the principles of design science research as defined by Hevner et. al. and includes stages of problem awareness, suggestion, development, evaluation, and results and conclusion (Hevner et al. 2004; Vaishnavi and Kuechler 2007). The long-term goal of the research stream stemming from this dissertation is to propose a flexible, customizable, requirements-based natural language processing software analysis framework which can be adapted to meet the research needs of multiple different types of domains or different categories of analyses.

Table of Contents

Keywords

Requirements engineering, requirements lifecycle, requirements processes, requirements discovery, requirements classification, natural language processing, text mining, open-source, open-source software development, software development project success, requirements innovation.

1. Introduction

The increased attention the open source (OS) phenomenon received in the last over 20 years and the increased market penetration ability the OS software (OSS) products showed throughout this timeframe has attracted the interest of researchers (Mockus et al. 2002; Hippel and Krogh 2003). Open-source software development projects can be identified in a wide spectrum of domains. Open-source researchers identified OS software development (OSSD) and OS adoption efforts in areas such as Internet or Web infrastructure, networked computer games, higher education, military computing, and bioinformatics to name only a few (Scacchi and Alspaugh 2008; Scacchi et al. 2009). Research communities started to explore open-source related products and processes in domains such as economics, law, public policy, geography, art, anthropology, physics, organization science, biology, management, and information systems (Scacchi et al. 2009). OSSD appears to produce high quality software products with fewer resources and less complex development processes and organizational structures than more traditional approaches. In spite of the intrinsic differences between OS and closed source software development, the natural similarities between the two software development paradigms justifies the possibility of expanding the coverage and applicability area of findings and artifacts from one to the other. Consequently, an analysis of OSS management, membership, development lifecycles, and products may lead to improvements in all software development. Studies of software requirements provide techniques and procedures for systematically analyzing the software

development phenomenon. Therefore, this study adopts a requirements-centered perspective on OSSD and creates artifacts to aid in the automated analysis of OSSD projects and products. This thesis represents an initial stage during which I build the grounds required for designing, developing, evaluating, and proposing a more general software analysis framework with applicability in all software development and related environments.

Current software development is characterized by a significant increase in complexity in most of the areas of the software development lifecycle (SDLC). Most of the modern software products are large and complex. Enterprise-wide software solutions of significant complexity are widely spread in the business domain while the personal use of computing solutions is increasingly dominated by multi-generation software and integrated and embedded systems. In the context of these increasing complexities, automation of analysis and evaluation artifacts becomes critical. The automation of requirements-based artifacts is especially important because the complexity of current software systems is directly mirrored in the early SDLC phases (where requirements play a major role) of the projects developing such software.

Requirements are justified by the underlying goals that contribute to their creation (Sommerville and Sawyer 1997). In the context of OSSD, developers are also expected users of the product being developed. They are the stakeholders expressing the needs that define these goals. Consequently, it may appear that the requirements analysis stage is absent, given that requirements are generally understood by the developers (Fitzgerald 2006). Nonetheless, Scacchi has identified *software informalisms*, which are "the information resources and artifacts that participants use to describe, proscribe, or prescribe what's happening in a OSSD project" (Scacchi 2009). Scacchi identifies two dozen types of unstructured software informalisms, which include chats, email, forums, project digests,

etc. By analyzing these natural language (NL) artifacts, one can better understand the requirements, and thus the OSSD phenomenon.

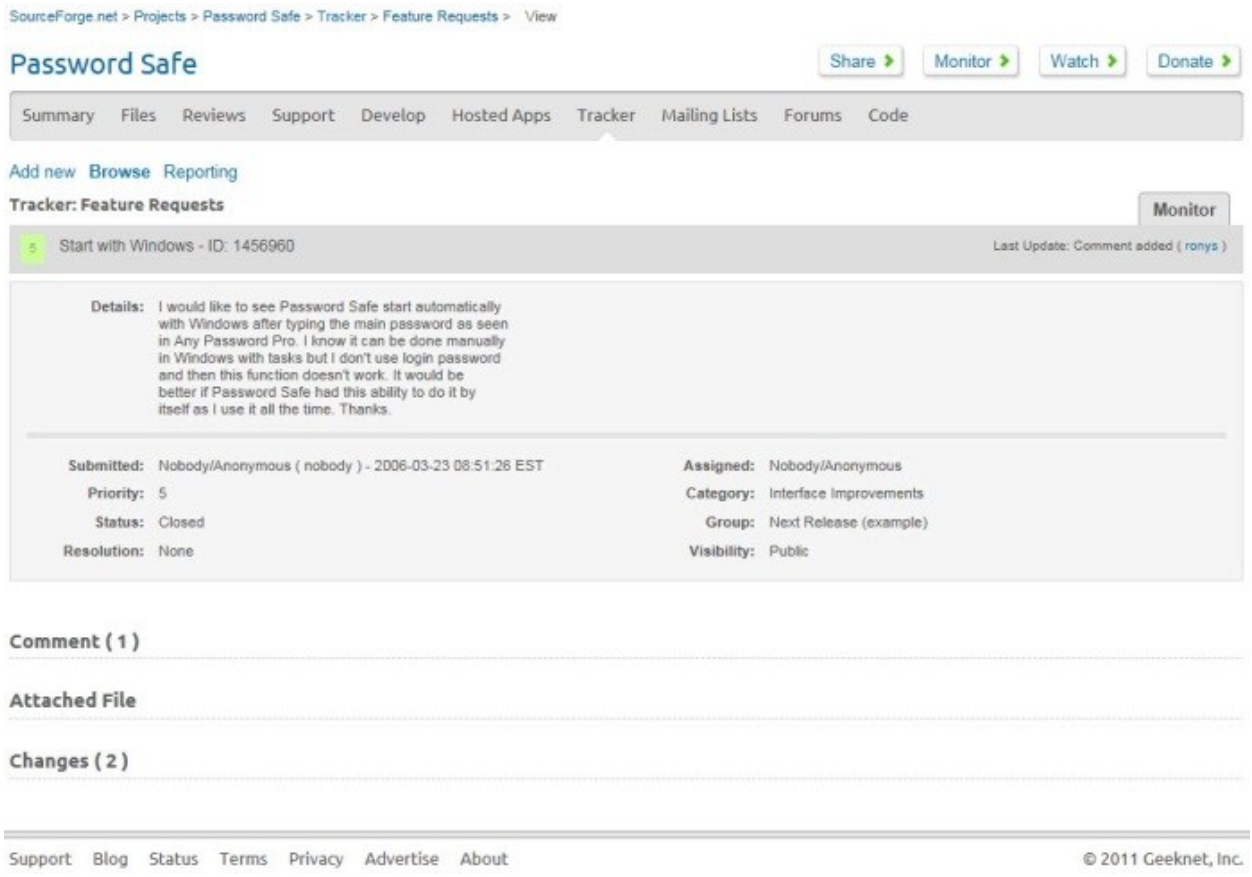

Figure 1. Password Safe feature requests from SourceForge's Tracker.

Consider Figure 1, which presents a feature request, a kind of requirement, from the Feature Tracker of the Password Safe project on SourgeForge. The Password Safe project has 630 feature requests, 976 bug reports, and thousands of forum posts. In this study, a feature request refers to a desired piece of functionality of the system being built. These feature requests can be found in feature requests forums where other types of communication (technical writing, social conversation elements, stories, etc.) are also present. Obviously, the communication associated with various OSSD projects is expected to contain different amounts and types of feature requests and, consequently various

amounts and types of requirements. OSS feature requests forums are software informalisms where both future users of the software and current developers post. It is expected that these two categories of OSS contributors generate distinct types of posts. However, this study does not differentiate between user generated and developer generated feature requests. To comprehensively understand the OSSD phenomenon, researchers need to analyze such data, to identify communication patterns, to discover and classify processes and various elements of communication. The natural language informalisms identified by Scacchi are used to manage projects. Their analysis provides insights into the best practices of OSSD.

1.1. The Importance of Requirements and of Requirements Traceability

The concept of requirements traceability has been defined as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" (Gotel and Finkelstein 1994). Requirements traceability is a guiding theory in the software development line of research (Gotel and Finkelstein 1994; Ramesh et al. 1997). The literature on requirements traceability provides a number of specific suggestions for performing efficient requirements traceability (Hayes et al. 2006). By following the lifecycle of requirements, we can follow techniques and infer the strategies of development. Ideally, we can discover practices that distinguish successful projects from the unsuccessful ones.

Requirements can be considered from three simple levels:

1. *Requirements metrics* count individual requirements, including their total number, individual versions, their classifications, and their trace links (types and

numbers, including contribution structures (Gotel and Finkelstein 1997) e.g., roles, relationships, responsibilities)

2. *Requirements management metrics* count collections of requirements, including requirements snapshots (the collective versions), their temporal relationships, and trace links (e.g., a snapshot's association to a code release)
3. *Requirements management model* (RMM) specifies the associations among artifacts as well as the process model for the creation and management (Ramesh and Jarke 2001).

Advanced developers apply all three levels to development (Ramesh 1998). The ultimate goal of the research stream starting with this study is first to build successively more capable tools for discovering these concepts from the natural forms found in OSSD, and second to build successfully more flexible analysis artifacts that exhibit applicability in all software development and related environments. In so doing, I aim to understand the evolving requirements management models of OSSD, and to employ flexible and powerful techniques that are customizable to various datasets and environments.

Precise, empirically derived OSSD RMMs describe current practices, and suggest best practices for software requirements management. If we had OSSD RMMs at this moment, we would have been able to answer questions such as the following:

- What types and numbers of requirements are associated with various types of projects (successful, high quality, secure, etc.)?
- What are common ratios concerning requirements numbers and types, resources, and release rates?

- How do the types of requirements vary with the size or type of software being developed?
- For a given size or type of software product, which RMMs have the highest chance of being most successful?

The research presented here represents the initial steps towards performing automated software requirements discovery and classification, and provides the results of an exploratory study of OSSD projects based on these analysis techniques. Given an OSSD project, the developed artifact applies a natural language parsing approach to:

- Identify requirements
- Classify requirements

Application of requirements traceability practices improves the likelihood of software project success, specifically through improved quality, functionality, and timely releases. Traceability links record the history of artifact development, from high-level requirements descriptions to the lower-level programming codes. From the artifact traces, one can infer the development tasks used to construct the artifacts. A trace link from source code to binary code implies the task of applying a compiler. A trace link from feature requirement to use case implies the task of use case specification. Development traces indicate the development process model used. Sometimes developers apply a process model that varies from their stated process model. Analyzing development traces reveals the actual process model used. A software tool to aid the discovery, modeling, and analysis of development traces will help in analyzing software development. If such a tool were to exist, then it could be applied to existing software projects to aid empirical studies. For example, according to information systems development (ISD) theory, development process models vary in their success – some process models work well and others less so

depending on a variety of factors, such as project type, personnel, etc. A tool for discovery, modeling, and analysis of artifact traces will help improve ISD process theory. Discovered models can be analyzed according to a variety of success factors, such as quality, functionality, and release patterns.

Requirements discovery is a prerequisite to developing requirements traceability tools. To date, there is no software tool that can identify and classify requirements from open-source software informalisms. Open issues include: (1) how is a single requirement recognized and delimited?, (2) given the text of a requirement, how can it be classified?, and (3) how is a recognized requirement related to another recognized requirement? Solving these problems will provide for a software tool that can automatically review natural text documents and create identified requirements, their classification, and associated traceability links. Such a software tool is a prerequisite to a comprehensive tool for discovery, modeling, and analysis of development traces, which in turn will support empirical analysis of ISD process theory. This study proposes a solution to the first two issues above and provides the prerequisites for developing a solution to the third problem.

With this introduction, I now present a design-science hypothesis for natural language requirements discovery:

H1: The automated discovery and classification of requirements contained within software informalisms of Open-Source Software Development projects can be achieved through the design of a requirements analysis process and the development of a software artifact to implement it.

Related research suggests that grammar-based analysis may provide efficient techniques for the analysis of natural language data. Ideally, a requirements recognizer should work

for any natural language document. However, Natural Language Processing (NLP) analyses perform better when they are customized to a document corpus in which the language is applied using common forms or idioms. Therefore, the refined design-science hypothesis below addresses the issues of a language with specialized sub-language as expressed by a sub-culture (such as OSSD):

H1.1: A multi-layered grammar, varying in domain specificity, can be constructed for the automated requirements discovery and classification of requirements contained within software informalisms of Open-Source Software Development projects.

I apply a design science approach to address these objectives. As with any design science study, a special attention is placed on how well the design works. Therefore, I employ a number of evaluation and validation methods among which I compare the results of the automated artifact with the results of an idealized perfect requirements recognizer.

1.2. Usage Scenarios

I present two usage scenarios for the developed artifact, Requirements Classifier for Natural Language (RCNL), in order to provide context. First, consider usage by an academic, Jane, studying OSSD. Jane can apply RCNL to OSSD projects to obtain metrics on the quantity and classification of requirements. Identifying text segments as requirements and their classifications are open to interpretation, even among experts. Thus, Jane may choose to review the requirements and the classifications produced by RCNL. She may even alter RCNL rules to suite her interpretation. Once satisfied with the results, Jane can compare project requirement metrics and correlate those metrics with success or other factors of interest. Second, consider John, an analyst for an OSS company. He can apply RCNL to the thousands of forum postings he receives monthly. By continually monitoring the quantity and classification of requirements posted, he

can maintain an overview of the kinds of concerns his users are expressing. In both scenarios, manual analysis of a vast amount of text is not practical. It is time-consuming and classification requires expertise, which is too costly. Software, such as RCNL, enables an analyst to get a quick overview of OSS requirements.

Herein, I describe the design and engineering of two versions of RCNL implementing two distinct strategies (Section 3 and Section 4), and experiments measuring their capabilities and applicability (Section 5). The results suggest that the parsing strategies and toolkit may be useful in classifying requirements in OSSD projects.

The automated requirements classifier is aimed at helping researchers discover patterns and trends in OSS development. By reviewing many projects with a classifier, a research can gain a perspective on what kinds of requirements are common. Such observations can be correlated with other project factors, such as project success, timeliness, or quality. This may lead to advice of this form: "many successful OSS projects for embedded systems include requirements classified as X." Requirements classification may eventually be applied to a project during development. A project manager may discover, through classification, that there are no requirements of type X (e.g., security). With such knowledge, the project manager can seek remedial action.

1.3. NL Requirements Discovery

OSSD requirements take many forms, most of which are represented as natural language text within software informalisms (Scacchi 2009). For each form, there are many requirements. For example, the KeePass Password Keeper project has 1,522 feature requests, 923 bug reports, and thousands of other various forum posts. Cleland-Huang et. al. found that software informalisms are filled with thousands of requirements, as well as thousands of lines that are not requirements

– for example, social communications, code segments, slang, typos, formatting elements etc. (Cleland-Huang et al. 2006; Scacchi 2009). Therefore, requirements discovery in this context is first about delimiting each requirement within its source. Once requirements are identified, then subsequent processing can begin.

Consider three strategies for recognizing software requirements:

1. *Grammar-based strategy*: The text of the specific software informalism providing the data is parsed according to a grammar. The grammar defines what text within a sentence is a software requirement. For example, a Subject-Action-Object (SAO) grammar would tag each SAO triple at a sentence level as a requirement. This strategy implements the patterns commonly characterizing formal requirements specifications in which each requirement is expected to clearly state a subject, an action, and an object. The subject is the actor in the requirements statement. The action determines the feature being described in the requirement. The object is the entity being impacted by the action performed by the actor. For example, let's consider the following requirement statement from the phpMyAdmin project: "`when […], you should flush [the] table, because […].`" In this example, the subject is "`you`" which is used to denote the user of the software product. The action that should take place is "`should flush`" and the object impacted by the action is the "`table`." The elements of text preceding and succeeding this SAO pattern provide additional explanatory context for the action of the requirement. I consider a grammar-based strategy to provide a targeted, within sentence approach to requirements discovery in NL data.
2. *Delimiter-based strategy*: As denoted in the example associated with the grammar-based strategy above, often an SAO triple is accompanied by explanatory expressions which

place the action of the statement in a specific context. Moreover, many times the ideas that comprise the informal communication on the desired features of a software product are described over more than one sentence or even paragraph. Therefore, comprehensively capturing the entire context of a requirement expressed in informal communication calls for a parsing strategy that crosses over sentence boundaries. In this strategy, the text is split into segments according to delimiters, which may be keywords or expressions. The text between the delimiters is tagged as a requirement. Each post in a Feature Request forum in OSSD commonly addresses one or a small number of ideas or suggestions. The posts often include a variety of sentences and phrases providing context to the idea(s) presented. Sometimes, the feature being suggested is not clearly specified but rather implied or described without being ever expressed in a concise statement. Given these facts, the delimiter strategy considers each forum posting as a discussion around a limited number of features of interest separated by specific delimiters. Therefore, each Feature Request post is a requirement if no delimiters are identified within the post. The identified delimiters determine the number of delimited requirements present in a Feature Request post.

3. *Hybrid strategy*: The grammar-based strategy and the delimiter-based strategy represent distinct approaches and at a different level of detail. Both strategies yield valuable results and, in order to comprehensively analyze requirements expressed in software informalisms, one should devise a strategy combining them together. I call such a strategy a hybrid strategy. First, the delimiter-based approach is applied. Then, each requirement text is parsed with the grammar-based approach. This allows for the recognition of an aggregate requirement (the result of a delimiter-based analysis) and its

supporting sub-requirements (the results of a grammar-based analysis). Of course, if the text includes hierarchal requirements numbers, e.g., 1, 1.1, 1.1.1, then such numbers can be used for requirements groupings – unfortunately, this is less common in the open source domain.

Unrecognizable text affects how each strategy performs. If the grammar-based approach completely characterizes the informal text, then grammar-based and delimiter-based strategies will tag the same text segments as requirements. More commonly, the grammar-based tagging only partially characterizes the text. Thus, the unrecognized text, preceding or following an SAO triple for example, will not be tagged as being within a requirement. Using the delimiter-based strategy can provide a more natural tagging, as recognized by analysts. Finally, the hybrid strategy provides the best of both – an entire text segment tagged as a requirement, and its parts are characterized according to a grammar. Herein, I report on the results of performing requirements discovery and classification on text from Sourceforge's Feature Tracker with the grammar-based and the delimiter-based strategies and on a comparison between a two strategies. The development and evaluation of artifacts that implement the hybrid strategy is not covered here but considered for future research.

Figure 2 shows the result of applying a grammar-based parsing strategy using the developed artifact, Requirements Classifier for Natural Language (RCNL). The highlighted text has been parsed as grammar fragments, recognized as requirements, and classified according to a requirements ontology (Vlas and Robinson 2011). Notice that some text is not considered to be part of any requirement, and thus the text is not highlighted. The seemingly irrelevant text includes the feature identifier (numerical), the UNIX date the feature was posted (numerical), as well as elements of social communication such as

"Thanks in advance for the Developers consideration" (Note the presence of typos, and poor grammar – one of the prominent challenges in analyzing NL data from Open Source requirements).

1.4. NL Requirements Classification

Requirements engineering theory specifies measures that can guide the analysis of software development. For example, one would expect that the specification of a secure operating system would have many security requirements and of many different types. Their absence would be a cause for concern. Thus, requirements classification helps requirements management by determining the presence and proportion of various requirement types.

Requirements classifications provide taxonomies of common kinds of requirements. Reliability, efficiency, integrity, and usability are common requirements classes. Quality models, such as McCall (McCall et al. 1977), Boehm (Boehm et al. 1978), IEEE (IEEE), and ISO (ISO 2001; ISO 2011), specify the characteristics of requirements belonging to a class. The descriptions of these characteristics can be used to classify requirements.

McCall's software quality model is probably the most widely accepted model in both researcher and practitioner communities. McCall's model organizes a number of characteristics into a 2-level hierarchy of 23 criteria and 11 factors. The descriptions of these characteristics can be used to mine words and phrases that are indicative of requirements belonging to these characteristics. For example, a requirement that includes the word *faster* or *slow* is indicative of a performance requirement. Building on this perspective, the technique of keyword classification uses libraries of keywords, expressions, and grammar rules to match against delimited text elements (discovered requirements). Figure 2 shows and example of requirements classification. The elements of

text highlighted are first tagged as requirements based on one of the two implemented strategies and then classified if containing one or more of the items of the classification library. In Figure 2 there are few types of requirements highlighted, with some colors overlapping when a discovered requirement is classified multiple times.

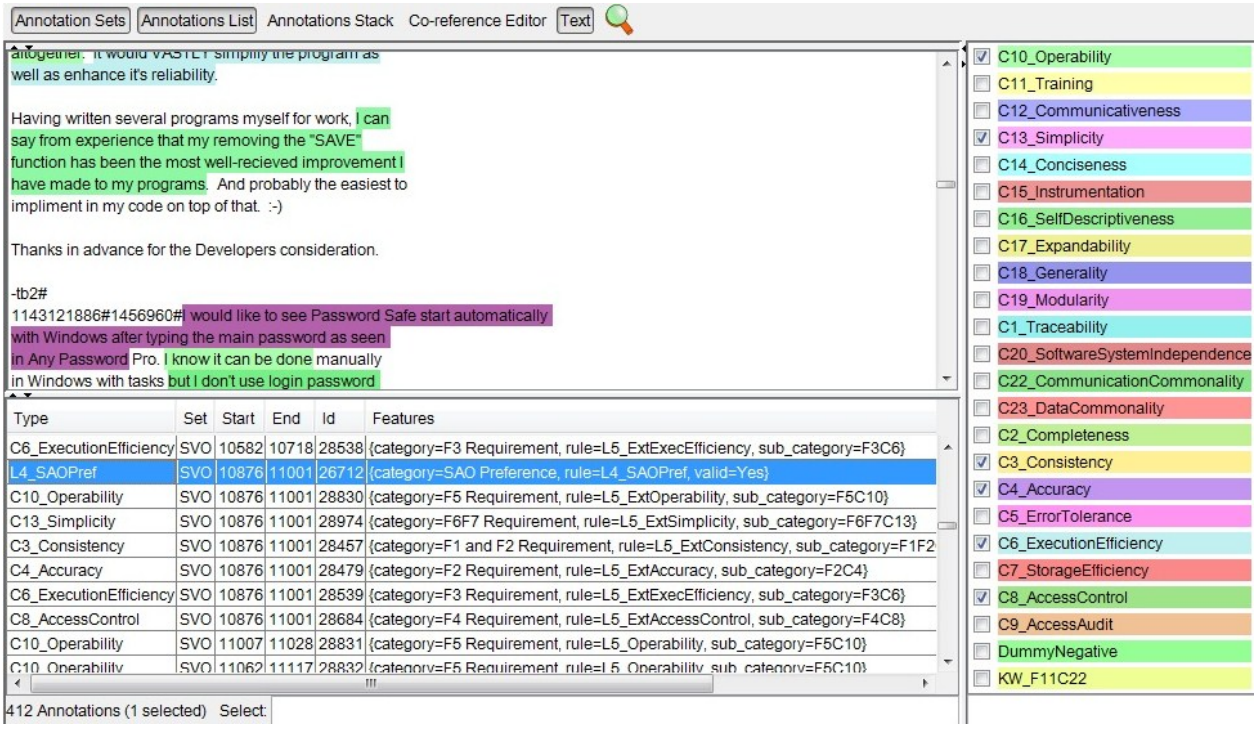

Figure 2. Grammar-based recognized Password Safe feature requirements with the classification color legend at the right.

The length of delimited requirements within text has a direct impact on the quality of the discovery and classification results and, consequently, determines performance criteria for the associated processes.

1. The shorter the word length of a recognized requirement the less likely the requirement will be classified, because classification is based on the contained keywords. This leads to many recognized but unclassified requirements and calls for an efficient classification method.

2. The longer the word length of a recognized requirement the more likely the requirement will be classified. At the same time, having longer recognized requirements leads to an increased chance of having two or more conceptually unrelated ideas considered as one requirement. This leads to fewer total recognized requirements, each of which is likely to be classified at least once. Such a situation calls for an accurate requirements discovery method.

The issues of requirements recognition (or delimiting) and classification are interrelated. This raises the issue of what exactly is a natural language requirement? Open source communication is extremely informal and, therefore, requirements include an unusual amount of extraneous language and symbols and improper syntax. Most importantly, what kinds of text should be considered as a requirement for the purpose of requirements management, including recognition and classification? In this study I address the questions of recognition and classification of requirements in open source communication.

1.5. An Emergent Grammar and Perspective

The work presented in this thesis aligns with a number of concepts from the linguistics domain and supports an emergent perspective of the OSSD phenomenon and of the OSS communication. Following a higher level view described by Chomsky, linguistics presupposes that a language is generated through the continuous application of the rules of a generative grammar, and that this process is characterized by the presence of two types of structures, deep structures and surface structures (Chomsky 1980; Chomsky 1986). Consequently, a language is never complete. It follows a perpetual route of construction and definition in which new elements are generated as a result of a socially informed generative process. Therefore, the Chomskyan linguist perceives the structure of language not as externally derived but as defined in the language user's mind (Truex and Baskerville 1998). It develops constantly through

communication, in real time (Auer and Pfänder 2011). The OSS language is temporal, emergent and determined as the outcome of disputes and continuous refining, in a similar manner to the way a culture is a continuously evolving entity. I adopt this point of view in my study by acknowledging the nature of the OSS language and the lexical generative transformations used by the prototypical language user in OSSD (the OSS contributor) for expressing desired functionality of the software system being constructed.

