A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects

Conference Paper · January 2011

DOI: 10.1109/HICSS.2011.28 · Source: DBLP

CITATIONS
26

READS
162

2 authors:

Radu Vlas
University of Houston - Clear Lake
16 PUBLICATIONS **76** CITATIONS

SEE PROFILE

William N. Robinson
Georgia State University
71 PUBLICATIONS **1,947** CITATIONS

SEE PROFILE

A Rule-Based Natural Language Technique for Requirements Discovery and Classification in Open-Source Software Development Projects

Radu Vlas
Computer Information Systems Department
Georgia State University
rvlas@cis.gsu.edu

William N. Robinson
Computer Information Systems Department
Georgia State University
wrobinson@gsu.edu

Abstract

Open source projects do have requirements; they are, however, mostly informal, text descriptions found in requests, forums, and other correspondence. Understanding of such requirements can provide insight into the nature of open source projects. Unfortunately, manual analysis of natural language (NL) requirements is time-consuming, and for large projects, error-prone. Automated analysis of NL requirements, even partial, will be of great benefit. Towards that end, we describe the design and validation of an automated NL requirements classifier for open source projects. Initial results suggest that it can reduce the effort required to analyze requirements of open source projects.

1 Introduction

The increasing development and use of open source software has gained the interest of researchers who are beginning to explore qualities of OSS development[1, 2]. Analysis of OSS management, membership, processes, and products may lead to improvements in all software development. Studies of OSS requirements provide a means for systematically analyzing questions of OSSD. Thus, discovery and analysis of OSS requirements can lead to improvements in software development.

Many OSS projects are successful, according to studies that link success and software quality factors [3, 4]. In OSSD, the software product is developed, distributed, and supported by users. Common characteristics also include (1) large numbers of developers, (2) volunteering rather than delegating, (3) limited or no emphasis on design activities, (4) few constraints such as project plan, list of deliverables, or timelines [1].

At a glance, it may appear that the requirements analysis stage is absent, given that requirements are generally understood by the developers, who are also potential users of the product [5]. However, Scacchi has identified *requirements informalisms*, which are "the information resources and artifacts that participants use to describe, proscribe, or prescribe what's happening in a OSSD project"[6]. Scacchi identifies two dozen types, which include chats, email, forums, project digests, *etc*. By analyzing these unstructured, informal, natural language artifacts, one can better understand the requirements, and thus the development of OSS.

Our automated requirements classifier is aimed to help 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. Of course, first we must provide a good OSS classifier. That is the subject of this research project.

1.1 Discovering requirements

As Scacchi's study shows, requirements take many forms in OSSD projects, most of which are represented as NL text[6]. For each form, there are many requirements. For example, our selected 16 projects from SourceForge average 146,716 words just in feature requests (§5). Cleland-Huang *et. al.* have found that forums are filled with thousands of requirements[7]. Unfortunately, the NL texts contain much that are not requirements—for example, social communications, code segments, slang, typos, *etc.* [6, 7]. Thus, requirements discovery is first about delimiting each requirement within its source. Once requirements are identified, then subsequent processing can begin.

1530-1605/11 $26.00 © 2011 IEEE

Requirements engineering theory specifies measures that can guide the analysis of development. Requirements classification and tracing are common starting points. Classification provides an overview of the kinds of requirements present. Reliability, efficiency, integrity, and usability are common categories. Quality models, such as McCall[8], Boehm[9], IEEE[10], and ISO, specify qualities and their characteristics, which can be used to specify or recognize requirements.

1.2 Classifying NL requirements

This research aims to provide an automated technique for the discovery and classification of requirements in NL texts of OSSD projects. To that end, we have developed software, a Requirements Classifier for Natural Language (RCNL). In this study, we limit the text sources to the text found in work items, such as SourceForge's issue tracker (similar to Bugzilla or Jira) [11]. Herein, we describe the design (§3) and engineering (§4) of the tool, and experiments measuring its capabilities (§5). These results suggest that the tool may be useful in classifying requirements in OSSD projects.

2 Related research

2.1 Requirements in OSSD

Jensen and Scacchi's study on OSSD requirements show that informal communication, rather than classical formal modeling, is common [12]. Requirements emerge through a dynamic, social process. The informalism concept is strengthened by the acknowledging of the importance of discussion forums as a means of reaching common understanding and acceptance of requirements [11].

In a study of non-functional requirements (NFR), Cleland-Huang *et. al.* uses a semi-automated technique for identification and classification of requirements from both structured and unstructured documents[13]. 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 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. This semi-automated method requires a researcher's intervention and control throughout the entire process.

Two other studies suggest the use of Natural Language Processing (NLP) tools for achieving requirements identification and classification or analyzing existing requirements documents. Sampaio *et. al.* presents a NLP technique based on WMATRIX for analyzing requirements documents with the intent of identifying aspects specific to Aspect Oriented Software Development (AOSD). The 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 [14].

Fantechi and Spinicci propose a semi-automated process for improving requirements quality through reduction of inconsistencies. Their process also uses NLP; Phrasys is used for phrase and sentence extraction. The proposed technique processes structured requirements documents into Subject-Action-Object (SAO) triples in order to analyze interactions between entities[15].