The academic literature in the field of information systems highlights an inherent dualism between technology and users. Therefore, information systems researchers often focus on the implied relationships between humans and objects of the surrounding world. This perspective is known as the Cartesian worldview and posits that humans understand and act based on mental representations of the objects in the world (Scada 2004). In my analysis of OSSD requirements and their presence in OSS informalisms I adopt a perspective consistent with the concept of emergent grammars and I associate that with both defining elements of the Cartesian perspective: technology as an object of the world, and users as the human component. OSS communication exhibits the characteristics of both a technology-determined software development discourse, as well as a participant-determined social discourse.

The concept of "emergent grammar" has been first described by Paul Hopper as being based on the fundamental assumption that structures are "unfinished and indeterminate." It captures the dynamics of an ongoing process of "languaging" (Hopper 1987; Hopper 1992; Hopper 1998; Hopper and Traugott 2003; Auer and Pfänder 2011). While describing the emergent systems viewpoint, Truex et. al. acknowledge the transition from the traditional perspective placing value on organizational and process stability to the more modern perspective that values flexibility, dynamicity, and agility (Truex et al. 1999). The goal set they present for information systems

development also applies to the OSSD phenomenon since it emphasizes principles that are generally accepted in open-source:

1. Lengthy analysis and design phases are regarded as poor investments. A higher emphasis is placed on experience and agility.
2. User satisfaction is never completely achievable as users and their needs change continuously.
3. The development and capture of abstract requirements (formalized as requirements documents in traditional closed-source software development) is neither possible nor attempted. Requirements are emergent as they continuously change and evolve along with the users who generate them.
4. Complete and unambiguous specifications are ineffectual because the evolving nature of specifications has to be acknowledged. Organizations and users change, thus any effort on providing a complete and unambiguous set of specifications would only result in a continuous investment of resources for reaching a continuously moving objective.
5. The lifespan of a software system is not predetermined or foreseeable. Instead, information systems evolve continuously to adapt to the changes in their environment.

Since language is regarded as an infinite set of sentences, there is an implied assumption that these sentences are generated by a grammar, which we call "generative grammar." Therefore, a generative grammar assigns structure to sentences. A good generative grammar is defined as one in which rules and lexical assignment descriptions are "rigorous and sufficiently explicit to determine how the sentences of the language are characterized by the grammar" (Chomsky 1980; Truex and Baskerville 1998). As described in the following sections of this thesis, I define and encode a generative grammar into the development of the analysis artifact.

In OSSD, grammar is an emergent entity that is constantly renegotiated by individual users of the system. Due to this ever changing nature of the grammar of OSS communication, I develop in this study a library of analysis rules (encoded patterns) for capturing the current state of this grammar. My efforts are based on the knowledge derived from our understanding of an "IS idealized stability" and complemented with patterns describing aspects of flexibility and agility. The agility of information systems development has been coined by Truex et. al. as "amethodical" (Truex et al. 2000). It supports conflict over consensus, recognizes the values of attending different voices and interests, accepts bargaining and negotiation as the main way to develop solutions, and values innovation as the means of achieving adaptability. All these attributes can also be used to explain and describe the OSSD lifecycle and in general the open-source approach.

The information systems literature assumes a synchronic perspective although it seems to be more an idealistic perspective than a truism. Here I implicitly follow same perspective by focusing on performance as defined in the Chomskyan approach (language use) and adopting the belief that it is possible to create a precise, unambiguous and concise language of science (Chomsky 1980; Truex and Baskerville 1998).

In linguistics research, Chomsky was the first to introduce the concepts of "deep structure" (D-structure) and "surface structure' (S-structure). The definitions of these two concepts have been borrowed and used in the ISD domain by numerous researchers. Deep structures are the sets of rules defining and describing the real-world and the way it functions. Therefore, deep structures can be regarded in ISD as a lower level concept that can also be used to encode a set of system requirements. In contrast to this, surface structures are a higher level concept that is

concerned with the interface between the software system and the organizational environment defined by its users (Chomsky 1986; Wand and Weber 1995; Truex and Baskerville 1998).

Surface structures are derived from deep structures through the application of transformational rules and define the area normally explored by systems analysts (Chomsky 1986). Deep structures are determined by a more fundamental set of rules, phrase structure rules. A number of information systems studies acknowledge that there currently is a higher focus on researching surface structures and recommend an increased effort to understand and capture the underlying concepts and knowledge that generate the surface structures. Specifically, they emphasize the need to address not only the "what" question, but also "why" and "how" in order to understand the reasoning and the underlying knowledge that defines deep structures (Leifer et al. 1994; Truex and Baskerville 1998). I address here this research call and perform an analysis of OSS communication that is intended to provide the tools and results for further exploring the OSSD phenomenon.

Following an emergent grammar perspective, the structure of OSS communication is "unfinished and indeterminate" (Hopper 1987; Auer and Pfänder 2011). Consequently, the aim of this thesis is not to discover the rules of this grammar (a futile exercise since they continuously evolve) but to discover and highlight a major part of the emergent grammar used in OSS communication, as it is characterized by its current state. This study provides an analysis that defines a number of structured utterances which are OSS specific and "cannot be explained entirely by the rules of canonical grammar" (Auer and Pfänder 2011).

1.6. Research Method

Much of information systems research follows the behavioral science paradigm, in which researchers aim to understand phenomena related to the development and use of IS. In recent years, the design science paradigm has grown. Design science researchers

develop IS artifacts and improve their performance. March and Smith specify four activities (theorize, justify, build, and evaluate) for conducting IS research, with behavioral science addressing the first two activities and design science addressing the last two (March and Smith 1995). The activities occasionally apply the same methods, such as a controlled experiment for behavioral theory justification and for design science artifact evaluation. The two paradigms are mutually supportive in that the results in one approach can provide for new research designs in the other (Cao et al. 2006).

Multi-paradigm research represents only about 20% of IS research studies in the 1990s (Mingers 2003). However, it has been argued that a multi-paradigm research provides for broader and more conclusive explanations (Cao et al. 2006). This project is design science research in that I justify, build and evaluate a designed artifact. The design stems from the general theories of IS development and traceability management, as well as the technical theories of concept tagging, grammar-based parsing, and classification. This study is a design science part of a larger multi-paradigm research project. Subsequently, the RCNL technology will be used to obtain data for a project-level exploration of success and quality in OSSD, to develop an analysis framework with applicability to all types of software development, and to develop versions of the analysis framework for other domains. The goals of this study and of the immediately following studies are to extend IS development process theory to explain the unique characteristics of OSSD.

This study follows the design science approach defined by Hevner et. al. in developing the RCNL classifier (Hevner et al. 2004). Design science, as explained by Hevner et. al., provides seven guidelines for research on designed artifacts. I apply the Hevner et. al. descriptive approach to design science – developing and then evaluating the design of RCNL using case

studies, performance analysis, and argumentation. Additionally, I compare the results of automated discovery and classification with those generated by an expert.

I have designed, developed, evaluated and applied the RCNL for text-based requirements analysis. A key element of RCNL is its multi-level ontology, in which the lower, specific levels apply generic English grammar-based concepts while the upper, abstract levels apply OSS-specific requirements-based concepts. The RCNL's flexibility and ability to be adapted to the specifics of multiple analysis domains resides in its multi-layered architecture. The empirical application of the RCNL is limited in this study to the text found in work items of SourceForge's Feature Tracker which includes a large number of forum posts (similar to Bugzilla or Jira) (Lintula et al. 2006).

This research study provides five contributions:

1. A grammar-based design of software automation for the discovery and classification of natural language requirements
2. Two complementary parsing schemes implemented within the design
3. Requirements discovery, classification, and analysis of 30 OSSD projects
4. A project-level requirements-based exploratory analysis of OSSD projects
5. A wave theory of requirements innovation

Together, these contributions provide a path for subsequent studies of OSS requirements and enable subsequent software tools facilitating automation of requirements traceability analysis in support of IS development process studies.

2. Related Research

This chapter provides a detailed background review of requirements in the context of open source software development, an overview of pattern-based analysis used in studies of

requirements engineering, concepts of requirements discovery and classification, and perspectives on software product quality and on software development project success.

2.1. Requirements and Requirements Processes in Open-Source

Requirements represent an essential component of any software development project. Requirements lifecycle analysis is a challenging task mainly because requirements evolve continuously along with the evolution of the software development project. At the very bottom of this continuous change lie social factors (stakeholders' different views) and technical factors (production constraints, usage experience, or feedback received) (Anderson and Felici 2002). In spite of a consistent body of research in this area, requirements evolution models and analysis methodologies are still providing an incomplete solution to the problem. There are two challenges defining this situation: first, the large quantity of longitudinal data that must be collected and analyzed and second, the lack of methodologies to provide real-time support for analyzing requirements evolution (Jarke and Paulk 1994; van Lamsweerde 2000). If these circumstances can be overcome, understanding requirements and their evolution can become a major step towards identifying rare environmental events or towards providing strategies to deal with environmental changes (Lutz and Mikulski 2003).

In the context of open-source, the common belief is that OSSD projects lack requirements and their associated requirements processes, at least to the extent seen in closed-source software development projects. However, OSSD projects do have requirements and requirements processes. Open-source requirements are mostly informal, lacking a clear, formal organization of information in well-structured documents (Scacchi 2002; Jensen and Scacchi 2004). Even if sometimes requirements are clearly described, often times they may be only inferred from or suggested in text descriptions found in change requests, bug fixes requests, forums, blogs, email

exchanges, and other types of electronic communication. Therefore, the open-source requirements emerge as the result of a dynamic social process of communication among open-source participants. They go through an evolutionary process of iterative review and improvement. In this process requirements become generally accepted by all participants and they start to be regarded as "set" requirements. Further discussion among participants on an accepted requirement usually results in another step in the refining of its details. No efforts to document, formalize, or substantiate accepted requirements as formal system requirements have been identified (Scacchi 2009). The informal component of OS requirements is confirmed by Lintula who acknowledges the importance of discussion forums as a means of reaching common understanding and acceptance on open-source requirements (Lintula et al. 2006).

The lack of standardization exhibited by requirements processes and requirements specification in open-source accounts for, among other factors, the difficulty of analyzing this domain and the relative scarcity of research studies exploring this area. Consequently, an in-depth understanding of the nature of OSSD projects and of the presence and evolution of requirements throughout the entire project lifecycle is limited.

In the process of better understanding the OSS product development, some researchers focused on defining and analyzing requirements (Scacchi 2002; Noll 2008). Research studies exploring development processes in open-source concluded that there are no formal processes similar to the ones closed-source development employs in requirements engineering. However, Scacchi identified informal processes similar to requirements elicitation, analysis, specification, validation, and management (Scacchi 2002; Scacchi 2006; Scacchi 2009). These processes have a significant social component and define the general characteristics of requirements lifecycle in open-source. In many cases, open-source software projects adopt a post-hoc approach in which

requirements take shape after their corresponding implementation in the developed product is realized.

The ability to better understand open-source requirements and their lifecycle provides insights into the nature and evolution of open-source projects. Unfortunately, the informal nature of open-source requirements makes a manual analysis of the natural language text used in open-source requirements not feasible due to being time-consuming and error-prone, especially in the case of large projects.

A model to provide a consistent perspective on requirements and their components, and an associated method for automated analysis of NL requirements are necessary in order to achieve a comprehensive understanding of and an improved management of open-source requirements and software development projects. In this research I design, develop, evaluate and apply an automated (requirements-based) tool for the analysis of software development projects, which incorporates a model and a method as delineated above. The model, the method, and the associated tool are designed for the specifics of OSSD projects but their external validity and level of flexibility allows them to be adapted to meet the specific needs of other domains and units of analysis as well. The results confirm this flexibility, indicate they can be adapted to support a full lifecycle analysis, and suggest that they can reduce the effort required to analyze requirements in OSSD projects.

The method I propose for performing an automated open-source requirements discovery and classification provides researchers with the means to explore new areas of OSSD (such as project evolution and success), and practitioners (OSSD project champions, core and peripheral contributors) with the means to better understand and manage/steer the project towards increasing the chance of success. In this thesis, I also present studies in which I apply the

proposed artifacts. While one of these studies represents the direct applicability of the proposed artifacts in the context of OSSD, the other is an exploratory analysis of OSSD project success.

2.2. Pattern-Based Analysis of Requirements

The use of patterns for recognizing requirements is an accepted approach in the field of requirements engineering. A number of researchers proposed in the last part of the 1990s a range of approaches to analyzing natural language text based on patterns (Rumbaugh 1994; Gamma et al. 1995; Cockburn 1997; Toro et al. 1999b). Toro et. al. proposes a series of requirements templates that can help capture requirements. They include linguistic patterns (L-patterns, natural language commonly used for describing requirements) and requirements patterns (R-patterns, generic requirements templates) (Toro et al. 1999a). While attempting to bridge between natural language and formal requirements specifications, Toro's study reaches a middle ground between them. The objective of the techniques proposed is to perform a re-specification, rather than requirements discovery.

Konrad and Cheng propose a set of requirements patterns for embedded systems. Their work does not address requirements discovery but explores requirements patterns identification from existing project requirements. They validate the initial patterns by applying them to two case studies in order to inform future design decisions (Konrad and Cheng 2002). Also for embedded systems, Denger addresses the problem of requirements imprecision (Denger et al. 2003). This study uses a pattern-based analysis of existing requirements in order to identify missing information and to fix the existing inconsistencies. Similar to the other studies, it explores a specific domain (embedded systems) and does not address the problem of requirements discovery. However, the patterns identified are derived from elements of natural language.

2.3. Requirements Discovery

A number of researchers highlight the benefits and appropriateness of using NLP techniques in requirements engineering. On a more general perspective, Kevin Ryan highlighted in the early 1990s the increased need for NLP analyses as a direct reaction to the increased complexity of software systems (Ryan 1993). Other authors suggest more specific uses of NLP. For instance, Sampaio presents a NLP technique based on WMATRIX for analyzing requirements documents with the intent of identifying elements specific to Aspect Oriented Software Development (AOSD). Sampaio's approach can explore both structured as well as unstructured documents. The identification process is supervised and controlled by a researcher and generates a structured document. The identification process is based on frequency analysis and key term extraction (Sampaio et al. 2005).

Ambriola and Gervasi also confirm the use of NLP in the analysis of requirements. Their tool, CIRCE, is designed to support the analysis of structured documents in which requirements are expressed through NL text. CIRCE parses input data and applies a number of modelers in order to present the analyst with rendered (UML-based) views of the model being analyzed. The user can either validate the output models or decide on the specific use of the available data (Ambriola and Gervasi 2006).

Another group of researchers from the Italian academia, Fantechi and Spinicci, propose an artifact for improving requirements quality. Their study describes a semi-automated process for reducing inconsistencies in requirements documents. Their proposed process is another example of the use of NLP techniques in the analysis of requirements. Phrasys, a NLP-based software environment, is used for phrase and sentence extraction. The proposed technique processes structured requirements documents and identifies SAO triples in order

to analyze interactions between entities (Fantechi and Spinicci 2005). Another researcher who uses the SAO pattern is Leonid Kof. He also uses SAO patterns for analyzing structured requirements documents. His goals are to identify associations between terms and to construct domain taxonomies (Kof 2005).

An approach to requirements discovery and classification that is not using only formal requirements documents is the one presented in a study of non-functional requirements (NFR) by Jane Cleland-Huang and her co-authors. This study uses a semi-automated technique for identification and classification of requirements from both structured and unstructured documents. The NFR classification process proposed has three stages: mining phase, classification phase, and application phase. In the mining phase, the authors perform term extraction based on a training set for identification of keywords. The classification phase enhances this process by performing a sentence extraction and a NFR classification from the available documents based on three resources: the terms extracted during the first phase, a document of unclassified software requirements specifications, and unstructured documents listing potential NFRs. Third phase is the one when the user determines the applicability of the results outputted by the software artifact. This semi-automated method requires significant researcher intervention and control throughout the entire process. The proposed artifacts are designed for the specifics of a combination of formal closed-source requirements documents and test data. Moreover, the analysis techniques employed are not NLP-based but statistical (Cleland-Huang et al. 2006)

2.4. Requirements Classification

Requirements have been traditionally classified as either functional (FR) or non-functional (NFR), even though some researchers consider this classification to be too broad (Bass et al.

1998). While adopting this perspective, researchers refer to FR as goals (or hard goals, or behavioral requirements), and to NFR as soft goals (Mylopoulos et al. 1999; van Lamsweerde 2007). Functional requirements are concerned with specifying particular features of the system to be developed. Therefore, a complete set of FR should comprehensively describe the functionality of the new system. Non-functional requirements are concerned with two areas: (1) properties that affect the system as a whole (such as usability, portability, maintainability, or flexibility), and (2) quality attributes (such as accuracy, response time, reliability, robustness, or security) (Chung et al. 1999; Moreira et al. 2002). Some variations to it include the listing of security concerns under FR, adding supportability under NFR, or specifying sub-categories of these two (Grady 1992).

Additional requirements classifications start from an agent-based perspective informing the V-model of requirements. In this approach requirements are listed as user-stakeholder, system, sub-system, or component requirements (Broy and Stauner 1999; Weber and Weisbrod 2003; Hull et al. 2005). From a goal-orientation perspective, goals are identified and analyzed based on the agents that can achieve them. From an agent-oriented perspective, the agents are identified and analyzed based on the goals they need to achieve (van Lamsweerde 2007).

Requirements classifications are often times based on various taxonomies of common types of technologies, or their detailed and specific characteristics. More common requirements classes are reliability, efficiency, integrity, and usability, to name only few. These requirements types and other similar to them are commonly derived from more general product quality models, or from more specific software quality models. Among the quality models that are widely accepted within both the practitioners' and researchers' domains one can mention McCall's (McCall et al. 1977), Boehm's (Boehm et al. 1978),

IEEE (IEEE), and the series of ISO quality models (ISO 2001; ISO 2011). The quality factors included in these models have been often used by researchers for building taxonomies of requirements while the descriptions of these factors have been used to derive classification rules, patterns, and principles.

2.5. Software Product Quality and Software Development Project Success

An important aspect of software development lifecycle is project success. While this concept seems simple and intuitive, there is little agreement as to what constitutes software quality or project success (Pinto and Slevin 1987). Traditionally, projects were perceived as successful when they met time, budget, and performance goals. Obviously "success" is more than meeting budget and deadlines. TAM (Davis 1989; Venkatesh et al. 2003) posits that perceived usefulness and perceived ease of use determine an individual's decision to use a system, which in turn determines that project's success. Taking into consideration that different groups of stakeholders have different views on success, other dimensions have been defined as determinants of a project's success: project efficiency, impact on customer, business success, and preparing for the future (Shenhar et al. 1997). DeLone and McLean identified six dimensions of success: "systems quality" which measures technical success, "information quality" which measures semantic success, and "use," "user satisfaction," "individual impacts," and "organizational impacts" which measure effectiveness success (DeLone and McLean 2003). The quality of information, system, and service leads to higher user satisfaction, which leads to user's intention to use the product, and certain net benefits occur.

In the last few decades, the importance of business related aspects on the success of the development lifecycle has been constantly highlighted. Along with this, we can note an increasing emphasis being placed on not only the developers of products and services and on the

development processes but also on the users and on the efficiency of the communication between users and developers. In the context of software development, the essential role the users play in the success of a system's development and its subsequent use has been recognized constantly since the dominance of modern software development methodologies, such as the series of methods grouped under the umbrella term of agile development. Along these lines, researchers underlined the importance of requirements as a result of user-developer communication and a determinant of project success: "requirements are essential for creating successful software because they let users and developers agree on what features will be delivered in new systems" (Wiegers 2003). Early user involvement has been related to higher requirements quality. Empirical studies confirmed this and showed that involving users and customers as sources of information is related to project success (Kujala et al. 2005). The relationship between requirements quality and project success is also explored by Hooks and Farry (Hooks and Farry 2001). They show that the two variables are positively correlated. Dvir summarizes these findings by stating that user involvement in the development of requirements is positively and significantly correlated with the overall success of a project (Dvir 2003; Dvir 2005). Therefore, requirements and their evolution represent two of the essential attributes of software development projects and require a special attention in studies on project success.

Although the open-source domain attracted in the recent years an increasing number of researchers exploring a wide array of areas, the impact of requirements and their associated processes on OSSD remains an under-explored area of research. The OS areas studied by researchers include aspects of social networking analysis, or exploring related areas such as psychology-based perspectives on the OS participants, the economic impact of OS, laws and policies affecting the OS domain, organizational participation in the OS phenomenon,

applicability and use of OS products in various domains and industries, and NLP-based analysis techniques. A number of studies explored OSSD projects and their associated development processes while adopting a requirements-based perspective (Scacchi 2002; Noll 2008; Scacchi 2009). Current findings are limited to acknowledging the existence of requirements and their associated processes (elicitation, analysis, specification and modeling, validation, and management) in open-source projects (Scacchi 2009). While requirements processes received slightly more attention from a research community apparently focused more on social networking analysis and participant involvement, the end product of OSSD projects, the software product itself and elements that are part of its development lifecycle received a slightly less consistent attention. One important concept that is worth further investigation is represented by open-source requirements and their impact on the quality of OS software products and on the success of OSSD projects.

One way these objectives can be approached is by considering quality to be a main independent variable in a model predicting success and by extending the assumption of quality from the requirement level to the software product level and subsequently to the project level (DeLone and McLean 1992; Crowston et al. 2003). Wiegers' and Dvir's findings that software project success is determined by project's requirements support such a perspective (Wiegers 2003; Dvir 2005). In the exploratory study following the design of an automated requirements discovery and classification artifact, I explore the concept of requirements quality by classifying open source requirements through a mapping to a generally accepted software quality model, such as McCall's (McCall et al. 1977). Therefore, McCall's 23 software quality criteria determine the taxonomy of 23 types used to classify open-source requirements (see Appendix). The objective of this study is to explore characteristics of OSSD projects by studying the impact

of open-source requirements on the dimensions of success from the IS success model proposed by De Lone and McLean. Specifically, I explore the indirect impact open-source requirements exhibit on project success (via characteristics of system use and indicators of user satisfaction). I place requirements-related information that I obtain from using the RCNL tool in the context of other OSSD project related measures and I discover a set of patterns through reasoning and logical inference. For the validation of these findings, I extract and analyze additional OSSD project level information. The findings are validated when they are confirmed by the project characteristics identified in the additional information collected.

3. The Grammar-Based Approach

3.1. Classifier Design

I follow the principles of a design science research (Hevner et al. 2004) to develop the RCNL classifier for the grammar-based strategy. It embodies the theory of a pattern-based approach to discovering requirements from real natural-language. As described above in Section 1.3, the grammar-based strategy proposes the within sentence pattern-based analysis of natural language text. This approach is focused on the use of the SAO (Subject-Action-Object) pattern for requirements discovery. The implementation of this perspective results in proposing a multi-level ontology, in which the lower levels are based on the English grammar specific to informal communication while the upper levels are requirements-based and adapted to the specifics of OSSD. The RCNL ontology is realized in an implementation that uses a multi- level GATE (General Architecture for Text Engineering) parser (Cunningham et al. 2002). I apply the Hevner et. al. (Hevner et al. 2004) descriptive approach to design science – developing and then evaluating the design of RCNL using scenarios and argumentation.

OSSD project data are a central problem to discovery and classification. The data sources are real unstructured natural language text. For OSSD project texts, this means that most of the text does not conform to English grammar. In fact, nearly all the texts are fragments containing many typos, misspellings, technical writing, elements of social communication, and idioms (e.g., text smileys).

To address these characteristics of the data sources, the classifier is a type of weak ontology-based information extraction (OBIE) system. Such systems work well for parsing and tagging fragments of unstructured natural language text (Wimalasuriya and Dou 2010). The ontology is comprised mostly of grammar-based concepts, except for the last level that contains the requirements classification concepts.

I validate the classifier using two methods. First, I apply the classifier to a variety of data sources and measure its classification quality. Such classification scenarios guide the design and development of the classifier. Second, I compare the classifier's results with those of a human expert. I report of these validation efforts in Section 5. Next, I illustrate discovery and classification with an example, and then present the discovery and classification ontology.

Illustrative Text Tagging

Figure 3 illustrates the tagging of a sentence from the data collected from one of the data sources (feature requests for the PasswordSafe project) using the RCNL grammar-based requirements parsing ontology. For illustrative purposes, the text selected is well-structured, follows the general rules of the English language and does not contain typos, misspellings, technical writing, or other elements of informal communication that would distract from the explanatory purpose of this example. The tags from first two levels (level 0 and level 1) are not

shown or shown selectively, because they are mostly common parts of speech (POS) tags and in order to avoid complexity and confusion. For clarity purposes, only the tagging of those pieces of text that would suffer subsequent annotation and participate in pattern matching is highlighted for these first two levels.

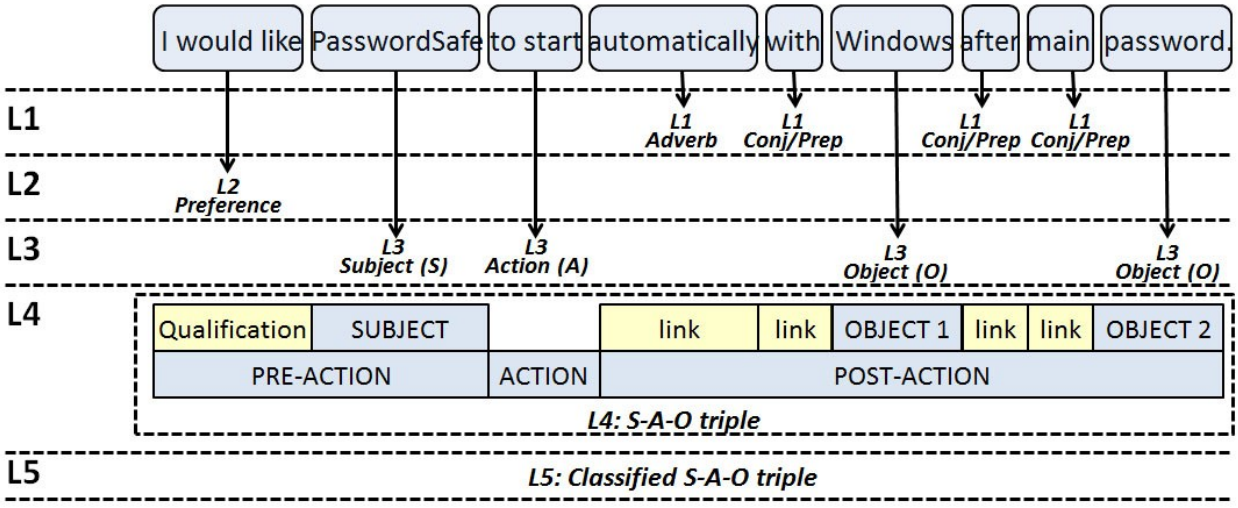

Figure 3. Text tagged with the grammar-based requirements ontology.

Figure 3 shows that the recognized requirement (*L4 – SAO triple*) is comprised of a qualifier (*L2 – Preference*) followed by subject, action, and object tags from L3. The entire statement is tagged as a classified requirement (*L5 – Classified SAO triple*). In informal communication in open-source often only pieces of statements result in being tagged as L5 patterns due to the presence of extraneous text. In the grammar-based analysis, the within sentence recognized requirements are labeled *micro-requirements*.

Requirements Parsing Ontology

The grammar-based RCNL ontology for OSSD projects includes six levels, as summarized in Table 1. The first two levels contain common natural language grammar concepts – tokens and POS. The next three levels contain concepts of logical statements – qualifications, entities comprising micro-requirements (subjects, actions, and objects), and

recognized micro-requirements (SAO triples, SAO extensions, and SAO atomic elements). The final level 5 contains the classification concepts. Although the levels are numbered from 0 through 5, each level may be only partially dependent on the lower levels. For example, level 4 characterizes subject-action-object triples using information from all previous levels while level 3 characterizes SAO elements comprising micro-requirements based on information from levels 0 and 1 only (no information from level 2 is used in level 3).

Table 1. The grammar-based RCNL requirements parsing ontology.

RCNL Level	Level Name	Description	Elements covered
L0	*Token*	Defines basic elements of text commonly included in all types of communication.	Word, punctuation, symbol, list, filename, sentence, email address, url, phrase, syntactical separator
L1	*POS*	Defines most common Parts-of-Speech (POS) elements.	Adjective, adverb, verb, conjunction, preposition, determiner, negation, noun
L2	*Qualification*	Identifies expressions defining a context that can indicate a requirement.	Belief, certainty, necessity, preference, qualifier, quantifier, qualifying phrase
L3	*Entities*	Identifies the three basic elements of a requirement.	Subject(S)/actor, action(A)/verb, object (O)
L4	*Micro-Requirement*	Discovers parts of text identified as micro-requirements.	SAO triples, SAO extensions, SAO atomic elements
L5	*Classification*	Classifies pieces of text identified at previous level and elements of lists.	SAO triples, SAO extensions, list items and introductory phrases

The first two levels are common to all NL parsing systems. Level 0 (L0) concepts include words, punctuation, as well as idioms common to OSSD projects, such as email address, URLs, and file reference. It also includes concepts such as various symbols, list delimiters, sentences, phrases, and various syntactical separators. Level 1 (L1) concepts are

the common English parts of speech (POS), such as adjectives, adverbs, verbs, conjunctions, prepositions, determiners, negations, and nouns.

Level 2 (L2) concepts are qualifiers, such as beliefs, certainties, necessities, preferences, and various qualifying and quantifying expressions. These mostly depend on specific words identified in level 0, but may also depend on the POS of level 1 (e.g., determiners, deictic words, quantifiers, numerals, modals, negating expressions, etc.).

Let's consider the following text from the dataset (from feature request posts of the AWStats project):

```
I think this is great if awstats html tag can calculate ROI ...
```

The keyword "`think`" indicates the presence of an expression of belief. Any beliefs, preferences, expressions of certainty or necessity, or quantifiers or qualifiers that potentially modify a micro-requirement are tagged at L2. Such expressions usually introduce a micro-requirement.

Level 3 (L3) concepts are simply subject, action, and object. Subject is not simply a noun, but an actor (person, object, entity, or concept). The actor may execute some action on an object. The action is expressed through a verb or set of verbs defining the desired course of events. The object of this action can be any entity or set of entities in the environment impacted by the performing of the action.

The analysis of requirements through the lenses of a structured approach built on the subject-action-object (SAO) triple is not new. Fantechi and Spinicci use this approach for analyzing interactions among entities in a semi-automated process of reducing requirements inconsistencies in structured requirements documents (Fantechi and Spinicci 2005). Leonid Kof is also using the SAO pattern, but in his case the objective is to develop domain taxonomies out of formal requirements documents (Kof 2005; Kof 2007). In the

grammar-based RCNL ontology, L3 patterns discover all subjects, actions, and objects present in text that can potentially be part of a micro-requirement.

A subject-action-object assertion is the concept of Level 4 (L4). Adjectives, adverbs, and other elements may be involved, but level 4 represents the central micro-requirements statement. Often, a level 4 micro-requirement is qualified by a level 2 expression as shown before.

In informal communication it is not uncommon for the actor of a statement or for the objects impacted by the action of the statement to be inferred rather than specifically mentioned. This is achieved through the use of deictic words and represents some of the exceptions that are considered in L4. Since the existence of a subject and object in text is optional, let's try to exemplify this situation with an example. The following text (from the dataset) is tagged as two L4 requirements (separate by the "`, but`").

```
Keep the current view of top keywords, but add a new option to display
the following information.
```

The L4 patterns can reference the other levels. In fact, the qualifier tags of L2 often introduce the L4 micro-requirement. This example (from the dataset) illustrates this situation:

```
I want to see when I get more visitors, and be able to compare my
traffic to other days.
```

The expression "`I want`" (L2) qualifies the micro-requirement (L4) that it introduces.

Finally, level 5 (L5) concepts are the domain specific classification of the level 4 statements. I have designed two L5 classifiers:

1. Standard classifier based exclusively on McCall's quality model (I call this *McCall classification* or *McCall*)

2. Extended classifier based on McCall's quality model and enhanced with libraries of patterns encoding OSSD-specific attributes (I call this *McCall+ classification* or *McCall+*)

McCall's model specifies 23 quality criteria for software. These concepts are represented in L5. In particular, I specify rules for recognizing the 23 quality criteria in the pieces of text that are annotated in L4 as micro-requirements. I aimed to accurately capture the quality model as specified by McCall (McCall et al. 1977).

In addition to the classification rules directly derived from McCall's quality model, I specified my own classification rules for the 23 quality criteria. In particular, these extend the McCall's classification rules to recognize concepts and terms unmentioned in the McCall specification. I call these the OSSD extensions of McCall's model, or *McCall+*.

The McCall OSSD extensions are based on two sources. First, the NLP literature for requirements parsing suggests keywords and parsing strategies – in particular, Cleland-Huang's study on non-functional requirements (Cleland-Huang et al. 2006). Second, analysis of NL associated with OSSD projects suggests further keywords and parsing strategies. In particular, I iteratively extend the RCNL parser, and test it on sample data, until I reach a consistent level of correct classification. Additionally, I make use of the SensAgent online dictionary for gathering all synonyms who are properly describing the same meaning as the original classification keyword or expression (www.sensagent.com). The Appendix section includes illustrative classifications. Here, I report on only the McCall+ classifications, because of their significantly better classification efficiency over the McCall classifications.

3.2. Classifier Engineering

The RCNL classifier is implemented in GATE (Cunningham et al. 2002). The General Architecture for Text Engineering is developed by the Sheffield Natural Language Processing Group at the University of Sheffield and is surrounded by a large community of collaborators and users. Next, I describe at a high level the engineering involved in realizing the RCNL framework in GATE. In particular, I describe rules for tagging text according to the ontology, additional text processing, and the overall text processing activity.

The parser implements the RCNL ontology to recognize and classify NL micro-requirements. The patterns used in the tagging of NL text are encoded in RCNL using the JAPE (Java Annotation Pattern Engine). For each level, JAPE rules specify how GATE tags text with concepts of that level. The rules are organized in a pipeline and executed sequentially, from level 0 to level 5. The final output includes qualified (L2) micro-requirements (L4) that are also classified (L5) according to the rules of McCall+ classifier. Any piece of text may have multiple tags generated by rules from multiple levels.

GATE supports levels 0 and 1 directly, identifying tokens and some parts of speech. The RCNL classifier rules augment and extend the native GATE tags to aid processing for OSSD projects.

Rule-Based Tagging

GATE defines an architecture for executing plugins over NL text. GATE users may develop their own plugins; however, GATE provides a variety of plugins for common NLP tasks.

GATE also provides JAPE (Java Annotation Pattern Engine), a rule-based text-engineering engine that supports Java and regular expressions. Another benefit of using GATE is the annotation indexing and search engine with an advanced graphical user interface called ANNIC (Annotations in Context). The analyses of this study use ANNIC

for development of rules and inspection of results, and JAPE for rule design and implementation.

JAPE rules specify a left-hand side (LHS) in which the pattern to be matched is defined and a right-hand side (RHS) in which the annotation and its features to be created for all the discovered instances of the pattern are being specified. Multiple and complex patterns can be defined in the LHS of a JAPE rule. Similarly, the RHS of a JAPE rule can be used to specify multiple annotations and features to be created for each matching pattern or for each matching element of a pattern.

The current implementation of the grammar-based RCNL classifier consists of over 200 JAPE rules, not including the rules designed for generating evaluation metrics. To illustrate how the grammar-based RCNL ontology is recognized through JAPE rules, I present rules from levels 3 and 5. The rules presented here are simplified for clarity.

A Level 3 Rule

To illustrate the rule techniques, here is a rule from L3.

```
Rule: PotentialSubjectFinder
(
   (
      {Token.category == PP} |
      {Token.category == PRP} |
      {Token.category == "PRPR$"} |
      {Token.category == "PRP$"} |
      {L1.category == "Noun"} |
      {L0.category == "Filename"} |
      {L0.category == "email"} |
      {L0.category == "url"} |
     // ...
      ({L1.category == "Determiner"} {L1.category == "Noun"})
   ) [1,5]
)
:SubjectFound
-->
 :SubjectFound.L3 = {category = "Subject"}
```

The LHS part of the rule defines a pattern searching for pronouns (as defined in pre-defined rules in GATE), or nouns (as defined in L1), or filenames, Url's, email, (as defined in L0), or a determiner followed by a noun (up to 5 instances of this pair). When either one of these is found, the text matching the pattern is annotated as an L3 `Subject`.

A Level 5 Rule

Here is an L5 classification rule.

```
Rule: L5_Comunicativeness
(
   {L4.valid == "Yes",L4_Requirement contains KW_F5C12}
)
:L5_ComunicativenessFired
-->
 :L5_ComunicativenessFired.Comunicativeness = {category = "F5C12"}
```

The LHS part of the rule matches text annotated as L4 (micro-requirement) that contains keywords associated with factor 5 and criteria 12 of McCall's model. The matched text is annotated as `Communicativeness`, which is the label for factor 5, criteria 12.

Auxiliary Text Processing

Three auxiliary kinds of text processing are noteworthy. First, list processing presents an interesting problem. OSSD project texts include technical yet informal communication containing numerous examples of specifications expressed with lists. Lists typically have an introductory phrase followed by one or more list items:

```
<Introductory phrase> [<list item>]+
```

Sometimes the introductory phrase and each list item are complete micro-requirements. However, most often the introductory phrase can be classified as a micro-requirement while the list items are examples or statements that extend the meaning of the introductory phrase. To address such issues, the L5 tag associated with the introductory phrase is propagated to all the list items. As such, a list item can have two tags: a tag from parsing the list item, and a tag

propagated from the introductory phrase. List classification processing occurs in L5 at the same time with the regular classification of micro-requirements.

A second auxiliary text processing involves the checking for micro-requirement containership. It is possible, but rare, that a micro-requirement is fully contained inside another micro-requirement. When found, a final finishing rule re-annotates micro-requirements to indicate that only the larger micro-requirement should be considered for classification. This avoids double annotating and double classifying same piece of text.

The last auxiliary text processing generates metrics for evaluating the analysis and the rules. These include:

- Tokens covered: Total number of tokens in text covered by the text annotated as micro-requirements.
- Sentences covered: Total number of sentences in text including text annotated as micro-requirements.
- Micro-requirements tagged: Total number of micro-requirements discovered
- Classifications created: Total number of classifications created
- Requirements classified: Total number of discovered micro-requirements that are classified

Figure 4 shows the RCNL classifier as implemented in GATE. The screen image shows text tagged as micro-requirement, and two classifications, `Operability` (factor 5, criteria 10) and `StorageEfficiency` (factor 3, criteria 7). The image shows how the OSS documents are referenced as GATE's Language Resources, at the left. Below that are the Processing Resources, which provide the syntactic and rule-based processing components for the pipeline processing of the documents.

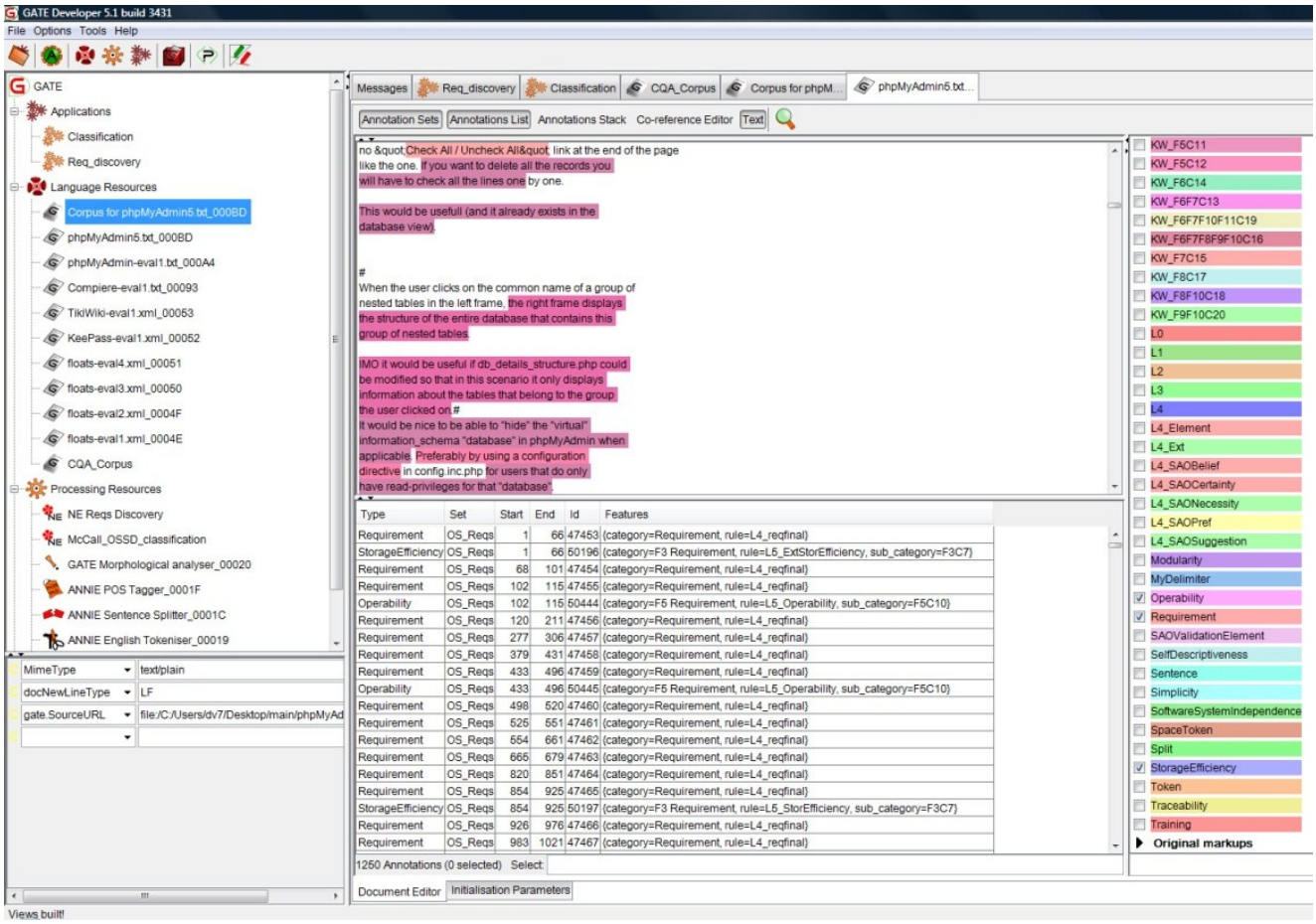

Figure 4. The RCNL classifier as implemented in GATE.

4. The Delimiter-Based Approach

As in the case of the grammar-based strategy, the development of the RCNL classifier for the delimiter-based strategy also follows the principles of a design science research (Hevner et al. 2004). A similar pattern-based approach to discovering requirements from real natural-language is used in this strategy. As described above in Section 1.3, the delimiter-based strategy proposes the within posting pattern-based analysis of natural language text. This second approach is based on the discovery of semantic delimiters for requirements discovery. The implementation of this perspective results in proposing a multi-level ontology similar to the one proposed for the

grammar-based strategy. The RCNL ontology is also realized in an implementation that uses a multi-level GATE parser (Cunningham et al. 2002).

In addition to the validation methods mentioned in previous section, which I also use for the delimiter-based parser, in this research I also consider an evaluation in which I compare the accuracy and efficiency of the two strategies. However, the two strategies are not to be considered mutually exclusive but complementary, as detailed in Section 1.3. Next, I illustrate discovery and classification with an example, and then present the discovery and classification ontology.

4.1. Classifier Design

Illustrative Text Tagging

Figure 5 illustrates the tagging using the RCNL requirements parsing ontology of a sample posting from the Feature Request forum of KeePass Password Keeper project. For illustrative purposes, the text is well-structured and simple and few segments of text are omitted. For instance, parts describing the desired functionality are omitted as well as the signature of the posting author. The tags from first three levels (L0, L1, and L2) are not shown, because they are mostly common tags used in the annotation of basic elements of text, or do not play an important role in this example.

Figure 5 shows how a concept separator (L3, "`also`") is used to separate a feature request posting into two topics. Due to this separation, the posting is split up into two macro-requirements (L4). Note that first macro-requirement includes a sentence delimiter (the punctuation symbol at the end of the sentence) as well as first part of the subsequent sentence. This is expected behavior for the delimiter-based strategy. Both annotated macro-

requirements are tagged as a classified requirement (L5) is a classification pattern is recognized within their boundaries.

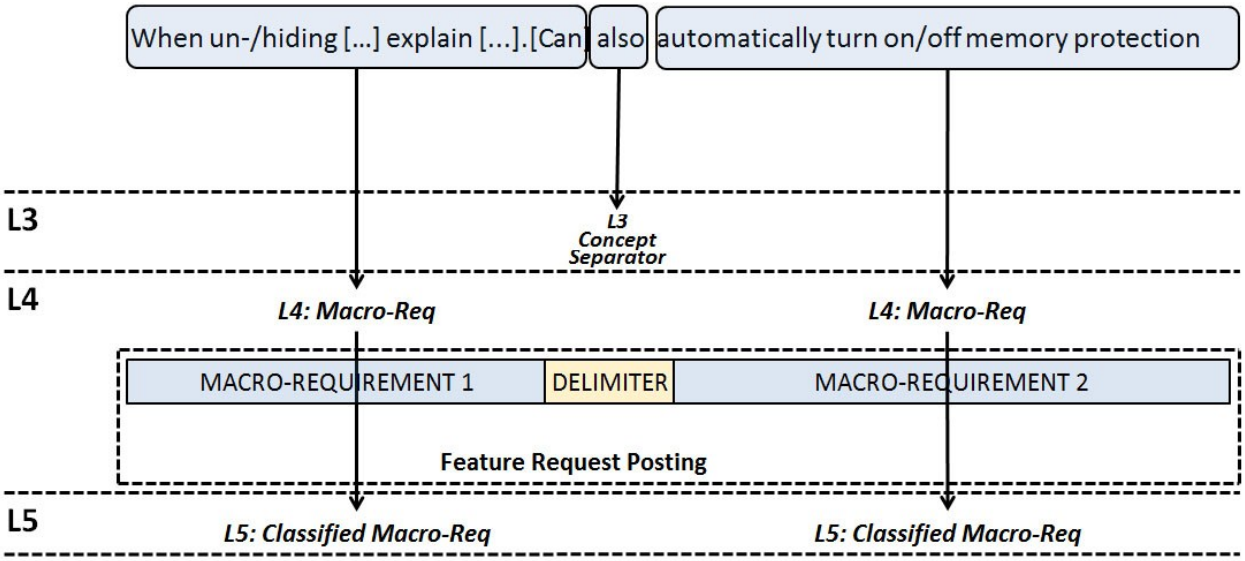

Figure 5. Text tagged with the delimiter-based requirements ontology.

Requirements Parsing Ontology

The RCNL ontology for the delimiter-based approach is designed on the same principles as the RCNL ontology for the grammar-based approach. Table 2 summarizes the six levels of the RCNL ontology for the delimiter-based approach. Although the levels are numbered from 0 through 5, each level may be only partially dependent on the lower levels. For example, Level 3 characterizes concept separators using information from Levels 0, 1, and 2. Level 4 identifies parts of a posting that describe a desired new feature using information from level 3 only. The first three levels and the classification level (L5) include same grammar and requirements-oriented concepts as in the first strategy and offer same results and benefits.

Table 2. The delimiter-based RCNL requirements parsing ontology.

RCNL Level	Level Name	Description	Elements covered
L0	*Token*	Defines basic elements of text commonly included in all types of communication.	Word, punctuation, symbol, list, filename, sentence, email address, url, phrase, syntactical separator
L1	*POS*	Defines most common Parts-of-Speech (POS) elements.	Adjective, adverb, verb, conjunction, preposition, determiner, negation, noun
L2	*Qualification*	Identifies expressions defining a context that can indicate a requirement.	Belief, certainty, necessity, preference, qualifier, quantifier, qualifying phrase
L3	*Concept Separators*	Identifies pieces of text separating logical concepts.	Semantic separators
L4	*Macro-Requirement*	Discovers parts of text identified as macro-requirements.	Sets of statements delimited by concept separators (L3)
L5	*Classification*	Classifies pieces of text identified at previous level and elements of lists.	Macro-requirements (L4)

The delimiter-based parsing strategy relies on discovering concept separators in Level 3. They indicate that the discussion on a desired piece of functionality concludes and the discussion on a new desired feature starts. Level 3 patterns are comprised of words and expressions. Punctuation symbols are not considered to be semantic separators but only grammatical separators.

Level 4 provides a basic parsing of feature requests postings for tagging the amount of text between the beginning of the posting and first concept separator, or between two consecutive concept separators, or between last concept separator and the end of the posting as macro-requirement.

The delimiter-based recognizer relies on tokens and parts of speech to recognize macro-requirement delimiters at the feature request posting level. Consequently, a delimiter-based requirement can include multiple micro-requirements. If the requirement in Figure 3 were all the

text in a posting, then it would also be recognized as a macro-requirement. However, if it were included with other statements, such as in the case presented in Figure 5, with intervening delimiters, then each set of delimited statements would be recognized as a macro-requirement. The delimiter-based recognizer mostly considers each posting to the SourceForge Feature Tracker as a macro-requirement. There are a few exceptions, in which keywords (e.g., first, second, third ..., conversely, addition to) separate a post into more than one requirement. As the results of applying the RCNL classifier to open-source projects show in the next sections, there is an average of less than 2 macro-requirements per posting.

4.2. Classifier Engineering

The implementation of the RCNL classifier under the delimiter-based strategy follows uses the same resources and technologies as the implementation for the grammar-based strategy. The implementation environment is GATE, and the rule encoding engine is JAPE.

Parsing Pipeline

Figure 6 illustrates how the RCNL rules are executed within GATE and the distinction between the implementations of the two strategies. The process begins with a NL resource, which is processed through a series of special purpose programs, or plugins. Pre-processing includes five stages followed by Named Entity (NE) Transducers, as shown in Figure 6. NE transducers are plugins that call on specific parsing pipelines comprised of JAPE rules designed by the researcher. The five pre-processing plugins perform the following tasks:

1. Language resources are converted into GATE format.
2. An English Tokenizer plugin identifies tokens within the text provided. These tokens are basic elements of text, such as words, punctuation, spaces, etc.

3. A Sentence Splitter plugin identifies and annotates pieces of text corresponding to sentence and paragraph structures.
4. A POS Tagger plugin identifies Parts-Of-Speech in the text. POS are elements such as adjectives, adverbs, nouns, verbs, conjunctions, etc.
5. A Morphological Analyzer plugin identifies each token's lemma and affix. I use this plugin in the analysis because it provides stemming capabilities to the NLP analysis.

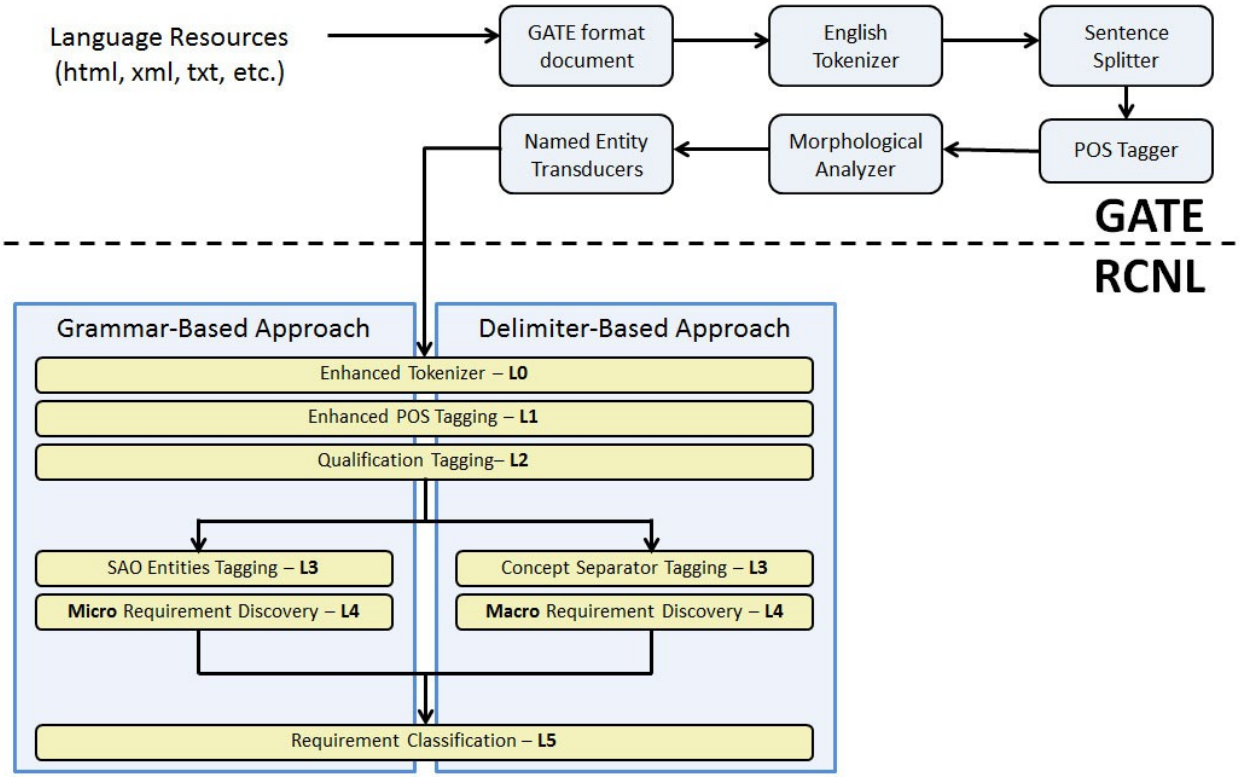

Figure 6. Text tagged with the delimiter-based requirements ontology.

The analysis following pre-processing is completed with using Named Entity Transducers and runs the RCNL JAPE rules on the pre-processed data. Discovery and classification in the grammar-based analysis is comprised of around 200 JAPE rules, while in the delimiter-based analysis it is comprised of around 120 JAPE rules.

I design, develop, and organize the sets of JAPE rules for discovery, classification, and evaluation separately from each other. While discovery is distinct between the two strategies, classification rules are the same in both approaches. Finally, a third set of rules post-processes the analysis results preparing measures of interest for the evaluation process.

Rule-Based Tagging

The current implementation of the RCNL classifier for the delimiter-based strategy includes JAPE rules of significantly higher complexity than in the case of the implementation for the grammar-based approach. To illustrate how the RCNL ontology is recognized through JAPE rules in the delimiter-based strategy, I present two simplified rules from Levels 3 (concept separator) and 4 (macro-requirement discovery).

A Level 3 Rule

To illustrate the rule techniques for the delimiter-based approach, here is a rule from L3.

```
Rule: macro_separator_prekeyword2_ext
Priority: 20
(
   ({SK2_ext within Sentence}) :seppreK2ext
   (({!Split, Token within Sentence}) [0,1000]) :restofsentence
   {Split}
)
:macroseppreKW2extFired
-->
 :seppreK2ext.Separator_Prekeyword2_ext = {category = "Macro
Requirement Separator Pre-Keyword Extend"}
```

The LHS part of the rule describes a pattern searching for a separator keyword of type 2 (SK2) within a sentence, followed by a number of tokens different from a custom defined set of delimiters, followed by one of the custom defined separators. In a simpler explanation, this rule is searching for a specific type of concept separator placed anywhere but at the end of a sentence. When the entire pattern matches, only the keyword of type 2 will be tagged for further processing, leaving the rest of the sentence not tagged.

A Level 4 Rule

Here is a (slightly simplified) L4 discovery rule.

```
Rule: req_body
Priority: 20
(
   {!Micro_req_separator, NLI within Posting}
   |
   (({Micro_req_separator}) [0,1])
   (({!Micro_req_separator, Token.string != "#", Token within Posting})
[1,10000])
)
:reqbodyFired
-->
 :reqbodyFired.Macro_Requirement = {category = "Posting-
Based_Requirement", rule = "req_body"
}
```

The LHS of the rule matches a pattern comprised of either a Numbered List Item (NLI) within a sentence as defined in level 0 (L0), or a sequence of up to 10000 tokens not including separators or posting delimiters. It is important to note here that all separators and annotations used in this rule are defined in rules corresponding to previous levels of the RCNL ontology.

5. Evaluation and Applications

Having created a requirements classifier for the unstructured and typo-rich NL text of OSSD projects, I applied a number of validation methods to assess it. Additionally, I implemented two strategies for OSSD requirements discovery classification and assessed their characteristics.

- Which strategy classifies the most text?
- Which strategy is most consistent with the way a human analyst would annotate requirements within the text?
- Is there any value in running both strategies, i.e., the hybrid approach?

To answer these questions, I first consider a glass-box evaluation method. In particular, I run the RCNL classifier on a variety of pilot data segments and inspect output's quality in order to improve the design of the classifier. These iterations continue until I reach saturation, meaning

that I reach marginal improvement over previous feedback iterations. Second, I consider a back-box evaluation method. Specifically, I use the RCNL classifier on a large dataset of OSSD project data and measure tool's computational performance. Third, I consider an intrinsic evaluation of the RCNL tool. For this method, I compare the automated classification generated by the classifier to that of a human expert. As a result of this comparison I generate a set of evaluation measures that are well-established in the field of information retrieval (IR) and NLP and then interpret the measures. Fourth, I use the developed artifacts for conducting a requirements-based exploratory study of OSSD projects. In this study I explore OSSD project level characteristics such as project type and success, and propose a wave theory of innovation in OSSD. Next sections present the experiments associated with the evaluation and exploratory efforts mentioned here.

5.1. The SourceForge Dataset

Like many researchers in OSSD, I selected SourceForge projects for data collection (Lintula et al. 2006). SourceForge provides access to over 324,000 OSSD projects and over 3.4 million registered user's activities, as of June 2012. I decided to take advantage of the enhanced online access offered to the SourceForge dataset by the Department of Computer Science & Engineering at Notre Dame University through the SourceForge Research Data Archive (SRDA) (Madey ; Gao et al. 2007). In particular, I processed the May 2011 data from SourceForge.

I narrowed the dataset to substantial projects that actively use requirements. I define this as:

1. Having more than 4 developers
2. Having more than 5,000 downloads
3. Having more than 600 feature requests posts

After a review of the project data, Biet-O-Matic was removed because it is entirely German NL text and the classifier's rules are designed for processing exclusively English NL data. The result is a dataset comprised of 30 projects (see Table 3) with an average size of 4,962 sentences or 110,517 tokens. While the project with the longest history and the most active community is SourceForge.net (over 5,200 feature requests posts), the total number of downloads indicates phpMyAdmin as the most popular (almost 6 million downloads). In terms of project contributors, Tiki Wiki CMS Groupware is the project with the largest set of contributing members (over 130). The Matplotlib project had many spam posts, perhaps the result of a virus on a contributor's computer. In order to avoid a significant skewing effect on the results of the analysis, I manually removed all spam postings and analyzed the remaining data of the project.

Table 3. The dataset of OSSD projects.

Project Name	Contributors (>=4)	Downloads (>=5000)	Feature Requests Posts (>=600)	Tokens	Sentences
SourceForge.net	17	72,595	5212	534,484	24,554
Gallery	11	2,124,290	2194	210,574	9,229
KeePass Password Safe	8	129,001	1486	141,052	6,733
FileZilla	8	3,770,280	1335	126,220	5,789
phpMyAdmin	9	5,908,777	1317	153,744	6,580
PhpGedView	9	39,023	1218	135,540	6,041
WinMerge	5	586,110	1189	116,259	5,415
POPFile - Automatic Email Classification	5	452,133	1061	120,836	5,433
Arianne RPG	13	198,415	1033	98,760	4,429
MegaMek	18	181,061	1025	120,013	5,166
Tiki Wiki CMS Groupware	138	308,746	879	114,262	5,408
AWStats	5	585,317	861	105,795	4,217
Scintilla	10	402,190	805	92,019	4,281
Matplotlib (spammed)	5	17,022	752	11,947	490
floAt's Mobile Agent	14	1,274,217	725	58,240	2,665
Compiere ERP + CRM Business Solution	30	805,876	724	64,421	2,937
TortoiseCVS	16	661,084	724	75,510	3,396
JabRef	17	35,794	716	74,978	3,490
Gutenprint - Top Quality Printer Drivers	28	896,429	704	49,890	2,475
Fire	11	710,644	701	54,142	2,413
SW Test Automation Framework	7	56,216	686	98,950	3,831
EGroupware Enterprise Collaboration	23	421,201	671	63,709	2,868
more.groupware	13	225,301	666	64,523	3,375
MediaWiki	25	96,189	638	76,111	3,208
Windows Installer XML (WiX) toolset	8	120,634	637	77,272	3,150
ScummVM	25	1,140,520	630	80,467	2,907
Slash	8	108,533	618	95,123	3,889
Password Safe	8	207,386	616	56,681	2,598
Tcl	34	1,378,095	611	193,868	9,764
OSCARMcMaster	13	6,447	603	50,112	2,123

Min	5	6,447	603	11,947	490
Max	138	5,908,777	5,212	534,484	24,554
Average	18.0	763,984	1,035	110,517	4,962

5.2. Experiment Configurations

The analysis considered uses experimental configurations to process and evaluate each project from the dataset:

1. Requirements recognition based on (a) the grammar-based parser, and (b) the delimiter-based parser
2. Requirements classification based on (a) only McCall's quality model and (b) the extensions to McCall's quality model, called McCall+

The computational performance analysis of the RCNL classifier considers the time to recognize and classify requirements using the two strategies:

- Grammar-based strategy averages 463.5 tokens processed per second, which amounts to about 3.97 minutes for an average size project in the dataset (project size determined as the total number of tokens). The JAPE rules that are part of the grammar-based implementation of the RCNL classifier are many, but relatively simple. Consequently, they process quickly due to limited memory and computational demands.
- Delimiter-based strategy averages 211.0 tokens processed per second, which amounts to about 8.73 minutes for an average size project in the dataset (project size determined as the total number of tokens). The JAPE rules that are part of the delimiter-based implementation of the RCNL classifier are few and relatively complex. Consequently, the processing time is longer due to the greater memory and computational demands.

Note that neither grammar is a simple context-free grammar as found in most programming languages. The parser applies knowledge of English language terms and grammar. Naturally,

classification takes the most time. The results are obtained on a 3.2 GHz computer running Windows 7. Database retrieval of the features, storage of the results, and evaluation processing are not considered in these numbers. The performance analysis shows that both strategies offer reasonable computational times to project leaders interested in receiving support from the automated requirements discovery and classification tool.

All JAPE rules implemented in the RCNL classifier are organized in a manner showing concern for modularity and separation of functional characteristics. This, along with having all JAPE rules controlled by a flexible custom configuration, provides the researcher with the ability to turn JAPE rules on and off for conducting custom analyses. The experiments, described next, enable or disable various rules to determine their contribution to the classifier's performance.

5.3. Data Analysis and Results

The results of the analysis on the dataset of 30 OSSD projects show that the grammar-based strategy discovers an average of 7,304 micro-requirements per project compared to the 1,607 macro-requirements per project discovered using the delimiter-based strategy. As Table 4 shows, the grammar-based strategy identifies more requirements per project because it discovers all SAO triples within text. In contrast, the delimiter-based strategy identifies, on average, a little more than one requirement delimiter for every other Feature Tracker post. On average, the delimiter-based strategy identifies 1.62 macro-requirements per Feature Tracker post (see Table 5). The delimiter-based strategy includes all the text of each post, thus text coverage for the delimiter-based strategy is, by definition, 100 percent. In contrast, the grammar-based strategy excludes text that does not conform to the SAO patterns used by the parser; thus, its sentence coverage averages only 84.3 percent. This is an expected result if we consider the density of elements of social communication that are present in the informal open-source communication.

Table 4. Requirements recognition using the two strategies.

#	Project	SAO-based		Delimiter-based
		Sentences covered	*Micro-Requirements discovered*	*Macro-Requirements discovered*
1	SourceForge.net	83.4%	35,107	7,368
2	Gallery	85.7%	13,910	3,139
3	KeePass Password Safe	83.9%	9,712	2,295
4	FileZilla	85.3%	8,551	1,853
5	phpMyAdmin	83.5%	9,689	2,124
6	PhpGedView	87.4%	8,982	2,074
7	WinMerge	87.7%	8,116	1,932
8	POPFile - Automatic Email Classification	87.2%	8,258	1,681
9	Arianne RPG	90.0%	7,331	1,557
10	MegaMek	86.9%	8,008	1,668
11	Tiki Wiki CMS Groupware	83.8%	7,824	1,703
12	AWStats	80.8%	6,166	1,493
13	Scintilla	84.3%	6,304	1,444
14	matplotlib	81.6%	685	184
15	floAt's Mobile Agent	85.2%	3,950	1,007
16	Compiere ERP + CRM Business Solution	86.1%	4,505	1,108
17	TortoiseCVS	86.9%	5,390	1,196
18	JabRef	83.2%	4,954	1,171
19	Gutenprint - Top Quality Printer Drivers	80.8%	3,192	903
20	Fire	87.3%	3,784	930
21	SW Test Automation Framework	86.6%	6,056	1,306
22	EGroupware Enterprise Collaboration	86.2%	4,440	1,065
23	more.groupware	83.0%	4,495	1,089
24	MediaWiki	87.4%	5,033	1,078
25	Windows Installer XML (WiX) toolset	85.7%	4,777	980
26	ScummVM	86.6%	4,448	944
27	Slash	88.2%	6,307	1,241
28	Password Safe	88.0%	3,911	976
29	Tcl	65.9%	11,794	1,810
30	OSCARMcMaster	70.4%	3,432	883
	Average	**84.3%**	**7,304**	**1,607**

Table 5. Additional results of the automated discovery and classification process.

#	Project	Micro / Macro	Macro / Posting	Micro / Posting	*Micro Classified / Posting*	*Macro Classified / Posting*
1	SourceForge.net	4.76	1.44	6.85	3.80	1.30
2	Gallery	4.43	1.42	6.31	3.69	1.32
3	KeePass Password Safe	4.23	1.54	6.50	4.23	1.41
4	FileZilla	4.61	1.43	6.58	4.05	1.34
5	phpMyAdmin	4.56	1.49	6.78	4.28	1.39
6	PhpGedView	4.33	1.67	7.23	4.21	1.50
7	WinMerge	4.20	1.62	6.79	4.03	1.45
8	POPFile - Automatic Email Classification	4.91	1.57	7.72	4.32	1.42
9	Arianne RPG	4.71	1.47	6.94	3.47	1.32
10	MegaMek	4.80	1.61	7.73	4.37	1.50
11	Tiki Wiki CMS Groupware	4.59	1.55	7.14	4.03	1.40
12	AWStats	4.13	1.71	7.08	3.69	1.41
13	Scintilla	4.37	1.77	7.71	4.45	1.56
14	Matplotlib	3.72	1.75	6.52	3.69	1.43
15	floAt's Mobile Agent	3.92	1.37	5.36	3.12	1.27
16	Compiere ERP + CRM Business Solution	4.07	1.51	6.12	3.88	1.41
17	TortoiseCVS	4.51	1.61	7.24	4.25	1.36
18	JabRef	4.23	1.62	6.86	4.17	1.43
19	Gutenprint - Top Quality Printer Drivers	3.53	1.29	4.56	2.82	1.12
20	Fire	4.07	1.30	5.29	2.99	1.22
21	SW Test Automation Framework	4.64	1.90	8.83	5.30	1.75
22	EGroupware Enterprise Collaboration	4.17	1.55	6.44	3.87	1.43
23	more.groupware	4.13	1.44	5.95	3.47	1.30
24	MediaWiki	4.67	1.67	7.82	4.17	1.43
25	Windows Installer XML (WiX) toolset	4.87	1.52	7.42	4.47	1.39
26	ScummVM	4.71	1.53	7.23	4.13	1.41
27	Slash	5.08	1.62	8.22	4.52	1.47
28	Password Safe	4.01	1.55	6.22	4.12	1.42
29	Tcl	6.52	2.97	19.33	7.70	2.32
30	OSCARMcMaster	3.89	2.24	8.69	4.28	1.85
	Average	**4.45**	**1.62**	**7.31**	**4.12**	**1.44**

Consider the Compiere ERP project as an example. The analyzed text has 64,421 tokens. The tokens comprise 2,937 sentences, 86.1% of which are recognized by RCNL as including one or more requirements statements (micro-requirements), according to the grammar-based parser. The remaining text is unrecognized by the grammar-based parser, often because it is code segments, social tags, etc. In contrast, the delimiter-based parser considers all of the text within a post to belong to one or more macro-requirements.

Once a segment of text is recognized as a requirement (micro or macro), it is passed to the classifier, which attempts to classify it according to a specified ontology. The same classifier applies to both grammar-based (micro) requirements and delimiter-based (macro) requirements. Some requirements remain unclassified by either strategy. This occurs when the classifier cannot match the given requirements text with a classification rule. The difference between strategies is that the grammar-based requirements are shorter segments with length varying between SAO-length and sentence-length, while the delimiter-based (macro) requirements are mostly post-length requirements, usually including few sentences. The length of requirements has a direct impact on classification efficiency. Table 6 presents the classification efficiency of McCall+ for the two strategies implemented as well as an efficiency comparison between McCall and McCall+ for the grammar-based strategy. McCall+ (the extended McCall classification) provides classifications per requirement at an average rate of 2.4 and 4.2 for grammar-based and delimited-based requirements, respectively. This is expected because the longer-length delimiter-based requirements have more words, and thus a higher likelihood of matching more than one classification. Conversely, the grammar-based requirements have fewer words and thus, on average, they match fewer classifications. While comparing the classification efficiency of

the two classifiers (McCall and McCall+), one can clearly note that the extensions to the standard McCall model did improve classification – 57.53% is much better than 26.42%.

Table 6. Average number of classifications per discovered requirement.

#	Project	Grammar-based	Delimiter-based	Classification coverage (grammar-based)	
		McCall+	*McCall+*	*McCall*	*McCall+*
1	SourceForge.net	2.2	4.2	19.21%	50.08%
2	Gallery	2.3	4.3	22.96%	52.12%
3	KeePass Password Safe	2.7	4.9	31.92%	63.46%
4	FileZilla	2.4	4.3	29.77%	60.11%
5	phpMyAdmin	2.4	4.6	25.07%	61.61%
6	PhpGedView	2.3	4.1	23.36%	56.45%
7	WinMerge	2.5	4.2	22.05%	58.25%
8	POPFile - Automatic Email Classification	2.2	4.3	27.04%	58.54%
9	Arianne RPG	2.2	3.6	28.82%	61.84%
10	MegaMek	2.5	4.6	29.75%	60.74%
11	Tiki Wiki CMS Groupware	2.4	4.2	34.83%	65.00%
12	AWStats	2.2	3.5	25.84%	56.50%
13	Scintilla	2.5	4.3	20.45%	53.35%
14	Matplotlib	2.2	3.7	28.22%	56.54%
15	floAt's Mobile Agent	2.3	3.9	28.28%	58.38%
16	Compiere ERP + CRM Business Solution	2.6	4.4	25.64%	49.27%
17	TortoiseCVS	2.5	4.3	37.10%	66.30%
18	JabRef	2.5	4.5	26.12%	58.21%
19	Gutenprint - Top Quality Printer Drivers	2.2	3.6	31.86%	63.24%
20	Fire	2.3	4.0	22.31%	55.92%
21	SW Test Automation Framework	2.5	4.3	25.78%	57.80%
22	EGroupware Enterprise Collaboration	2.3	4.1	23.47%	57.17%
23	more.groupware	2.4	4.1	20.12%	54.95%
24	MediaWiki	2.2	4.0	25.01%	55.40%
25	Windows Installer XML (WiX) toolset	2.6	5.0	28.19%	60.01%
26	ScummVM	2.4	4.3	17.67%	39.80%
27	Slash	2.2	4.2	26.11%	56.45%
28	Password Safe	2.7	4.9	26.98%	58.72%
29	Tcl	2.4	3.7	34.04%	60.25%
30	OSCARMcMaster	3.3	3.4	24.48%	59.33%
	Average	**2.4**	**4.2**	**26.42%**	**57.53%**

The analysis includes a number of metrics in addition to the ones presented and which are associated with the delimiter-based requirements (macro-requirements) and with the grammar-based requirements (micro-requirements). These metrics are presented in Table 5 and Table 6 and we can draw interesting inferences from them.[1]

1. A feature request posting for the selected projects averages 1.62 macro requirements. (There is an average of more than 0.5 delimiters that split a post into more than one macro-requirement.)
2. On average, there are 4.45 times more micro-requirements than macro-requirements. (An aggregate macro-requirement contains 4.45 sub-requirements.)
3. On average, there are 4.2 classifications per macro-requirement and 1.4 per micro-requirement. This means that Feature Tracker posts refer to more than one of McCall's criteria, but each sub-requirement addresses (slightly more than) one criteria.
4. On average, there are 1.73 more classifications per requirement using the delimiter-based analysis than the grammar-based analysis. (The greater word length in an aggregate requirement increases the likelihood of matching classification keywords and phrases.)
5. The amount of text (token-level measure) recognized as a requirement is 100% for the delimiter-based analysis (by definition) and averages 66% for the grammar-based analysis (see (Vlas and Robinson 2011) for details). (Delimiter-based macro-requirements cover, by definition, all the text of a Feature Tracker post.)

As no artifact and no analysis are perfect, the RCNL classifier also has its limitations. The grammar-based strategy generates a number of false positives if a more general view of the OSS communication is considered. While all discovered micro-requirements correspond to the

[1] The results of the automated analysis provided many metrics. Among them are the counts of requirements for each classification, and the total number of classifications created per project and per project interval (183 days long). The projects averaged 6,788 total classifications and 395 new classifications each 183 days. Such analysis and other similar analyses are enabled by the automated classification.

versions of the SAO triple that the analysis considers, some of them can be considered to be false positives, such as elements of social communication, or sentences describing less important details of a major feature request, or restatements of the perceived value of a piece of functionality. This is natural given that the grammar-based strategy, due to its very nature, cannot capture any part of the context surrounding a micro-requirement. Therefore, a hybrid parsing strategy may be best. It can characterize an aggregate requirement and its supporting sub-requirements. However, a hybrid parsing strategy is not part of the scope of this research and is considered for the continuation of it.

5.4. Expert Analysis

To evaluate the two parsing strategies, I employ established techniques for performance evaluation of natural language processing tools in the fields of Information Extraction (IE) and Information Retrieval (IR) (Rijsbergen 1979; Frakes and Baeza-Yates 1992; Manning and Schütze 1999). The evaluation measures I use are precision, recall, and F-measure. Precision is usually expressed as a percentage value and represents the ratio of the correctly identified elements to the total number of identified elements. In other words, precision measures how many of the items identified by the automated classifier are correctly identified items. A high precision percentage is specific to a tool capable of identifying mostly correct items. Any mistake made by a tool in identifying correct items decreases the precision percentage. Recall is usually expressed as a percentage value and represents the ratio of correctly identified elements to the total number of correct elements in the dataset. In other words, recall measures how many of the items in the dataset are identified by the automated classifier. A high recall percentage is specific to a tool capable of identifying most of the correct items in the available dataset. Any correct item in the dataset that a tool doesn't identify will decrease the recall percentage.

Given the definitions for precision and recall, it is clear that achieving either 100% precision or 100% recall is easy. A tool can have 100% precision when it makes no mistakes. Therefore, a tool that identifies no items will have perfect precision. Similarly, a tool can have 100% recall when it identifies all correct items in a dataset. Therefore, 100% recall can be achieved by designing a tool that identifies everything from a dataset. Better precision can be achieved at the expense of recall, while better recall can be achieved at the expense of precision. The most appropriate way to evaluate a tool's efficiency is by using precision, recall, and a weighted measure of them. F-measure (sometimes called F-score) provides this weighted average (Rijsbergen 1979).

$$F - measure = ((\beta^2 + 1) \times P \times R) / ((\beta^2 \times P) + R)$$

where β indicates the weighting of precision and recall, P represents precision, and R represents recall. In the evaluation, precision and recall receive equal weighting, thus β is set to 1. Precision, recall, and F-measure can only be calculated if a key is available for providing an example of the ideal discovery and classification result. This key is called a gold (or golden) standard (or key) in the IR literature and represents the benchmark against which a tool's output is compared. A gold standard is normally created by experts in the field and is the result of successive passes through a sample data in order to improve its quality. Here, I define a requirements engineering expert as a professional with 5 or more years of experience in the field of requirements engineering. Ideally, a gold standard provides a perfect, error-free sample result. The gold standard I use for evaluation is created by a requirements expert and provides the output of an ideal, error-free requirements discovery and classification process. Producing a gold standard requires an expert and time, and thus is very costly in practice. It is impractical to have

an expert identify and classify thousands of requirements. However, a gold standard for sampled data allows for the accurate evaluation of an automated tool.

I calculate precision, recall, and F-measure as a result of comparing a gold standard to the output of the RCNL tool. For the grammar-based strategy, I randomly select 25 data segments from the 515 data segments in the dataset. Then, I randomly extract a short text sample (between 1 and 4 postings long) from each selected data segment. These 25 randomly selected sample texts are manually tagged with annotations corresponding to micro-requirements and the 23 classification types from McCall+. The expert tagged and classified at an average rate of 515.22 tokens per hour. At this rate, the RCNL tool is about 3,239 times faster than the human expert. The expert tagging process for the grammar-based strategy is designed to annotate sentence level requirements defined as a SAO pattern and the associated extending patterns (e.g. adjectives, qualifying phrases, secondary phrases, quantifying expressions, etc.). The delimiter-based gold standard is developed in a similar manner and is created at an average of 1,695.93 tokens per hour. At this rate, the RCNL tool is about 448 times faster than the human expert. These processing effort results are explained by two factors. First, as indicated in Section 5.2, the RCNL processing time for a grammar-based analysis is shorter than the one for a delimiter-based analysis due to the complexity and memory requirements of the processing (JAPE) rules. Second, the human effort required for the identification of requirements at the forum post level is naturally less than the one required for the identification of requirements at the sentence level, due to the level of detail associated with these two tasks. Consequently, the automated requirements discovery and classification with the RCNL tool is significantly faster than a human expert, especially in the case of a grammar-based approach.

To evaluate against a standard, first I post-process RCNL's output. I add at the end of the analysis pipeline a number of JAPE rules for labeling and grouping previously created annotations into annotations of interest for the evaluation process. In a second phase of post-processing, I add the annotations from the gold standard to a separate annotation set in the output of the RCNL tool. Last, I configure and execute the Corpus Quality Assurance (CQA) plugin, which is a GATE tool for computing evaluation measures. CQA computes recall, precision, and F-measure by comparing the RCNL's output to a key – the annotations imported from the gold standard. In our context, recall is most important because I want to find the requirements, after which further processing may occur.

The results of the gold standard analysis are summarized in Table 7. The last three columns show recall, precision, and F-measure. The middle four columns show specifically how the gold standard compares to the automated output of the RCNL tool. The "Process" column indicates which process' resulting annotations are being considered in the computing of the evaluation measures: (1) requirements discovery (identification of micro-requirements or macro-requirements in the dataset), (2) requirements classification (the classification of the identified requirements), or (3) requirements discovery and classification (the identification and classification of micro-requirements or of macro-requirements). The "Annotations Evaluated" column clarifies this context by indicating specifically which annotations are used to compute the evaluation measures. The results depict an efficient discovery process (F-measure of 83% for grammar-based requirements discovery). They also indicate a less efficient classification process (F-measure of 29% for classification). A number of factors explain these evaluation results.

First, the classification process is designed starting from McCall's model. McCall's quality model was developed in 1977 when information systems were different from today's information systems. This model might not reflect current open-source users' perceptions of quality accurately. Second, when there is a mismatch between a gold standard requirement and the output of the automated tool, cascading mismatches occur – a missed gold standard requirement also misses the associated gold standard classifications (averaging 2.4 for grammar-based strategy and 4.2 for delimiter-based strategy). Third, the semantic content is significantly determined by the overall context provided in the entire posting. The grammar-based strategy performs the analysis at a sentence level. Thus, much of this context is not available, and therefore, some classifications are absent.

When combining discovery and classification into one comprehensive evaluation effort, there are 24 annotations grouped together as one composite annotation. (See Discovery & Classification in Table 7). The instances within text of these 24 annotations represent the sum of the instances corresponding to micro-requirement and to the 23 classification criteria. For example, there are 104 matches for discovery and classification (second row), which corresponds to the sum of 69 matches from discovery (first row) and the 35 matches from classification (third row). Because discovery and classification (second row) lists the combined values of discovery (first row) and classification (third row), it is also expected that the effectiveness (expressed through recall, precision, and F-measure) of it would be between those of the two processes it is comprised of (discovery and classification). More precisely, it is significantly closer to the effectiveness of classification because in the combined evaluation the classification process contributes 23 annotations whereas the discovery process contributes only one annotation.

Table 7. Expert comparison measures.

Strategy	Process	Annotations					Recall	Precision	F-Measure
		Evaluated	Matching	Only in Gold Standard	Only in Output	Overlapping			
Grammar-based	Discovery	*Micro-Requirement*	69	22	74	168	0.92	0.76	0.83
	Discovery & Classification	*Micro-Requirement and 23 Classification Criteria*	104	342	455	274	0.52	0.45	0.49
	Classification	*23 Classification Criteria*	35	320	381	106	0.31	0.27	0.29
Delimiter-based	Discovery	*Macro-Requirement*	189	14	72	136	0.96	0.82	0.88

Overall, the results are encouraging. Although there are no other tools in either the researchers' or practitioners' domain to provide the exact same type of analyses, I note the similarity between these results and those from a 2006 study by Cleland-Huang, Settimi, Zou, and Solc (Cleland-Huang et al. 2006). In their study, Cleland-Huang et. al. explore approaches to non-functional requirements' (NFRs) discovery and classification. First, they start with exploring whether a pre-defined fixed set of keywords can be efficient in classifying NFRs. Their findings include recall between 61% and 80% and precision between 40% and 57% but also highlight a shortcoming of this approach – the difficulty of finding accepted sources of keywords for specific types of NFRs. My study identifies same challenge. Consequently, Cleland-Huang et. al. develop a NFR-Classifier that includes a

mining phase for keyword extraction from a training set. The extracted indicator terms are ranked and determine the two alternate extraction methods considered: (1) top K terms selected and (2) all terms selected for each NFR type. When top-15 terms are used, the classification of different NFR types exhibits different efficiency levels. Recall ranges between 51% and 98% (with an overall recall of 77%) while precision ranges between 19% and 37% (with an overall precision of 25%). My approach to classification is also keyword based but I use a different, pattern-based approach to requirements discovery. While I acknowledge the distinct characteristics of the two studies, it is important to also note the similarity of the evaluation values. My recall ranges between 31% and 96% while precision ranges between 27% and 82%. I conclude that the two studies implement distinct approaches and complement each other while proposing similarly efficient tools.

5.5. Benchmarking

Following the evaluation and assessment of RCNL tool's performance and accuracy, I explored the possibility of comparing RCNL's behavior against that of a market leader tool designed for performing same type of analyses. A number of artifacts have been proposed by the research or practice communities for the automatic or semi-automatic analysis of requirements. The great majority of these tools are designed to process data from a dataset of already elicited but not completely or properly specified requirements. Among the 18 tools that I studied, the most well-known and comprehensive tools are Requirements Elicitation, Capture and Analysis Process (RECAP), the Conceptual Modeling Environment/Process Implementation Methodology (ACME/PRIME), the First Requirements Elucidator Demonstration (FRED), or Requirements Elicitation Assistance System (REAS) (Edwards et al. 1995; Feblowitz et al. 1996; Kasser 2004; Elazhary 2010). In spite of their complexity and relative popularity, none of these tools can be

used to perform analyses similar to the ones enabled by the RCNL tool (requirements discovery and classification over informal, fully unstructured, NL communication texts).

Another tool studied is AbstFinder. This is an interesting example of the way NLP techniques can be used in the analysis of requirements-rich and informal textual data (Goldin and Berry 1994). However, AbstFinder does not compare to RCNL since it is a tool that provides minimal requirements elicitation support. Another example of the way NLP techniques can be used in the process of requirements discovery is provided by the Requirements Elicitor proposed in 2006 by Mala and Uma (Mala and Uma 2006). Their tool implements a set of 12 syntactic structures based on the SVO (subject-verb-object) pattern. The RCNL tool implements a wider set of SVO/SAO structures. Requirements Elicitor does not compare to RCNL because it is specifically designed to support the transformation of normalized sentences into message records. Therefore, it does not include the processing abilities of RCNL.

The most similar tool to RCNL is the one designed by Cleland-Huang, Settimi, Zou and Solc (Cleland-Huang et al. 2006). They propose a tool for detection and classification of requirements with application to early aspects of software development projects. In spite of the apparent similarity to RCNL, Cleland-Huang's design exhibits a number of major differences. First, the analysis process is semi-automated requiring extensive researcher intervention for module customization, and data pre-processing and post-processing. Second, it has a narrower scope because it is designed to discover and classify only NFRs, while RCNL can not only process all types of requirements but also be adapted to various types of NLP-based analyses. Third, analysis techniques used by Cleland-Huang et. al. in their design are statistical whereas RCNL is based on NLP techniques. Fourth, it is

designed for and has been tested on closed-source data and test data. RCNL is currently configured for the specifics of OSSD projects communication. Fifth, the analysis dataset requires pre-processing that converts and stores data into a database format of a specific type and configuration. RCNL tool can be fed the analysis data in the form of un-processed text files, the format in which current OSSD project communication is available. In spite of the differences highlighted here, the study by Cleland-Huang, Settimi, Zou and Solc has findings that include recall between 61% and 80% and precision between 40% and 57% for the discovery phase, which is aligned with this study's findings. Their classification efficiency when top-15 terms are used shows a recall ranging between 51% and 98% (with an overall recall of 77%) and a precision ranging between 19% and 37% (with an overall precision of 25%). This numbers are also in line with the findings of this study.

Concluding, despite the large number of requirements engineering tools available, there currently is no comparison tool for RCNL. All available tools exhibit characteristics which makes them significantly distinct from the RCNL classifier, thus making a benchmarking effort not possible at this moment.

5.6. Configurable Rule-Based Analysis

One of the objectives of this dissertation is, as mentioned before, to design, develop, evaluate, and propose a flexible rule-based analysis environment for NL text that can be used in a variety of domains and for analyzing a large spectrum of NL data. This can only be achieved if the analysis artifacts are highly configurable. The RCNL classifier and the requirements parsing ontologies it implements are specifically designed and developed to exhibit such behavior and to provide the researcher with a configurable rule-based analysis environment. This feature of the proposed artifacts distinguishes them from all the requirements engineering tools described in the previous section (Section 5.6. –

Benchmarking) and pictures the RCNL classifier and its associated ontology as a unique proposition for both the researchers and practitioners in the OSSD area.

The grammar-based and the delimiter-based requirements parsing ontologies (see Table 1 in Section 3, and Table 2 in Section 4, respectively) are designed as layered architectures in order to allow easy intervention. For instance, a researcher interested in analyzing data other than informal NL text will need to adapt rules corresponding to levels 0 and 1 of the requirements parsing ontology. A researcher interested in running an analysis other than requirements-based will need to modify rules corresponding to levels 3 and 4 of the requirements parsing ontology. Similarly, a requirements-based analysis for closed-source software development will require intervention at level 2 of the requirements parsing ontology. Lastly, a different classification model or approach requires intervention only at level 5 of the ontology.

The "divide et impera" (divide and conquer) approach is a well-known and accepted development approach in software development. It is used by software developers for creating software systems with a focus on modularity, as a means of offering improved flexibility, adaptability, expandability, and maintainability. The implementation of the two requirements parsing ontologies into a software analysis tool (the RCNL classifier) follows the "divide et impera" principles of developing a modular architecture. Each entity from each of the levels of the requirements parsing ontology is implemented in the software tool with a separate pattern or set of patterns. In certain cases, multiple distinct patterns and their corresponding JAPE rule files are created for same entity from the ontology in order to propagate the modularity-oriented focus downwards at the entity development level. A

clear separation of patterns of interest is continuously maintained throughout the entire design of the artifacts.

The patterns considered are encoded in the RCNL classifier as JAPE rules, as described in Section 3.2 and Section 4.2. Although a JAPE rule file can contain any number of JAPE rules, out of a total of 135 JAPE rule files created for the grammar-based strategy, 104 of them contain only one JAPE rule. Similarly, only 5 of the 20 JAPE rules files created for the delimiter-based strategy contain more than one JAPE rule. Most of the JAPE rule files that contain more than one JAPE rule (usually only 2 or 3 rules) do so because of functional reasons. For instance, the use of priority rule firing cannot be avoided with designing a sequential rule firing alternative.

The requirements-based analysis of NL text is performed by the RCNL tool in GATE through the execution of a sequential rule processing pipeline. The researcher loads the desired analysis plugins into the created pipeline. As described in Section 4.2, some of the available plugins are predefined in GATE while others (named entity transducers – NE transducers) allow the specification of custom designed JAPE rules. Figure 7 presents the example of an execution pipeline created for performing requirements discovery based on the delimiter-based strategy. An execution pipeline can contain any number of NE transducers, which allows for a modular development and execution of JAPE rules. NE transducers implement a different grammar in order to load special types of JAPE files designed for batch processing of JAPE rule files. I call these files “main processing resources” or MPR. Figure 8 presents the example of a NE transducer used for loading the MPR file designed for requirements discovery under the delimiter-based strategy. Note that the contents of the loaded MPR file are being displayed in the right-hand side pane. In this

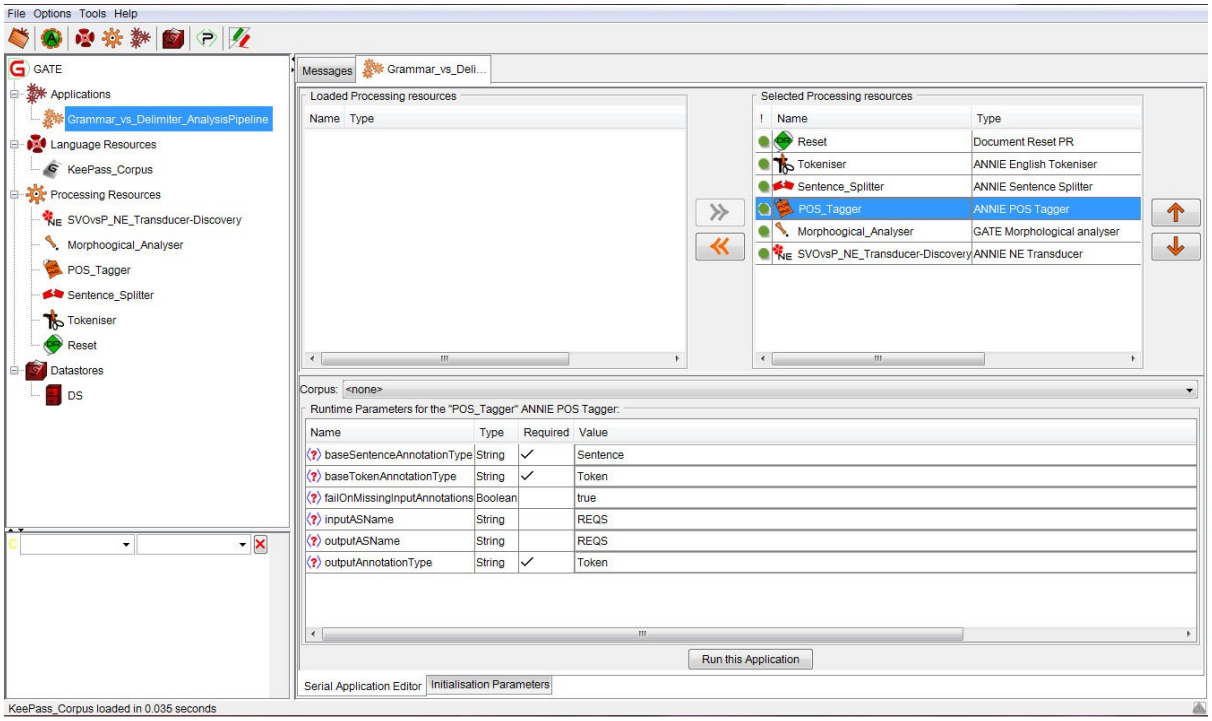

Figure 7. An execution pipeline for delimiter-based requirements discovery.

example, the delimiter-based requirements discovery MPR includes 20 JAPE rule files listed in the order in which they should be executed over the data corpus.

This modular architecture of processing resources allows for an easy handling and modification of analysis rules and the associated patterns. A researcher interested in exploring the individual impact of a certain rule or set of rules, or interested in testing variations to a designed NL analysis has a number of alternatives available for achieving this objective:

1. Modify the MPR file stored on his/her computer (permanent change), and re-initialize the associated NE transducer.
2. Modify the content of the MPR within the right-hand side pane shown in Figure 8 (temporary change within the analysis environment for analysis and testing purposes; the MPR file is not updated).

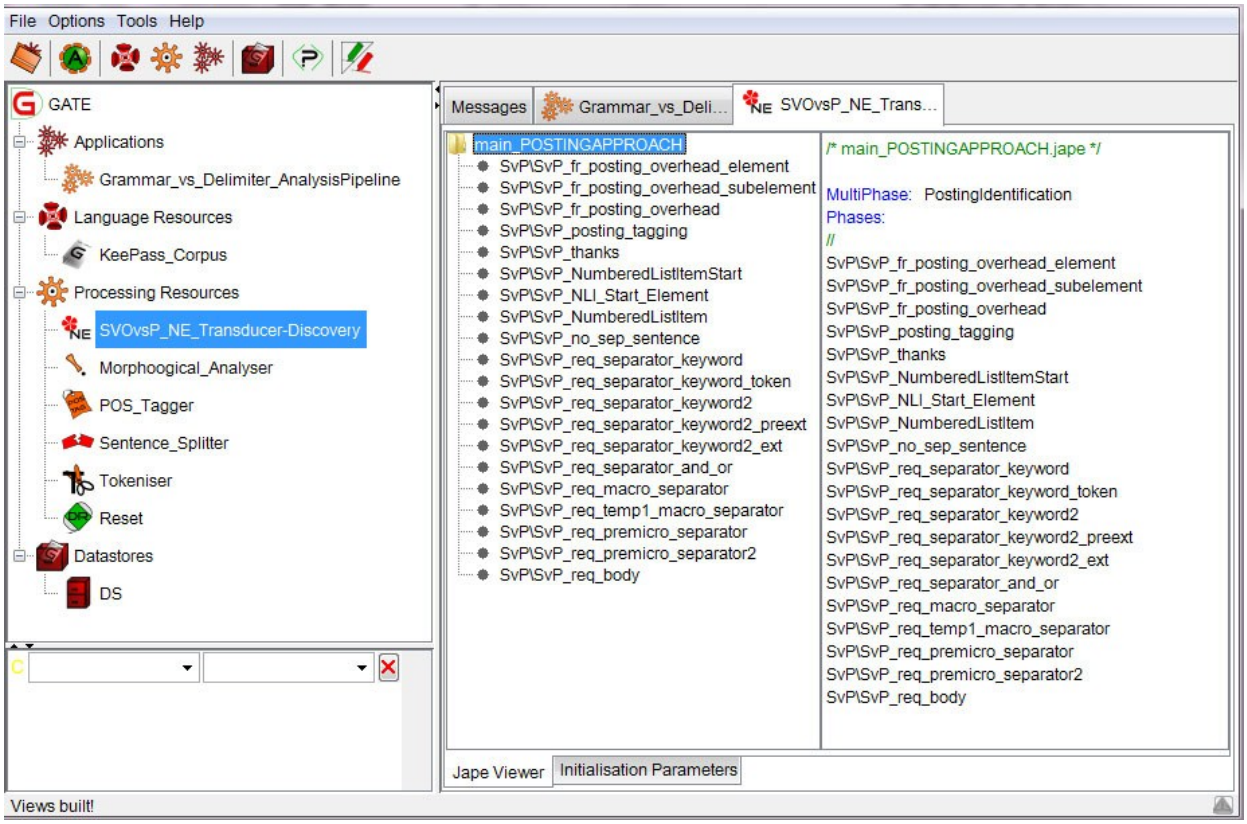

Figure 8. An execution pipeline for delimiter-based requirements discovery.

3. Create new analysis pipelines that load different plugins or plugins that are configured in a different manner (note the bottom right pane in Figure 7 for an example of plugin configuration settings).
4. Modify JAPE rule files stored on his/her computer (permanent change) and re-initialize the associated NE transducers.
5. Modify the content of JAPE rule files within the right-hand side pane shown in Figure 8 (temporary change within the analysis environment for analysis and testing purposes; the JAPE rule files are not updated).

The details presented in this section of the requirements-based analysis of NL text supported by the RCNL classifier and the ontologies it implements portray a highly configurable analysis environment. This contributes to proposing to the academic domain a

flexible tool that can efficiently be used in analyses of not only open-source communication but also other NL data sources. Next two sections in this dissertation show ways in which the RCNL tool can support different types of evaluation efforts and exploratory studies of OSSD.

5.7. Sensitivity Analysis

RCNL's ability to support highly customizable analyses in which various processing rules can be turned on and off enables a one-way sensitivity analysis of different patterns considered in the requirements parsing ontology. During this evaluation process, the researcher can assess the impact certain rules have on the tool's output.

For the one-way sensitivity analysis I selected 3 of the most complete and representative data windows from the dataset of OSSD projects. First, as highlighted in Sections 5.8 and 5.9, KeePass Password Keeper is indicated as potentially one of the most successful OSSD project from the dataset. Among the 13 segments of 6 months of data (data windows) for KeePass, I selected data window 3 because its analysis generates all types of annotations included in the requirements-based NL analysis. Second data window selected is phpMyAdmin's data windows 12. It also includes all types of annotations generated by the analysis despite being a data window characterized by less amount of communication. KeePass and phpMyAdmin are also selected because they have a relatively high requirements coverage ratios and stand out in the exploratory study presented in Section 5.8 as projects with a clearly defined focus which has a significant impact on the type of requirements they have (security-related features and database-related features respectively). Third data window selected is from the SourceForge.net project (data window 13) because this project provides largest data windows and is characterized by a relatively low requirements coverage ratio at the same time. The data window files selected from

KeePass, phpMyAdmin, and SourceForge.net are of size 82Kb (largest KeePass data window), 28Kb, and 115Kb respectively.

During the sensitivity analysis I tested the impact of 3 rules or sets of rules on the requirements discovery ability of the RCNL tool and 3 rules or sets of rules on the requirements classification ability of the RCNL tool. Table 8 presents the rules an rule sets tested, the analysis measure considered for assessing their impact, as well as the results of the one-way sensitivity analysis for the 3 data windows. The percentage change listed in last three columns represents the change recorded after turning off the rules.

Table 8. One-way sensitivity analysis results.

Rule or Rule Set Focus	Analysis Measure	KeePass Change (%)	phpMyAdmin Change (%)	SourceForge Change (%)
Filename, email, URL, technical writing	Requirements discovered	0.54%	0.46%	0.83%
Smileys		-0.08%	0.00%	0.06%
L2 - Qualification expressions		3.57%	1.83%	3.21%
Complete List	Classifications created	-4.08%	-1.49%	-3.83%
List item		-8.34%	-8.04%	-8.27%
Classification enhancements (McCall+)		-68.98%	-65.48%	-69.82%

At the lower level of analysis, common NL processing of informal communication involves tagging of tokens and parts-of-speech. This corresponds to levels 0 and 1 in the requirements parsing ontology. Here, I test the additions I make to the NL analysis provided by GATE. Specifically, I test the impact of annotating filenames, email addresses, specifications of URLs, and various types of technical writing. The results of the sensitivity analysis show that these types of elements of NL text have a low but consistent impact (less than 1% change but consistently positive change) on the results of the RCNL tool. The positive change in the number

of discovered requirements when these rules are turned off is explained by the role these technical pieces of text play as delimiters of text, or delimiters within an SAO triple. The tagging of text labeled as "smiley" has a very weak impact on the output of the tool. This highlights the difficulty of correctly interpreting the role played by elements of social conversation (delimiting effect or not) in open source communication.

At level 2 of the requirements parsing ontology I tested the composite effect of 5 qualification rules on the output of the RCNL tool. The 5 rules tested are the ones used for tagging expressions in text indicating preferences, beliefs, necessities, certainties, and suggestions. Qualification expressions of this type are numerous in NL text in open source and they have a significant and decreasing impact on the final number of requirements discovered. Qualification expressions contribute to the recognizing of qualifying phrases which can extend the text coverage of the already discovered requirements. This adds parts of the existing context surrounding a NL statement and generates a more lenient tagging style. Consequently, this might result in merging together discovered requirements with small text coverage and low proximity to each other. Therefore, tagging of qualification expressions results in an overall decrease of the total number of requirements discovered but, at the same time, contributes to an increase of the average text coverage of discovered requirements. This result is consistent across all 3 data windows used even though phpMyAdmin exhibits a weaker impact.

While testing the impact of rules on the requirements classification ability of the artifact, I first focused on lists and list items recognized in text. They receive a separate attention during classification, as described in the "Auxiliary Text Processing" Section. Complete lists are defined as an introductory phrase followed by a number of list items. During the classification process, the classification of an introductory phrase is extended to the entire list. Consequently,

turning off the tagging of pieces of text as lists naturally results in a decrease of the total number of classifications created during the classification process. The "Auxiliary text Processing" Section also explains that individual list items are classified if they contain a classification expression. Consequently, turning off the rules responsible for annotating list items results in a lower number of classifications being created during the classification process. The significant difference shown in Table 8 in percentage change between complete lists and list items is determined by the proportion of lists and list items in text. First, there is a larger number of list items because each list will include few list items or more. Second, in NL text not all lists have an introductory phrase and this leads to annotating only list items while there is no list recognized.

Last sensitivity testing considered is comparing classification efficiency between McCall and McCall+ classification. As described in Section 5.3 and shown in Table 6, the enhancements to classifying requirements based only on McCall's model result in more than doubling the classification efficiency. Here, the results show that turning off all classification enhancements generate over 60% less classifications. This is consistent with the results from Table 6 and confirms the importance of the classification enhancements.

A one-way sensitivity testing of rules from level 3 (entity) or level 4 (requirement) of the requirements parsing ontology is not possible because turning off any rules at these levels will render the RCNL classifier unusable. All rules in levels 3 and 4 are required for discovering requirements and they build on each other. Therefore, turning any of them off will generate zero requirements discovered.

5.8. An Exploration of OSSD Project Characteristics

The literature on open-source requirements is limited to either the associated processes or to the analysis of only parts of individual projects. Her, I explore the relationships exiting between

open-source requirements and characteristics of open-source projects (such as type, and software project success). I first develop a requirements-based taxonomy of OSSD project types and discover patterns linking between this taxonomy and project success. I also propose a classification of requirement types based on their representativeness in OSSD projects. This highlights the importance of various types of requirements in OSSD projects and helps identify exceptions determined by the specific area addressed by each project. Finally, I investigate the lifecycle of 16 OSSD projects and discover and discuss patterns of evolution for a number of requirements types.

Project success has been extensively explored in the literature on classical software development and has often been explained through the quality of the software product. According to Wiegers, a software project's success is mainly determined by characteristics associated with that project's requirements (Wiegers 2003). Studies that link success and software quality factors concluded that many OSSD projects are successful (Stamelos et al. 2002; Crowston et al. 2006). The success of open-source projects has also been acknowledged and studied in a wide variety of industries (Scacchi and Alspaugh 2008; Scacchi et al. 2009). This research differentiates itself from the existing literature on open-source by exploring the relationship between open-source requirements (e.g. requirements types, distribution, evolution) and OSSD project success.

Currently, there is a limited understanding of what types of requirements are present in OSSD projects. There is also a lack of studies exploring open source requirements and project lifecycle from a less process-oriented or participant-oriented perspective and with a higher emphasis on design and architecture-related characteristics. Another knowledge gap in open-source research is represented by the lack of methods and tools for the early estimation of open-

source software project success. This exploratory study comes to address these knowledge gaps and to propose the means to advance knowledge in this area.

Data Collection and Analysis

I select a set of 16 SourceForge projects (see Table 9) identified by a more strict selection criteria (more than 4 developers, more than 5000 downloads, and more than 700 feature request postings). The dataset includes projects that can be considered to be among most active and successful because they are selected based on the number of participants, the number of downloads, and the number of feature request posts.

Table 9. Selected OSSD projects for the exploratory study.

No.	Project Name	No.	Project Name
1	AWStats	9	PCGen
2	Compiere	10	phpGedView
3	FileZilla	11	phpMyAdmin
4	Fire	12	POPFile
5	Float's	13	SourceForge
6	Gallery	14	TikiWiki
7	KeePass	15	Tortoise
8	MegaMek	16	WinMerge

Data collection uses the online access offered by Notre Dame University to SourceForge data through the SourceForge Research Data Archive (SRDA) (Gao et al. 2007). Data collected is organized in 16 text files, one for each project. Each of the project files contains all feature request posts associated with that specific project, and listed in chronological order. The timestamp for each posting is also included in the data files.

The analysis of OSSD projects lifecycle requires a time-based analysis of data. I use the included timestamps to determine the duration of each project and I split up the project files into 6 months long data windows. The length of the last data window in each project is between 3 months and 9 months in order to include all feature requests postings available. For each data

window created, I process feature requests with the RCNL tool in order to discover and classify them. The results associated with last data window in each project might be slightly skewed upward or downward as a result of the different time length of the data window. The analysis of results takes into account this aspect.

The requirements-based analysis of OSSD projects lifecycle includes within project analysis and cross-project analysis. I explore the evolution of the number of requirements as a factor who shapes a project's lifecycle. I start with an analysis of the evolution of the overall number of requirements. This helps identify main patterns of project evolution. Next, I analyze the evolution of individual types of requirements throughout the duration of a project and identify patterns of requirements types' evolution. I also identify patterns of evolution for groups of requirements types. In the next step of analysis, I correlate requirements results with project-level attributes. Here I compare patterns of evolution across projects and discover the relationships between project type and type of evolutionary pattern. Finally, I place all patterns of evolution discovered in a broader project-level context in order to validate them. This more general context is constructed from additional project information such as project type and description, number of positive recommendations and awards, release dates and types, percentage of feature requests solved, and number of downloads. I collect this information from SourceForge.

Data Analysis and Findings

The set of requirements-based analyses I consider generate a number of interesting results. First I plot the overall number of requirements discovered per data window for the entire duration of the 16 projects. The resulting graphs indicate the existence of taxonomy of open-source project lifecycle types. I identify 3 main types: bell-shaped, half bell-shaped, and unstable.

I further classify the unstable type of projects as either “double-spiked” or “full unstable” projects. Figure 9 presents examples of these project types. In the dataset of 16 OSSD projects, 4 projects are of type “a” (bell-shaped), 3 projects are of type “b” (half bell-shaped), 6 projects are of type “c” (unstable: double-spiked), and 3 projects are of type “d” (unstable: full unstable).

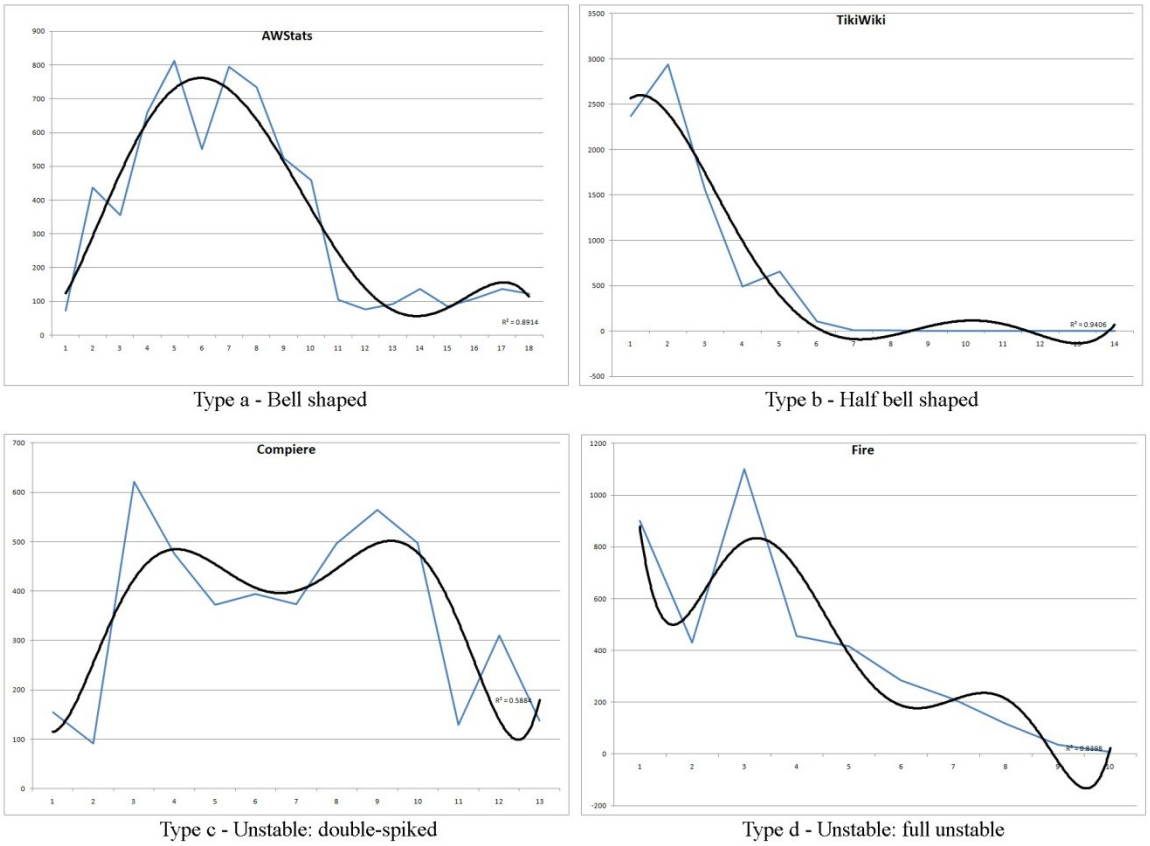

Figure 9. Taxonomy of OSSD project types.

The apparently unusual and unpredictable variations in the number of requirements per data window for a project can be explained by various events in the lifecycle of the projects. In order to explore this relationship, I extract information on the number and date of all types of software releases and on the number of new features implemented in these releases. On a plot of the number of new features released, one can find an expected behavior – OSSD projects’ lifecycles are continuously influenced by and react to the behaviors of communities surrounding them. Figure 10 presents the case of the KeePass project.

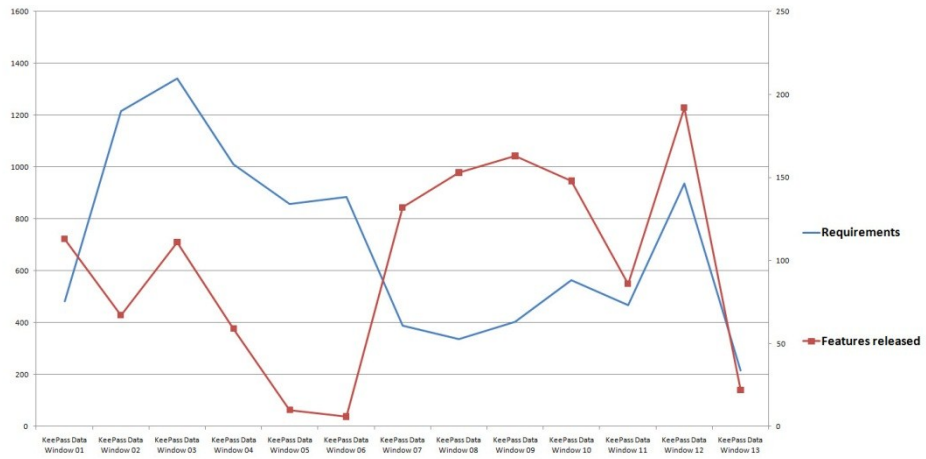

Figure 10. Evolution of features released and requirements for KeePass.

During first 3 years of the project (first 6 data windows), the number of desired features mentioned in project community's discussions (requirements identified in feature requests – blue line) is not matched by the number of features included in the project releases (red line). This suggests two effects: (1) an accumulation of desired features, and (2) a general discontent of the project community, which reflects in a decline of the number of feature requests being posted. Beginning of the 4th year marks a significant change as a large number of features is released. This seems to indicate an attempt to implement those features corresponding to the accumulation of requests from the first 3 years of the project. This also reactivates the interest of the community and fuels new discussions on the feature requests posting board. Therefore, the number of requirements discovered seems to be on a slightly positive trend. It can be assumed that the implementation of a large numbers of features in data windows 7 through 10 allowed the KeePass team to catch up with the requests for features. This explains the matching trends exhibited by the number of requirements discussed in feature request postings and the number of features released for the last part of the project's lifecycle. KeePass' lifecycle highlights a three-step evolutionary pattern that is also exhibited by other projects in the dataset. In step 1, the

project does not release enough features to cover for the amount of discussion in the project community. Step 2 indicates a move towards project maturity and is defined by a continuous effort to implement the accumulated number of features requested through frequent major releases. In Step 3, the project is mature and capable of matching current requests for new features.

The taxonomy of requirements types includes 23 criteria which are classified as "high representativeness (HR)," "medium representativeness (MR)," or "low representativeness (LR)." These three types are defined based on their associated percentage out of the total number of requirements within project. It should be mentioned here that requirements discovery and classification is performed using the grammar-based strategy, and thus, all discovered and classified requirements are within sentence micro-requirements. It can be found that 4 criteria consistently rank as HR, 7 criteria consistently rank as MR, and 12 criteria consistently rank as LR. These findings are presented in Figure 11. It is important to note here that project focus has an impact on the type of requirements that are normally classified as LR or MR within the project. For instance, KeePass is a password manager system and, therefore, access control (criteria 8) and access audit (criteria 9) are criteria that are essential to its success. Consequently, these two criteria are dominant among the types of requirements present the project. PhpMyAdmin is a database related project and storage efficiency (criteria 7) is one of its main concerns. These exceptions are highlighted in Figure 11.

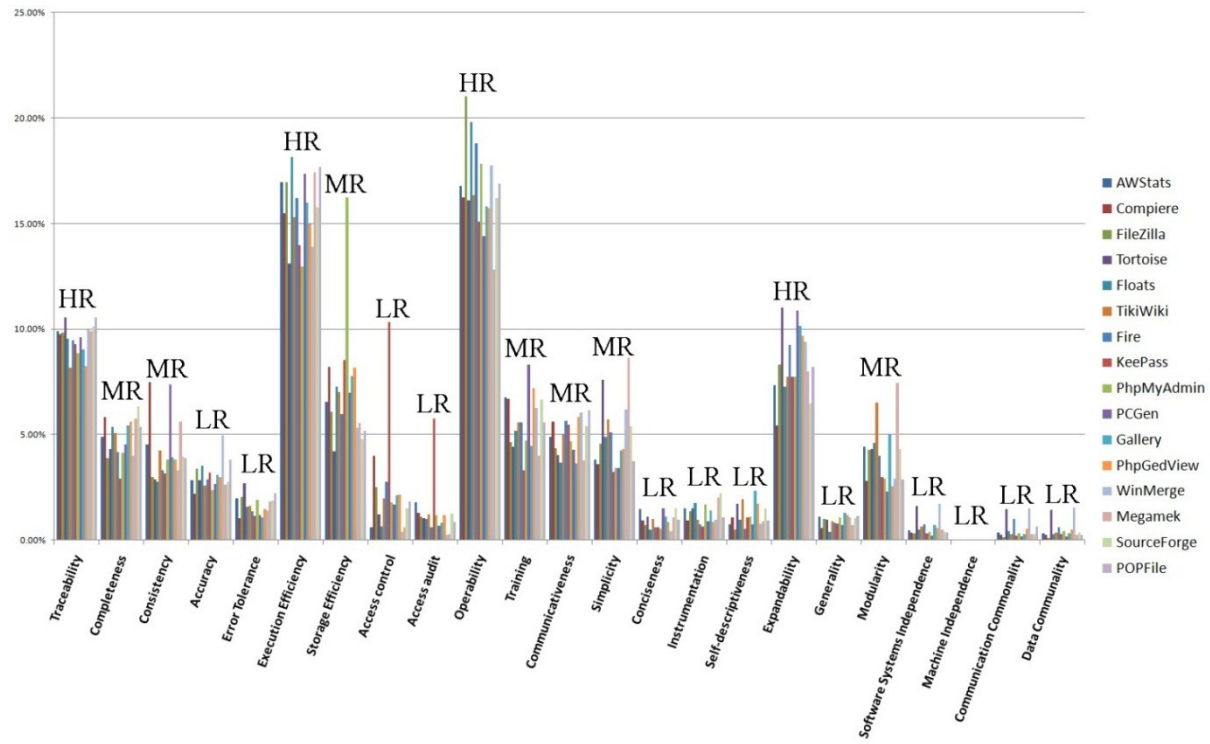

Figure 11. Taxonomy of requirements across OSSD projects.

An inspection of distinct requirements types throughout a project's lifecycle identifies few interesting patterns of evolution. First, we note a decrease in project activity (volume of postings in feature requests forums) immediately after a spike in traceability (criteria 1). The need to trace back requirements and other various characteristics of software during testing is well-known. Thus, increases of traceability can indicate an increase in testing activities. This is normally associated with the preparation of new major releases. With every major release, a large portion of the existing feature requests is addressed and the volume of discussion on feature requests forums is expected to decrease. We also note that traceability tends to become increasingly important towards the end of a project, sometimes surpassing other leading types of requirements. This is justified by the natural increase of testing activities in the last stages of software projects. For example, in the PCGen project, expandability (criteria 17) is one of the leading types in the first part, but traceability (criteria 1) is represented more than expandability in the second part. Similarly, access control (criteria 8) is a leading type during first part of KeePass project, while

storage efficiency (criteria 7) is a leading type during first part of Compiere project. In the last part, traceability (criteria 1) surpasses access control (criteria 8) and storage efficiency (criteria 7), respectively, in these two projects. Figure 12 presents these trends.

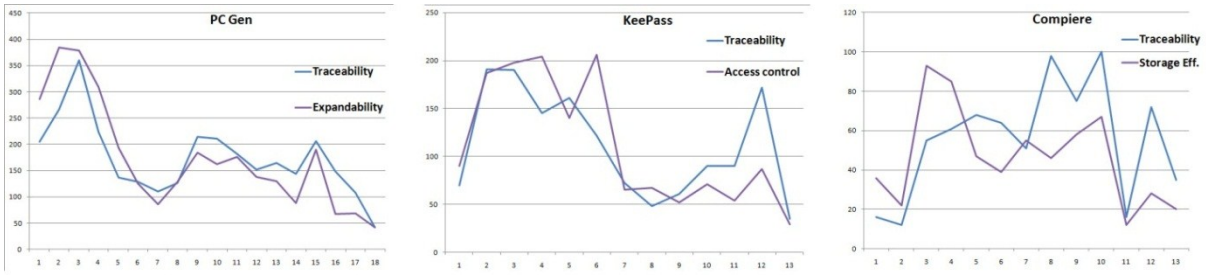

Figure 12. Evolution of traceability and other leading requirements types for PCGen, KeePass and Compiere projects.

Figure 13 presents a summary of main project-level information collected. This information includes few rough proxies for success. I sorted the set of 16 projects by the average number of weekly downloads reported by SourceForge. While none of these indicators provides a complete image of success, they complement each other for distinguishing more successful projects from the less successful ones. It can be concluded that Filezilla, KeePass, phpMyAdmin, and MegaMek, are among the most successful projects while Fire, and TikiWiki, are among the least successful projects. The four projects indicated as successful are of type "a" (bell-shaped) or type "c" (unstable: double-spiked). In contrast, the two projects identified as unsuccessful are of type "d" (unstable: full-unstable) and type "b" (half bell-shaped). The 2 unsuccessful projects exhibit a common behavior with regards to the taxonomy of within project requirements representativeness. Two of the successful projects exhibit an unusually high percentage of requirements types specific to their focus areas (KeePass – accessibility, phpMyAdmin – data storage and administration). One of the other successful projects (MegaMek, a turn-based board game) shows high percentages of modularity and simplicity requirements. This is explained by

the need for a simpler, modular architecture that games normally require in order to be able to offer expandability and flexibility over time.

Project Name	Project Description	Recommended by	Awards	Feature Reqs Solved	Patches solved	Weekly downloads	Reqs discovered
filezilla	FileZilla is a cross-platform graphical FTP, FTPS and SFTP client a lot of features, supporting Windows, Linux, Mac OS X and more. FileZilla Server is a reliable FTP server for Windows.	89%	2	x	x	714553	9231
keepass	KeePass Password Safe is a free, open source, light-weight and easy-to-use password manager for Windows. You can store your passwords in a highly-encrypted database, which is locked with one master password or key file.	84%	0	79%	99%	114068	7489
phpmyadmin	phpMyAdmin is a tool written in PHP intended to handle the administration of MySQL over the Web. Currently it can create and drop databases, create/drop/alter tables, delete/edit/add fields, execute any SQL statement, manage keys on fields.	84%	7	77%	92%	57516	6623
winmerge	WinMerge is a Windows tool for visual difference display and merging, for both files and directories. Unicode support. Flexible syntax coloring editor. Windows Shell integration. Regexp filtering.	93%	0	61%	97%	31148	4517
gallery	A slick, intuitive web based photo gallery. Gallery is easy to install, configure and use. Gallery photo management includes automatic thumbnails, resizing, rotation, and more. Authenticated users and privileged albums make this great for communities	79%	1	73%	76%	15214	12112
tortoise	TortoiseCVS is an extension for Microsoft Windows Explorer that makes using CVS fun and easy. Features include: coloured icons, tight integration with SSH, and context-menu interactivity.	92%	1	61%	95%	5163	5615
awstats	AWStats is a free powerful and featureful server logfile analyzer that shows you all your Web/Mail/FTP statistics including visits, unique visitors, pages, hits, rush hours, os, browsers, search engines, keywords, robots visits, broken links etc	83%	1	31%	63%	4169	6260
floats	FMA is a SMS Manager, Mobile Phone Monitor, Remote Control Agent, Phonebook Manager, Organizer, Fun and much more; whatever you want it to be, it is whatever a mobile phone should have :-) (Currently based on Sony Ericsson features set)	85%	0	28%	100%	2073	4124
pcgen	PCGen is a free open source RPG character generator (d20 systems). All datafiles are ASCII so they can be modified by users for their own campaigns.	92%	0	x	x	1829	15717
tikiwiki	Powerful multilingual Wiki/CMS/Groupware to build & manage your Wiki, Files/Image Galleries, CMS, Blog, Tracker/Forms, Forums, Directory, Polls, Surveys, Quizzes, Newsletters, Calendars, FAQs, Spreadsheets, Maps, Workflow, etc. - That all is Tiki!	87%	1	23%	61%	1550	8135
megamek	MegaMek is a networked Java clone of BattleTech, a turn-based sci-fi boardgame for 2+ players. Fight using giant robots, tanks, and/or infantry on a hex-based map.	96%	1	82%	98%	922	6635
phpgedview	PhpGedView is a revolutionary genealogy program which allows you to view and edit your genealogy on your website. It has full privacy functions, can import from GEDCOM files, and supports multimedia. It also simplifies family collaboration.	93%	0	49%	89%	766	6257
compiere	Compiere ERP+CRM is the leading open source ERP solution for Distribution, Retail, Manufacturing and Service industries. Compiere automates accounting, supply chain, inventory and sales orders. Compiere ERP is distributed under GPL V2 by CompiereInc.	48%	1	53%	97%	718	4615
fire	Fire is a multi-protocol instant messenger client for Mac OS X based on freely available libraries for each service. Currently Fire handles AOL Instant Messenger, ICQ, MSN Messenger, Jabber, limited IRC, Yahoo!, and Apple Bonjour communications.	62%	0	80%	0%	269	3964
sourceforge	Official Support and Documentation for SourceForge.net, provided by the SourceForge.net Service Operations Group (SOG).	82%	0	x	x	216	19832
popfile	POPFile is an email classification tool with a Naive Bayes classifier, POP3, SMTP, NNTP proxies and IMAP filter and a web interface. It runs on most platforms and with most email clients.	78%	1	88%	97%	32	8596

Figure 13. Project-level information.

Discussion of Exploratory Study Findings

This study presents the initial findings of a requirements-based analysis of a dataset of 16 OSSD projects. The findings include patterns of evolution for OSSD projects and for specific types of requirements identified in the study. The patterns are validated through logical reasoning and with additional information collected from SourceForge. Final findings determine an initial set of characteristics separating most successful projects from the less successful ones. This study's findings represent only an initial step towards achieving a more comprehensive

understanding of the relationships between the evolution of design and architecture elements and the success of OSSD projects. Further research is needed to validate and expand on these findings. OSSD project leaders can benefit from the results of this study by obtaining a means of determining if a project falls outside of the set of patterns commonly exhibited by successful projects or not.

A number of limitations characterize this initial exploratory study. First, the discovery of requirements in postings of feature requests is not an error-proof process. Similarly, the RCNL-based classification of requirements leaves some of them not classified and thus not contributing to the analysis. In spite of these limitations, I consider the positive evaluation results of the RCNL tool to justify its use in this study. The dataset consists of 16 OSSD projects that are selected based on criteria indicating success. While the selection criteria are not considered in the academic literature to be more than rough proxies of success, replicating this analysis on a larger dataset that includes less successful OSSD projects and projects from other domains is recommended and can yield interesting findings.

5.9. A Wave Theory of Requirements Innovation

This study demonstrates how the RCNL tool can aid in theory building. From its application to 16 OSSD projects, I conjecture a simple wave theory of requirements innovation: innovations expressed in requirements appear as a wave that is reflected in a subsequent wave of features that is reflected in a subsequent wave of product downloads.

Project managers face questions similar to the following. Given a project in the early or middle part of its lifecycle, should more resources be provided to contribute to the project's success? Should a new tactic be applied to fix the project? Alternatively, should de-escalation be attempted because the project is trending toward failure? One might be tempted to apply conventional analysis to answer these questions. For example, comparing

a COCOMO II nominal profile against the actual resources can reveal that a project has too short a schedule with too few resources. Such analysis, however, is inappropriate for OSSD projects, whose resource model is not represented in conventional software models like COCOMO II.

One could look to the models of open source researchers, who have made some progress towards linking project qualities with project success. Such works, summarized in Section 2.5, are preliminary. They depend on directly observable metrics, such as number of developers, number of bugs, number of patches, etc. There has been less effort applied to understanding the meaning of what open source developers do. Are they working coherently toward a commonly understood project release? Or are they thrashing about without a coherent theme? Successful OSSD projects transition through three common steps to produce a new themed release: innovate, improve, and deploy (Gamma and Beck 2004; Fitzgerald 2006).

This study is exploring techniques aimed to understand what open source developers are doing through analyses of their documents. Parsing techniques are applied to understand documents in terms of pre-defined models, such as models of requirements or distributed collaboration. These analyses may help us to address questions, such as the ones above.

In the work presented herein, I assume a requirements engineering perspective: requirements are in the topmost critical factors for project success, thus their analysis provides insight into a project's success. I look to requirements qualities to assess project qualities in the early or middle part of its life cycle.

The approach we demonstrate herein is to:

1. Discover OSSD requirements
2. Classify OSSD requirements
3. Characterize trends of the classified requirements into requirements factors
4. Correlate requirements factors with project qualities that may relate to project success

This work demonstrates the approach to steps 3 and 4 through an analysis case study. This exploratory work is designed to test the viability of the approach. Moreover, it is difficult to assess its effectiveness, in part, because OSSD project success is poorly defined, as are the requirements factors associated with success.

This study shows how OSSD projects can be analyzed according to two aggregated temporal requirements qualities in order to provide an indication of project success. The two main requirements factors considered are:

1. *Requirements development cohesion (RDC)*

   This measures the variations in count of different requirements types being developed within a period. A project period having low RDC means that developers are dividing the attention equally among all requirements types. In contrast, a project period having high RDC means that developers are focusing their attention on a few requirements types.

2. *Requirements traceability focus (RTF)*

   This measures the relative emphasis that developers place on traceability compared to other requirements qualities. A project period having high RTF indicates that developers are trying to understand a project. In contrast, a project period having low

RTF indicates that developers are focusing their attention other activities, presumably including product innovation.

These factors are obtained directly through natural open source artifacts. We believe that they have implications for the OSSD project success, as illustrated next through the analysis of 16 OSSD projects. The case-study demonstration of the four-step approach to NL open source analysis led to the conjecture of a simple wave theory of requirements innovation:

Innovations expressed in requirements appear as a wave that is reflected in a subsequent wave of features (in software), that is reflected in a subsequent wave of product downloads. Developers that stumble over one of these steps will likely see a reduction in product downloads.

This theory is consistent with the dataset of 16 OSSD projects, but remains a conjecture for more comprehensive analysis.

Innovation in Software Development

In software projects, innovation is a prerequisite for success. Much IS research on innovation considers the open-source paradigm (Hippel and Krogh 2003; Krogh and Hippel 2006), organization (Lyytinen and Rose 2003), or IT field as a whole. This leads to longitudinal effects, such as diffusion patterns (e.g., S-curve) (Bejan and Lorente 2011). More generally, technological innovation has been considered an exploratory process of integrating previously enumerated design elements (Jantsch 1967). Some have suggested that innovation can be modeled as decade long waves of innovation followed by lapses in innovation (Mansfield 1983).

Here, we are more interested in how a small open-source team innovates. Thus, the team and the individual are the units of study. Innovation in small teams may reflect innovation of the larger organization or field. In particular, small team innovation may

occur in waves. Following the approach of Jantsch, we assume that teams innovate by exploring and then integrating previously enumerated elements (Jantsch 1967). However, assuming the team has limited resources (i.e., members) they alternate between innovation and other project activities (Berkhout and van der Duin 2007). This results in a sequence of small innovation waves, many of which are realized as software features. For this case study, we assume that the innovative exploration occurs at the requirements-level, and is subsequently realized in the software through implemented features. We look for such innovation by looking for evidence of exploratory requirements development – the specification of requirements that appear to be innovative. Such requirements are integrative in that they reference multiple types of requirements.

Methodology and Data Collection

The methodological approach to theory building from SourceForge project data we demonstrate herein has four steps:

1. Discover open source requirements

   Use RCNL to identify requirements in open source documents. In particular, we limit our search to the NL text found in the Feature Request forum of each project on Source Forge.

2. Classify requirements

   Use RCNL to classify requirements. RCNL uses an extended McCall's model of 23 requirement qualities (McCall et al. 1977).

3. Characterize trends of the classified into requirements factors

   The longitudinal project data is divided into data windows, (w_1, w_2, …, w_n). The requirements in each window are classified. Then, derived factors, such as RDC and

RTF (from Section 5.7) are computed. Finally, their trends of consecutive windows (e.g., ΔRDC, ΔRTF) are computed.

4. Correlate requirements factors with project qualities that may relate to project success

   Finally, the derived factors are plotted, correlated, and otherwise compared as part of the exploration of relationships.

The dataset is comprised of the feature request posts from 16 OSSD projects, as listed on Table 8. The data collected is grouped in 16 text files (with sizes ranging from 229Kb to 2,304Kb), one for each project. This is the same source of data used to validate RCNL (Vlas and Robinson 2011; Vlas and Robinson 2012), which simplifies comparison and ensures validation of the requirements classification.

The analysis of OSSD projects lifecycle requires a time-based analysis of available data (data windows). We use the included timestamps to determine the duration of each project and we split up the project files into 6-month long data windows. The analysis of projects includes within and between project analyses. We explore the evolution of the number of requirements factors that shape a project's lifecycle.

Requirements Development Cohesion

Figure 14 shows a stacked graph of requirements variance for 14 projects (two of the projects did not have twelve 6-month windows). For each project, requirements variance is calculated as follows:

$$\sigma_{REQ} \stackrel{def}{=} \text{standard deviation } (R), \quad \text{where each } r_i \text{ is the count of requirements of type } i \qquad (1)$$

$$\Delta\sigma_{REQ} \stackrel{def}{=} d\sigma_{REQ}/dt = (\sigma_{REQ1} - \sigma_{REQ2}) / (t_2 - t_1), \quad \text{where } t_i \in \text{sequential data windows} \qquad (2)$$

Thus, σ_{REQ} measures RDC as the variations in count of different requirement types within a data window. A project period having low σ_{REQ} means that developers are dividing the attention equally among REQ types. In contrast, a project period having high σ_{REQ} means that developers are focusing their attention on a few requirement types. Theory suggests that this occurs during the exploratory process of learning and innovation. We are interested in $\Delta\sigma_{REQ}$ – a large $\Delta\sigma_{REQ}$ suggests a transition in the project requirements cohesion.

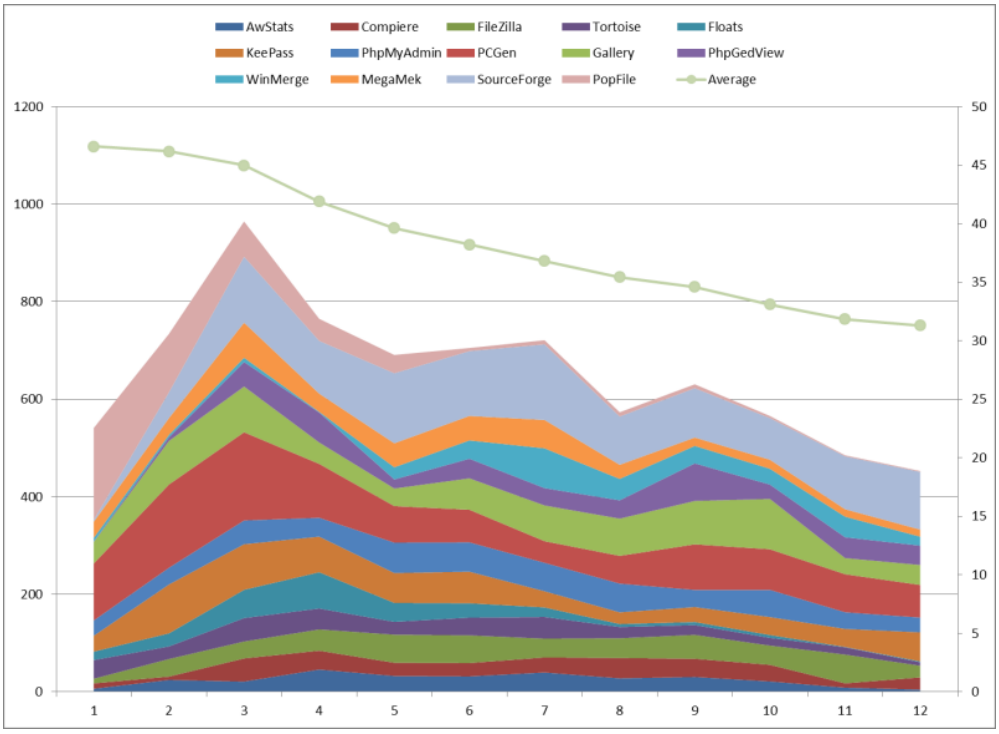

Figure 14. Stacked graph of requirements variance with average as line (Top, scaled right).

The line graph at the top of Figure 14 shows the average σ_{REQ} for 14 projects. Notice it has a negative slope, showing that, over time, projects tend toward equal treatment of requirement types. A careful analysis of the Figure 12 shows that some projects show waves of σ_{REQ}, revealing cycles of innovation followed by consolidation.

Figure 15 shows a (solid) line graph of KeePass's σ_{REQ} for 13 6-month data windows. Notice that the wave peaks at points 2, 6, 10 and 12. These suggest innovation in KeePass

as the developers focus on a few requirements types that are central to new product features.

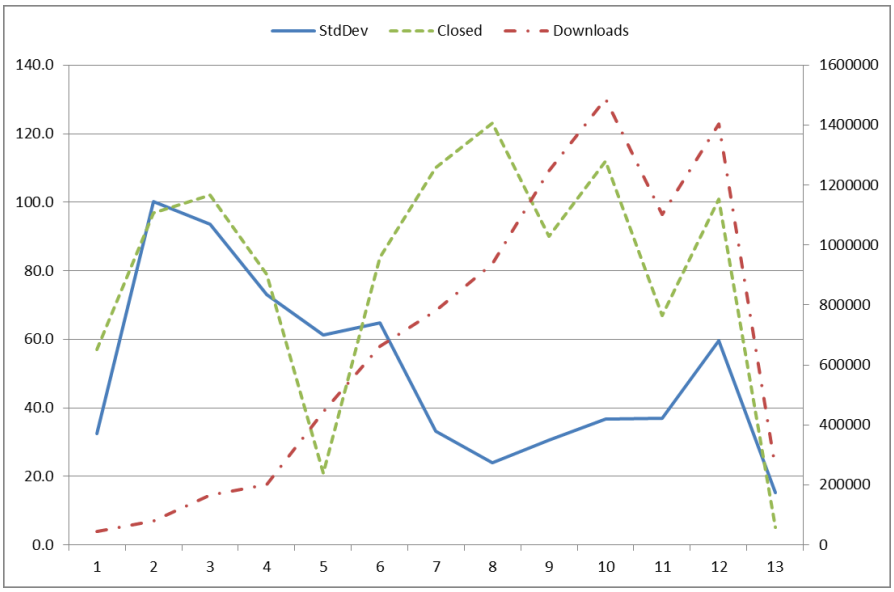

Figure 15. Requirements variance and Closed features (scaled left) with Downloads (scaled right) for KeePass.

The closing of feature requests marks the inclusion of new features in a release of the software product. In Figure 15, the Closed (dashed) line shows the count of feature closings. The feature closings line also has wave peaks at 2, 3, 7, 8, 10, and 12. It's interesting to note that some Closed wave peaks seemly reflect prior σ_{REQ} wave peaks. Theory suggests that, a successful innovation effort (σ_{REQ} peak) results in a subsequent feature (Closed peak). Moreover, when the team works to close a feature, it devotes less effort to innovation (assuming a relatively fixed number of developers). Thus, as Closed increased σ_{REQ} decreases.

These relationships between σ_{REQ} and Closed seem to hold (roughly) in Figure 15. Checking for correlation between the σ_{REQ} and Closed values using Pearson's correlation coefficient gives us $\rho_{\sigma,closed} = -0.42$, indicating a weak negative correlation. This is expected given that the theory suggests an inverse, time-shifted weak correlation –

especially true because some innovations will not be finalized as a product feature, and thereby create a missing feature peak.

Figure 15 also shows the number of downloads, as a (dashed-dotted) line graph. Just as waves of innovation (σ_{REQ}) lead to subsequent waves of product features (Closed), product features should lead to subsequent waves of downloads. Again, checking Pearson's correlation coefficient gives us $\rho_{closed,downloads} = 0.43$, indicating a weak positive correlation. Again, this is expected given that the theory suggests a time-shifted weak correlation – especially true because some features will not sufficiently interest users to warrant a download.

Table 10 shows the correlations for the eight projects that had sufficient data (e.g., feature-closed statistics) for analysis. The column headings are defined as follows:

- $\rho_{\sigma, closed}$

  Pearson's correlation coefficient between σ_{REQ} and the number of Closed (features)

- $\rho_{closed, downloads}$

  Pearson's correlation coefficient between the number of Closed (features) and the number of Downloads

- Features Solved

  The number of features requests "solved" through new or modified code (excluding "duplicate" or "dropped" feature requests)

- Patches Solved

  The number of patch requests "solved" through new or modified code (excluding "duplicate" or "dropped" patch requests)

- Closed/ Reqs

  The ratio of the number of Closed (features) to the number of discovered requirements (by the RCNL parser)

- Weekly Downloads

  The number of weekly downloads (from Source Forge). Downloads is used as a proxy for project success because: (1) it represents user interest, and (2) indirectly represents usage, and (3) provides a quantitative comparable metric for our dataset.

Table 10. Project correlations for requirements variance, closed features, downloads, and related project attributes.

Project Name	$\rho_{\sigma, closed}$	$\rho_{closed, downloads}$	Features Solved	Patches Solved	Closed/ Reqs	Weekly Downloads
awstats	-0.20	-0.40	31%	63%	0.047	714,553
compiere	-0.08	-0.18	53%	97%	0.107	114,068
filezilla	-	-	-	-	-	57,516
fire	-	-	80%	0%	-	31,148
floats	-	-	28%	100%	-	15,214
gallery		-	73%	76%	0.136	5,163
keepass	-0.40	0.43	79%	99%	0.140	4,169
megamek	-0.10	-0.01	82%	98%	0.069	2,073
pcgen	-	-	-	-	-	1,829
phpgedview	-0.40	0.09	49%	89%	0.107	1,550
phpmyadmin	-0.26	0.17	77%	92%	0.157	922
popfile	-	-	88%	97%	-	766
sourceforge	-	-	-	-	-	718
tikiwiki	-	-	23%	61%	-	269
tortoise	-	-	61%	95%	-	216
winmerge	0.06	-0.19	61%	97%	0.161	32

When taken as a whole, with the caveats of time-shifting and failures in the process steps (i.e., failure to implement an innovation as a close feature), the Table 10 $\rho_{\sigma, closed}$ and $\rho_{closed, downloads}$ values suggest that this theory is worth more exploration. Importantly, for our tooling efforts, it appears that our processing steps (discover, classify, characterize, and correlate) will support exploration and confirmation of open source development theories through analysis of their documents.

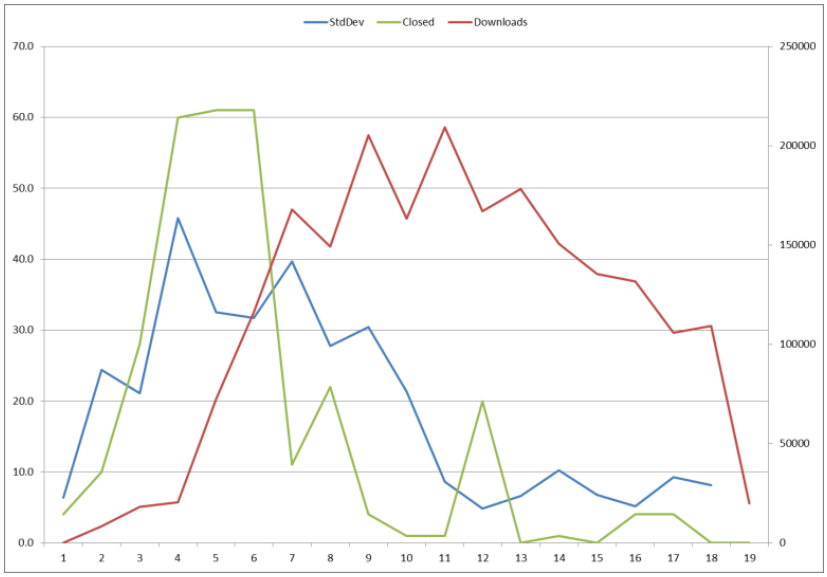

Figure 16. Requirements variance (StdDev) and Closed features (scaled left) with Downloads (scaled right) for AwStats.

Consider AwStats from Table 10. The value of $\rho_{closed,downloads} = -0.40$ seems to present a counter example. Let's also consider Figure 16, which graphs requirements variance, Closed features, and Downloads for AwStats.

Notice that there are relatively few closed features after data window 10. In comparing the waves of innovation, indicated by requirements variance (StdDev), with the wave of closed features, we see that the peaks of innovation are not reflected in subsequent feature closings. In comparing with other projects, AwStats has the third lowest percentage of feature requests closed at 31%, where the mean is 60%. It also has the second lowest percentage of patches solved, at 63% where the mean is 82%. Thus, it seems that AwStats is an outlier in the development process when compared with the other projects. The negative $\rho_{closed,downloads}$ correlations of Compiere and WinMerge may be explained in similar fashion.

Consider mapping $\rho_{closed,downloads}$ onto three values:

- Low $= \rho_{closed,downloads} < -0.15$

- Medium = $-0.15 \leq \rho_{closed,downloads} < 0.15$
- High = $\rho_{closed,downloads} \geq 0.15$

Projects with high $\rho_{closed,downloads}$ are consistent with the σ_{REQ} innovation wave theory. The others may have other factors that prevent innovative features from increasing downloads. Using the attributes of Table 10 as inputs, we applied decision tree data-mining to derive the following classification rules:

1. If $\rho_{\sigma, closed} > -0.26$, then $\rho_{closed,downloads}$ = Low
2. If $\rho_{\sigma, closed} \leq -0.26$, and …
    a. Closed/Reqs > 0.107 then $\rho_{closed,downloads}$ = High
    b. Closed/Reqs ≤ 0.107 then $\rho_{closed,downloads}$ = Medium

These rules cover the 7 projects (having $\rho_{\sigma,closed}$) with only 1 misclassification. The rules support the theory in that that $\rho_{\sigma, closed}$ affects $\rho_{closed,downloads}$. Additionally, these rules suggest that Closed/Reqs affects $\rho_{closed,downloads}$. This helps to explain why AwStats, Compiere, and WinMerge do not have increased downloads with increased feature closing. These aberrant projects have too small of Closed/Reqs ratio – too many requirements are being considered relative to the number of features being closed. This suggests that too many requirements ideas being discussed are reducing the effort to close features.

Requirements Traceability Focus

Traceability plays an important role in project management. As we show next, more emphasis on traceability than on operability may further explain why AwStats, Compiere, and WinMerge appear to have aberrant development practices.

By following a trace, developers improve their understanding of the project and its evolution. During testing, developers will trace from test cases back to requirements as part

of verification. Although open source methodologies rarely tout traceability – for system integration for example – they do promote the benefits of unit testing, which requires simple, direct traceability from test case to code.

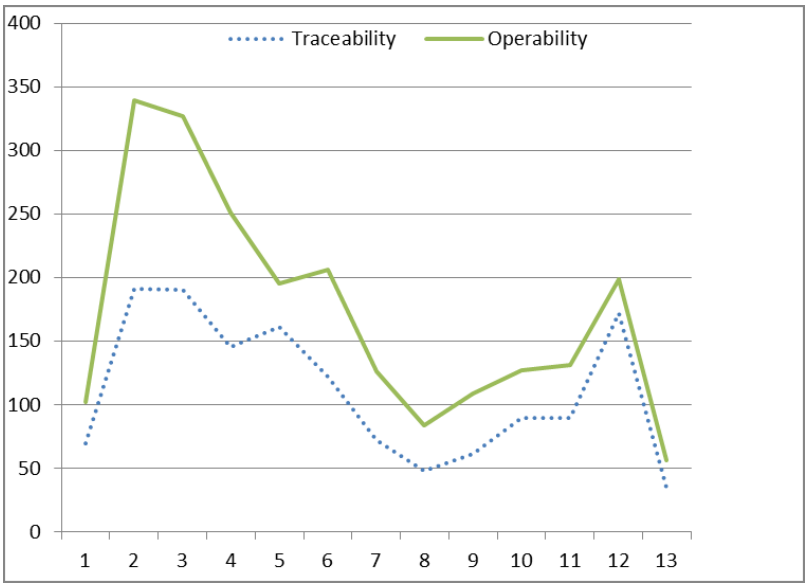

Figure 17. Evolution of Operability and Traceability in KeePass.

Our analysis reveals that open source has a greater emphasis on operability than on traceability. Both Figure 17 and Figure 18 show that KeePass and AwStats have more operability requirements than those addressing traceability. However, there is an interesting difference in the graphs. Notice that graphs of operability and traceability become closer around the 11th 6-month data window for AwStats – for their developers, traceability becomes nearly as important as operability.

Figure 19 shows this distinction more clearly by graphing the ratio of operability/traceability for KeePass, AwStats, Compiere, and WinMerge (in this study, RDF is the ratio of operability/traceability). Notice that the ratio increases substantially at point 11 for AwStats, while KeePass is mostly constant throughout the development. The other 2 projects, Compiere and WinMerge, similarly have points where their ratio raises above

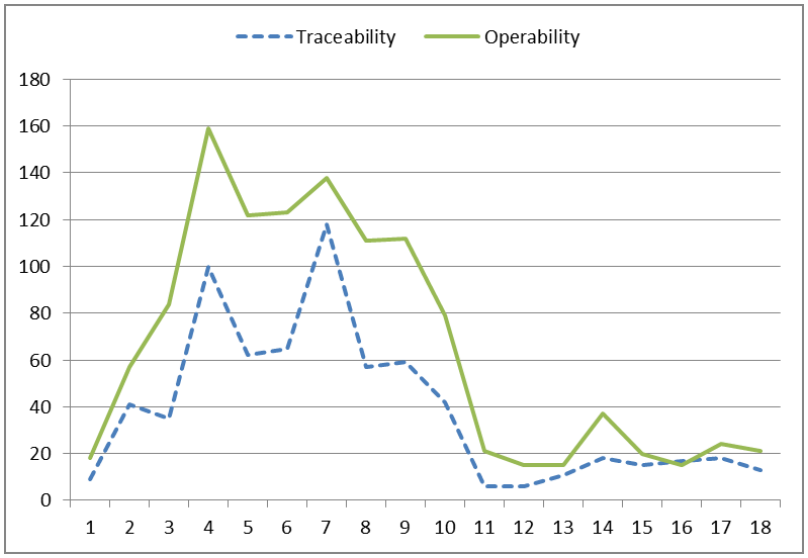

Figure 18. Evolution of Operability and Traceability in AwStats.

their average. Thus, the 3 projects that have $\rho_{closed,\ downloads} > 0$ (and thus seem inconsistent with the σ_{REQ} innovation theory) all have spikes in their operability/traceability ratio. When this distinction is considered, the theory is consistent with the data set.

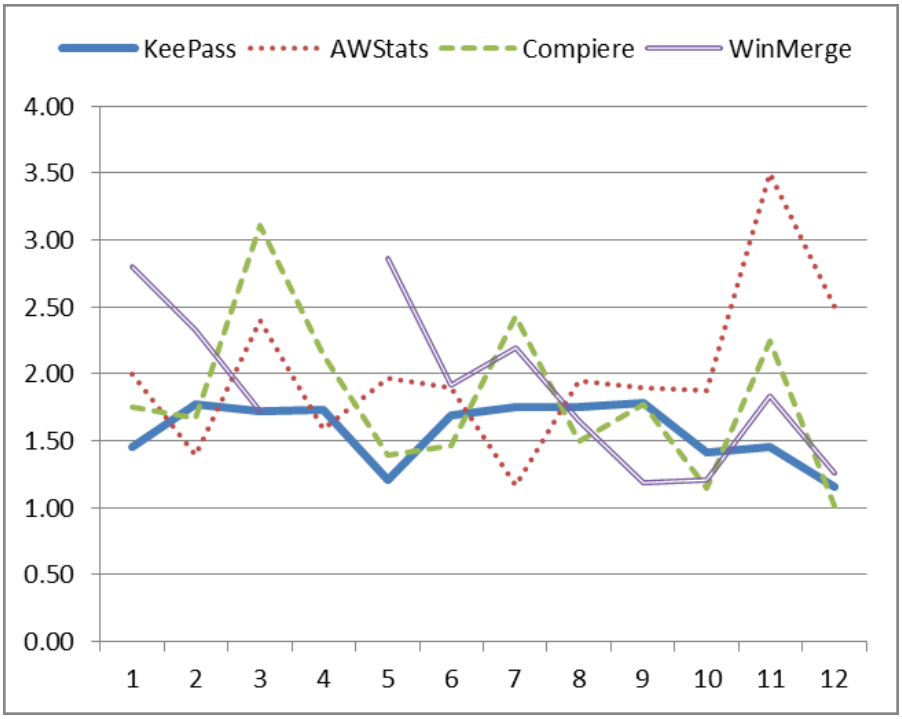

Figure 19. The operability/traceability ratio for four projects.

This increased emphasis on traceability is consistent with those projects that fail to convert many new requirements into implemented features. In terms of the preceding metrics:

- $\rho_{\sigma,\ closed}$ is weakly positive, indicating difficulty in converting innovations (σ_{REQ}) into closed features
- Closed/ Reqs is low (with Closed low), indicating more emphasis on discussing requirements rather than on implementing them
- Operability/traceability has spikes (with operability low and traceability high), indicating that traceability, and thus understanding the development and evolution, has become an issue
- $\rho_{closed,downloads} < 0$ (with Closed low and Downloads low), indicating users are not so interested in downloading the newly implemented features

Together, these suggest that, at some point, these projects have difficulty converting abstract requirements innovation (σ_{REQ}) into delivered functionality (Closed high and Downloads high).

Discussion of the Exploratory Study Findings

The previous sections summarize our preliminary analysis of 16 OSSD projects using NL requirements parsing and RCNL classification. We began this analysis to show how the RCNL can be used to analyze relationships among open-source documents. Because of this analysis, we have come to posit the σ_{REQ} innovation theory, which conjectures a sequential, wave-like process from requirements innovation (σ_{REQ}) to closing features to increased downloads. Consequently, we believe that we have shown how RCNL can aid in theory formation.

The σ_{REQ} innovation theory remains a conjecture until more data can be analyzed and more formal modeling of the time-shifted process correlations can be done. Additionally,

underlying assumptions should be validated. For example, it should be validated that increased σ_{REQ} activity results in increased innovation, rather than simply more randomized requirements. Likewise, it should be validated that increased operability/traceability spikes (with operability low and traceability high) is indicative of developers having trouble converting feature requests into closed features. Such detailed validation may require a grounded theory approach to analyzing the meaning of the underlying artifacts. In the meantime, however, RCNL does provide some indication that these assumptions hold based on its prior validation.

This article demonstrates how a NL requirements parsing and RCNL classification can be an aid to understanding what open source developers are doing through analyses of their documents. The work assumes a requirements engineering perspective: requirements are in the topmost critical factors for project success, thus their analysis provides insight into a project's success. The work looks at requirements qualities to assess project qualities in the early or middle part of its lifecycle. The approach assumes four common steps:

1. Discover open source requirements
2. Classify open source requirements
3. Characterize trends of the classified requirements into requirements factors
4. Correlate requirements factors with project qualities that may relate to project success

The resulting correlations provide insights into how open source developers do their work.

This article presents a case study of this approach, which posits the theory that innovations expressed as requirements appear as a wave (in quantity) that is reflected in a subsequent wave of feature closures, that is reflected in a subsequent wave of product

downloads. Developers that stumble over one of these steps will likely see a reduction in product downloads. This theory is consistent with the dataset of 16 OSSD projects, but remains a conjecture for more comprehensive analysis. The small sample size demands further analysis.

6. Discussion and Conclusions

This dissertation contributes to research and practice of OSSD. A systematic method for discovery and classification of requirements in OSSD projects is currently not available. Such a method enables important improvements, such as: (1) better understanding of open-source requirements, their types and lifecycles, and (2) better understanding of project scope, goals, and overall project direction. Such understanding in turn leads to better understanding and improvement of both OSSD project, but also more traditional software development. Moreover, the set of artifacts designed, developed and proposed in this dissertation (method, model, and tool) are specifically created as flexible and highly adaptable artifacts since they comprise a software analysis framework with potential future applicability in a wide set of domains. This framework is currently customized to meet the specific characteristics of OSSD but its requirements-based NLP analysis techniques and its architecture can be adapted to the specifics of other software development environment or methodology.

This research study provides few specific contributions:

1. A grammar-based design of software automation for the discovery and classification of natural language requirements
2. Two alternative parsing schemes implemented within the design
3. Requirements discovery, classification, and analysis of 30 OSS projects

4. An exploratory study on the impact of requirements types and evolution on OSSD project success
5. A conjecture wave theory of requirements innovation

Together, 1 and 2 above affirm hypothesis *H1.1: A multi-layered grammar, varying in domain specificity, can be constructed for the automated requirements discovery and classification of requirements contained within software informalisms of Open-Source Software Development projects*. In total, these contributions provide a path for subsequent empirical studies of OSS requirements and enable subsequent software tools facilitating automation of requirements traceability analysis in support of IS development process studies. The RCNL classifier provides a solution to existing industry problems and an alternative to existing methods that require substantial input from the researcher. The RCNL classifier runs autonomously. However, users and researchers may choose to customize rules from the top-most layers of the six-layer ontology to work most effectively with new datasets. Although I did develop and test it on a large SourceForge dataset, it may be that other OSS data or traditional software artifacts require changes to the lower levels of the parser (levels L0 through L2 in Table 1 and Table 2). To adapt the RCNL classifier to another quality model (other than McCall), only level 5 (L5) rules must be modified.

The adapting of the RCNL classifier to various domains and datasets is possible due to its highly customizable nature. However, I acknowledge the unstable nature of the emergent grammar used in the OSS communication that I analyze in this study. Consequently, I recognize that the artifacts developed here provide only a snapshot in the evolution of the OSSD language. The continuous use of these artifacts even in the same OS environment might require a continuous adaptation to the ever changing attributes of the

domain analysis. Similarly, the RCNL tool can be applied to traditional requirements documents only after a customization designed to capture the specifics of that domain is accomplished. Most closed source documents have clearly delineated requirements, with few classifications. The RCNL classifier can be used to identify incomplete or incorrect requirements specifications, extend existing requirements classifications, or provide new classifications where they do not exist.

Future work has two main directions. First, I will continually refine the parsing rules to improve the quality of the recognition and classification. This will be achieved largely through detailed analysis of partially correct and missing tags in the analyzed dataset. Another improvement direction is represented by capturing part of the context surrounding a grammar-based requirement. This will be achieved through the implementation of reference resolution techniques (Mitkov 1998; Li et al. 2004; Mala and Uma 2006). Other two areas of improvement are the automation of data collection process through the use of data integration platforms such as KNIME, and the enhancing of the NL analysis through the use of machine learning techniques. Second, I will extend the data to include structured text. Feature requests, bug reports, and other tracked work items have a variety of structured attributes including: author, data, version, references (links), etc. I believe such structured data can be used to increase the recognition and classification quality. With access to the structured data, I plan to extend the work to analyze traceability relationships, such as contributions, evolution, and the interrelation between requirements and code. Fourth, I will customize the artifacts proposed by this dissertation and apply them to analyses in new domains, such as state-level IT governance policies, and evolution of technological innovation in social media.

7. Appendix

Illustrative classifications of micro-requirements discovered.

Micro-requirement	Classification
Best way would be to configure a program via KeePass options and only link that program within the password entry.	C8 – Access control C9 – Access audit C17 - Expandability
… when a new version of Firefox would be installed in a different directory, I only had to change the path once	C13 – Simplicity C17 - Expandability
I would like to be able to manage icons used for KeePass entries, maybe even import icons from system dlls.	C1 – Traceability C2 – Completeness C6 – Execution efficiency C7 – Storage efficiency C9 – Access audit C10 – Operability C19 - Modularity
Perhaps the ability to manage icons will be included in a future release	C2 – Completeness C6 – Execution efficiency C7 – Storage efficiency
Plus I'd like to have a USB memory stick that could open the KDB file too.	C7 – Storage efficiency C12 - Communicativeness
… the first match is automatically selected	C3 – Consistency C4 – Accuracy C6 – Execution efficiency C10 – Operability C13 – Simplicity
… a plot that is being created using pdf backend it would be great if the url argument actually created a clickable html link.	C1 – Traceability C10 – Operability
I tried to compile the package with python 2…	C15 - Instrumentation
It should either be documented and [namespace export]ed, or should be changed to look like a private command.	C8 – Access control C11 – Training C12 – Communicativeness C17 - Expandability
Please make it possible to search in the notes field and jump to the first matching entry (using ctrl-f).	C1 – Traceability C3 – Consistency C6 – Execution efficiency C9 – Access audit C10 – Operability C16 – Self-descriptiveness
The most common approach is when there is a single-sign-on system on the back end but multiple entry points.	C2 – Completeness C3 – Consistency C6 – Execution efficiency C9 – Access audit C18 – Generality C19 - Modularity

Grammar-based strategy – false positives and non-requirements.

False positives	Non-requirements
…a user wants…	Good God!
This can only have usability advantages.	…just extra work …
…possible options should be…	Regards[,] mb277
…asked somewhere already, but I cannot see a feature request.	Like this exemple:
Therefore this is a low-prio bug but bothering in practice.	…can squeeze this in.

McCall's 23 software quality criteria.

No.	Criteria	No.	Criteria
1	Traceability	13	Simplicity
2	Completeness	14	Conciseness
3	Consistency	15	Instrumentation
4	Accuracy	16	Self-descriptiveness
5	Error tolerance	17	Expandability
6	Execution efficiency	18	Generality
7	Storage efficiency	19	Modularity
8	Access control	20	Software-system independence
9	Access audit	21	Machine-independence
10	Operability	22	Communication commonality
11	Training	23	Data commonality
12	Communicativeness		

8. References

Ambriola, V. and V. Gervasi (2006). "On the Systematic Analysis of Natural Language Requirements with CIRCE." Automated Software Engineering **13**: 107-167.

Anderson, S. and M. Felici (2002). Quantitative Aspects of Requirements Evolution. Proceedings of the 26th Annual International Computer Software Conference, COMPSAC 2002, Oxford, England, IEEE Computer Society Press.

Auer, P. and S. Pfänder (2011). Constructions: Emerging and Emergent, Walter de Gruyter.

Bass, L., P. Clements and R. Kazman (1998). Software Architecture in Practice. Reading, MA, Addison Wesley.

Bejan, A. and S. Lorente (2011). "The constructal law origin of the logistics S curve." Journal of Applied Physics **110**.

Berkhout, A. J. and P. A. van der Duin (2007). "New ways of innovation: an application of the cyclic innovation model to the mobile telecom industry." International Journal of Technology Management **40**(4): 294-309.

Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. J. MaCleod and M. J. Merritt (1978). Characteristics of Software Quality. New York, North-Holland.

Broy, M. and T. Stauner (1999). "Requirements Engineering for Embedded Systems." Informationstechnik und Technische Informatik **41**(2): 7-11.

Cao, J., J. M. Crews, M. Lin, A. Deokar, J. K. Burgoon and J. F. Nunamaker Jr (2006). "Interactions Between System Evaluation And Theory Testing: A Demonstration of the Power of a Mulitfaceted Approach to Systems Research." Journal of Management Information Systems **22**(4): 207-235.

Chomsky, N. (1980). On Cognitive Structures and Their Development. Language and Learning: The Debate Between Jean Piaget and Noam Chomsky. M. Piatelli Palmarini. London, Routledge and Kegan Paul.

Chomsky, N. (1986). Knowledge of Language: Its Nature, Origin and Use New York, Prager.

Chung, L., B. A. Nixon, E. Yu and J. Mylopoulos (1999). Non-functional Requirements in Software Engineering. Boston, Springer.

Cleland-Huang, J., R. Settimi, X. Zou and P. Solc (2006). The Detection and Classification of Non-Functional Requirements with Application to Early Aspects. Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), IEEE Computer Society.

Cockburn, A. (1997). "Using Goal-based Use Cases." Journal of Object–Oriented Programming **10**: 56-62.

Crowston, K., H. Annabi and J. Howison (2003). Defining Open Source Software Project Success. Proceedings of the 24th International Conference on Information Systems.

Crowston, K., J. Howison and H. Annabi (2006). "Information Systems Success in Free and Open Source Software Development: Theory and Measures." Software Process: Improvement and Practice (Special Issue on Free/Open Source Software Processes.) **11**(2): 123-148.

Cunningham, H., D. Maynard, K. Bontcheva and V. Tablan (2002). GATE: An Architecture for Development of Robust Hlt Applications. Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL'02), Philadelphia, Association for Computational Linguistics.

Davis, F. D. (1989). "Perceived usefulness, perceived ease of use, and user acceptance of information technology." MIS Quarterly **13**(3): 319-339.

DeLone, W. H. and E. R. McLean (1992). "Information Systems Success: The Quest for the Dependent Variable." Information Systems Research **3**(1).

DeLone, W. H. and E. R. McLean (2003). "The DeLone and McLean Model of Information Success: A Ten-Year Update." Journal of Management Information Systems **19**(4): 9-30.

Denger, C., D. Berry and E. Kamsties (2003). Higher Quality Requirements Specifications through Natural Language Patterns. Proceedings of the IEEE International Conference on Software: Science, Technology & Engineering (SwSTE'03), IEEE Computer Society.

Dvir, D. (2003). "An empirical analysis of the relationship between project planning and project success." International Journal of Project Management **21**(2): 89-95.

Dvir, D. (2005). "Transferring projects to their final users: The effect of planning and preparations for commissioning on project success." International Journal of Project Management **23**(4): 257-265.

Edwards, M. L., M. Flanzer, M. Terry and J. Janda (1995). RECAP: a requirements elicitation, capture and analysis process prototype tool for large complex systems. Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95), Fort Lauderdale, Florida, IEEE Computer Society.

Elazhary, H. H. (2010). "REAS: An Interactive Semi-Automated System for Software Requirements Elicitation Assistance." International Journal of Engineering Science and Technology **2**(5): 957-961.

Fantechi, A. and E. Spinicci (2005). A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents. Proceedings of the Workshop em Engenharia de Requisitos [Requirements Engineering Workshop] (WER2005), Porto, Portugal.

Feblowitz, M., S. Greenspan, H. Reubenstein and R. Walford (1996). ACME/PRIME: requirements acquisition for process-driven systems. Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD '96), Schloss Velen, Germany, IEEE Computer Society.

Fitzgerald, B. (2006). "The Transformation of Open Source Software." MIS Quarterly **30**(3): 587-598.

Frakes, W. B. and R. S. Baeza-Yates (1992). Information Retrieval: Data Structures and Algorithms. Eaglewood Cliffs, New Jersey, PTR Prentice-Hall, Inc.

Gamma, E. and K. Beck (2004). Contributing to Eclipse: principles, patterns, and plug-ins, Addison-Wesley Professional.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). Design patterns : Elements of reusable object - oriented software. Reading, MA, Addison-Wesley.

Gao, Y., M. V. Antwerp, S. Christley and G. Madey (2007). A Research Collaboratory for Open Source Software Research. Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07), Minneapolis, MN, IEEE Computer Society.

Goldin, L. and D. M. Berry (1994). AbstFinder, a prototype abstraction finder for natural language text for use in requirements elicitation: design, methodology, and evaluation. Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, CO, IEEE Computer Society.

Gotel, O. and A. Finkelstein (1997). Extended requirements traceability: results of an industrial case study. Proceedings of the Requirements Engineering, 1997., Third IEEE International Symposium on.

Gotel, O. C. Z. and C. W. Finkelstein (1994). An analysis of the requirements traceability problem. Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, CO, IEEE Computer Society.

Grady, R. (1992). Practical Software Metrics for Project Management and Process Improvement. Englewood Cliffs, NJ, Prentice Hall.

Hayes, J. H., A. Dekhtyar and S. K. Sundaram (2006). "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods." IEEE Transactions on Software Engineering **32**(1): 4-19.

Hevner, A. R., S. T. March, J. Park and S. Ram (2004). "Design Science in Information Systems Research." MIS Quarterly **28**(1): 75-105.

Hippel, E. v. and G. v. Krogh (2003). "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science." Organization Science **14**(2): 209-223.

Hooks, I. F. and K. A. Farry (2001). Customer-Centered Products: Creating Successful Products Through Smart Requirements Management, Amacom.

Hopper, P. J. (1987). "Emergent Grammar." Berkeley Linguistics Society **13**: 139-157.

Hopper, P. J. (1992). Discourse: emergence of grammar. International Encyclopedia of Linguistics. Oxford University Press, Oxford. W. Bright**:** 364-367.

Hopper, P. J. (1998). Emergent grammar. The new psychology of language: Cognitive and functional approaches to language structure. M. Tomasello. **1:** 155-176.

Hopper, P. J. and E. C. Traugott (2003). Grammaticalization, Cambridge Univ Pr.

Hull, E., K. Jackson and J. Dick (2005). Requirements Engineering. London, Springer.

IEEE (1998). IEEE Standard for a Software Quality Metrics Methodology, IEEE. **IEEE Std 1061-1998**.

ISO (2001). Software Engineering - Product Quality, ISO. **ISO/IEC 9126**.

ISO (2011). Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models, ISO. **ISO/IEC 25010**.

Jantsch, E. (1967). Technological forecasting in perspective: A framework for technological forecasting. Organization for Economic Co-Operation and Development. Paris.

Jarke, M. and K. Paulk (1994). "Requirements Engineering in 2001: (virtually) managing a changing reality." Software Engineering Journal: 257-266.

Jensen, C. and W. Scacchi (2004). Data mining for software process discovery in open source software development communities. Proceedings of the First International Workshop on Mining Software Repositories, Edinburgh, Scotland, IEEE Computer Society.

Kasser, J. E. (2004). "The First Requirements Elucidator Demonstration (FRED) tool." Systems Engineering **7**(7): 243-256.

Kof, L. (2005). "An Application of Natural Language Processing to Domain Modelling: Two Case Studies." International Journal of Computer Systems Science & Engineering **20**(1): 37-52.

Kof, L. (2007). Scenarios: Identifying Missing Objects and Actions by Means of Computational Linguistics. Proceedings of the 15th IEEE Requirements Engineering Conference.

Konrad, S. and B. H. C. Cheng (2002). Requirements Patterns for Embedded Systems. Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02), IEEE Computer Society.

Krogh, G. v. and E. v. Hippel (2006). "The promise of research on open source software." Management Science **52**(7).

Kujala, S., M. Kauppinen, L. Lehtola and T. Kojo (2005). The role of user involvement in requirements quality and project success. Proceedings of the 13th IEEE International Conference on Requirements Engineering, Helsinki University of Technology, Finland, Software Business & Engineering Institute.

Leifer, R., S. Lee and j. Durgee (1994). "Deep Structures: Real Information Requirements Determination." Information and Management **27**(5): 1-27.

Li, K., R. Dewar and R. Pooley (2004). Requirements capture in natural language problem statements, Heriot-Watt University.

Lintula, H., T. Koponen and V. Hotti (2006). Exploring the Maintenance Process through the Defect Management in the Open Source Projects - Four Case Studies. Proceedings of the International Conference on Software Engineering Advances (ICSEA'06), Como, Italy, IEEE Computer Society.

Lutz, R. R. and I. C. Mikulski (2003). "Operational Anomalies as a Cause of Safety-Critical Requirements Evolution." The Journal of Systems and Software **65**(2): 155-161.

Lyytinen, K. and G. M. Rose (2003). "The disruptive nature of information technology innovations: the case of internet computing in systems development organizations." MIS Quarterly **27**(4): 557-595.

Madey, G. The SourceForge Research Data Archive (SRDA), University of Notre Dame.

Mala, G. and G. Uma (2006). Automatic Construction of Object Oriented Design Models (UML Diagrams) from Natural Language Requirements Specification. Proceedings of the Pacific Rim International Conference on Artificial Intelligence (PRICAI) 2006: Trends in Artificial Intelligence, Springer Berlin / Heidelberg.

Manning, C. and H. Schütze (1999). Foundations of Statistical Natural Language Processing. Cambridge, MA, MIT Press.

Mansfield, E. (1983). "Long waves and technological innovation." The American Economic Review **73**(2): 141-145.

March, S. T. and G. F. Smith (1995). "Design and natural science research on information technology." Decision Support Systems **15**(4): 251-266.

McCall, J. A., P. K. Richards and G. F. Walters (1977). Factors in Software Quality. New York, Rome Air Development Center, Air Force Systems Command.

Mingers, J. (2003). "The paucity of multimethod research: a review of the information systems literature." Information Systems Journal **13**(3): 233-249.

Mitkov, R. (1998). Robust pronoun resolution with limited knowledge. Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL '98), Association for Computational Linguistics.

Mockus, A., R. T. Fielding and J. D. Herbsleb (2002). "Two Case Studies of Open Source Software Development: Apache and Mozilla." ACM Transactions on Software Engineering and Methodology **11**(3): 309-346.

Moreira, A., J. Araújo and I. Brito (2002). Crosscutting Quality Attributes for Requirements Engineering. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering Conference (SEKE '02), Ischia, Italy, Association for Computing Machinery (ACM).

Mylopoulos, J., L. Chung and E. Yu (1999). "From Object-Oriented to Goal-Oriented Requirements Analysis." Communications of the ACM **42**(1): 31-37.

Noll, J. (2008). Requirements Acquisition in Open Source Development: Firefox 2.0. Open Source Development, Communities and Quality, IFIP International Federation for Information Processing**:** 69-79.

Pinto, J. K. and D. P. Slevin (1987). "Critical factors in successful project implementation." IEEE Transactions Engineering Management **EM-34**(1): 22–27.

Ramesh, B. (1998). "Factors Influencing Requirements Traceability Practice." Communications of the ACM **41**(12): 37-44.

Ramesh, B. and M. Jarke (2001). "Toward reference models for requirements traceability." IEEE Transactions on Software Engineering **27**(1): 58-93.

Ramesh, B., C. Stubbs, T. Powers and M. Edwards (1997). "Requirements traceability: Theory and practice." Annals of Software Engineering **3**(1): 397-415.

Rijsbergen, C. J. V. (1979). Information Retrieval, Wiley Subscription Services, Inc., A Wiley Company.

Rumbaugh, J. (1994). "Getting started: Using use cases to capture requirements." Journal of Object–Oriented Programming **7**: 8-8.

Ryan, K. (1993). The Role of Natural Language in Requirements Engineering. Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, IEEE Computer Society.

Sampaio, A., N. Loughran, A. Rashid and P. Rayson (2005). Mining Aspects in Requirements. Early Aspects 2005: Aspect-Oriented Requirements Engineering and Architecture Design Workshop. Chicago, Illinois.

Scacchi, W. (2002). "Understanding the Requirements for Developing Open Source Software Systems." IEEE Proceedings - Software **149**(1): 24-39.

Scacchi, W. (2006). Understanding Free/Open Source Software Evolution. Software Evolution and Feedback: Theory and Practice. J. F. R. a. D. P. e. N.H. Madhavji. New York, John Wiley and Sons, Inc.**:** 181-206.

Scacchi, W. (2009). Understanding Requirements for Open Source Software. Design Requirements Engineering – A Multi-disciplinary perspective for the next decade. K. Lyytinen, P. Loucopoulos, J. Mylopoulos and W. Robinson. Berlin, Springer-Verlag**:** 467-494.

Scacchi, W. and T. Alspaugh (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense. The 5th Annual Acquisition Research Symposium.

Scacchi, W., K. Crowston, G. Madey and M. Squire (2009). Envisioning National and International Research on the Multidisciplinary Empirical Science of Free/Open Source Software.

Scada, J. (2004). Cartesian Metaphysics: The Scholastic Origins of Modern Philosophy. Cambridge, Cambridge University Press.

Shenhar, A. J., D. Dvir and O. Levy (1997). "Mapping the dimensions of project success." Project Management Journal **28**(2): 5-13.

Sommerville, I. and P. Sawyer (1997). Requirements Engineering: A Good Practice Guide, Wiley.

Stamelos, I., L. Angelis, A. Oikonomou and G. L. Bleris (2002). "Code Quality Analysis in Open Source Software Development." Information Systems Journal **12**(1): 43-60.

Toro, A. D., B. B. Jimenez, M. T. Bonilla, R. Corchuelo, A. R. Cortés and J. Pérez (1999a). Expressing Customer Requirements Using Natural Language Requirements Templates and Patterns. Proceedings of the 3rd IMACS/IEEE International Multiconference on: Circuits, Systems, Communications and Computers (CSCC'99), Athens, IEEE Computer Society.

Toro, A. D., B. B. Jimenez, A. R. Cortés and M. T. Bonilla (1999b). A Requirements Elicitation Approach Based in Templates and Patterns. Proceedings of the Workshop de Engenharia de Requisitos [Requirements Engineering Workshop] (WER 1999).

Truex, D. and R. Baskerville (1998). "Deep Structure or Emergence Theory: Contrasting Theoretical Foundations for Information Systems Development." Information Systems Journal **8**: 99-118.

Truex, D., R. Baskerville and H. Klein (1999). "Growing Systems in Emergent Organizations." Communications of the ACM **42**(8): 117-123.

Truex, D., R. Baskerville and J. Travis (2000). "Amethodical Systems Development: The Deferred Meaning of Systems Development Methods." Accounting, Management and Information Technologies(10): 53-79.

van Lamsweerde, A. (2000). Requirements Engineering in the Year 00: A Research Perspective. Proceedings of the 2000 International Conference on Software Engineering, ICSE 2000, Limerick, Ireland.

van Lamsweerde, A. (2007). Goal-Orientation in Requirements Engineering. Requirements Engineering - From System Goals to UML Models to Software Specifications, Wiley**:** 259-286.

Venkatesh, V., M. G. Morris, G. B. Davis and F. D. Davis (2003). "User acceptance of information technology: Toward a unified view." MIS Quarterly **27**(3): 425-478.

Vlas, R. E. and W. Robinson (2012). "A Pattern-Based Method for Requirements Discovery and Classification in Open-Source Software Development Projects." Journal of Management Information Systems **28**(4): 11-38.

Vlas, R. E. and W. N. Robinson (2011). A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects Proceedings of the Hawaii International Conference on Software Systems (HICSS-44), HI, USA, IEEE.

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

Weber, M. and J. Weisbrod (2003). "Requirements Engineering in Automotive Development: Experiences and Challenges." IEEE Software **20**(1): 16-24.

Wiegers, K. E. (2003). Software Engineering. Redmont, Washington, Microsoft Press.

Wimalasuriya, D. C. and D. Dou (2010). Components for Information Extraction: Ontology-based Information Extractors and Generic Platforms. Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM '10), Association for Computing Machinery (ACM).