2.2 Classification of requirements

Requirements have been traditionally classified as either functional (FR) or non-functional (NFR), even though some researchers consider this classification to be too broad [16]. While adopting this perspective, researchers refer to FR as goals (or hard goals, or behavioral requirements), and to NFR as soft goals [17, 18]. FR 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) [19, 20]. Some variations to it include listing of security concerns under FR, adding supportability under NFR, or specifying sub-categories of these two [21].

Additional requirements classifications start from an agent-based perspective informing the V-model of requirements and list them as user-stakeholder, system, sub-system, or component requirements [22-24]. 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 [18].

2.3 Requirements patterns

Duran *et al.* proposes a series of requirements templates that can help capture requirements [25-28]. They introduce linguistic patterns (L-patterns, natural language commonly used for describing requirements) and requirements patterns (R-patterns, generic requirements templates) [29]. While attempting to bridge between natural language and formal requirements specification, this study reaches a middle ground between them. Re-specification, rather than requirements discovery, is the focus of the technique.

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 [30]. Also for embedded systems, Denger addresses the problem of requirements imprecision [31]. This study uses a pattern-based analysis of existing requirements in order to identify missing information and to "fix" these 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.

3 Classifier design

We follow the design science approach [32] to develop the RCNL classifier. It embodies our theory of a pattern-based approach to discovering requirements from real natural-language. A key element of RCNL is its multi-level ontology, in which the lower levels are grammar-based while the upper levels are requirements-based. The RCNL ontology is realized in an implementation that uses a multi- level GATE parser. We apply the Hevner *et. al.* [32] *descriptive* approach to design science—developing and then evaluating the design of RCNL using scenarios and argumentation. Additionally, we compare the results of automated classification by RCNL with that by an expert.

The OSSD project data are a central problem to 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 Standard English grammar. In fact, nearly all the texts are fragments containing many typos, misspellings, and idioms (e.g., text smileys).

To address the data sources, our *classifier* is a kind of weak ontology-based information extraction (OBIE) system. Such systems work well for parsing and tagging fragments of unstructured natural language text[33]. The ontology is mostly grammar-concepts, except for the last level that contains the requirements classification concepts.

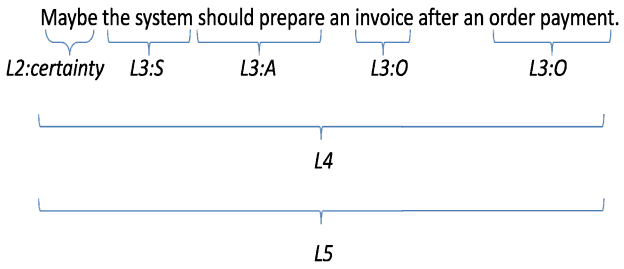

Figure 1 Text tagged with the requirements ontology.

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

3.1 Illustrative text tagging

Figure 1 illustrates the tagging of a sentence using the RCNL requirements parsing ontology. For illustrative purposes, the text is well-structured and simple. (Section 4 illustrates more representative OSSD project text.) The tags from first two levels are not shown, because they are mostly common parts of speech tags.

Figure 1 shows that a qualifier (L2) begins the requirement (L4), which is comprised of subject, action, and object tags from L3. The entire statement is tagged as a classified requirement (L5).

3.2 Requirements parsing ontology

The RCNL ontology for OSSD projects includes six levels, as summarized in Table 1. The first two levels contain common natural language grammar concepts. The next three levels contain concepts of logical statements. 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 3 characterizes subject-action-object triples using information from levels 0 and 1, but it does not depend on level 2.

Table 1 The RCNL requirements parsing ontology.

RCM Level	Description	Elements covered
L0: Token	Defines basic elements of text commonly included in all types of communication.	Word, punctuation, symbol, list, filename, url, email address, separator/delimiter, sentence, phrase
L1: POS	Defines most common Parts-of-Speech (POS) elements.	Adjective, adverb, conjunction, preposition, determiner, negation, noun, verb
L2: Qualification	Identifies expressions pointing to a context which might indicate the presence of a requirement.	Belief, certainty, necessity, preference, qualifier, quantifier, qualifying phrase
L3: Entities	Identifies the three fundamental elements of a requirement.	Subject(S)/actor, action(A)/verb, object (O)
L4: Requirement	Discovers parts of text identified as 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, URL, and file reference. Level 1 (L1) concepts are the common English parts of speech (POS).

Level 2 (L2) concepts are qualifiers, such as belief, preference, necessity, quantity, *etc*. These mostly depend on specific words identified in level 0, but may also depend on the POS of level 1 (e.g., determiners, deitics, quantifiers, numerals; modals, tense; negators).

Consider the following text from our dataset.

```
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, or quantifiers that modify a requirement are tagged at L2. Such phrases usually introduce a requirement.

Level 3 (L3) concepts are simply subject, action, and object. Subject is not simply a noun, but an actor (person, an object, or a 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 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[15]. Our level three patterns discover all subjects, actions, and objects present in text that can potentially be part of a requirement.

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

In the L4 patterns, the existence of a subject and object in text is optional. For example, the following text (from our 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 requirement. This example (from our dataset) illustrates.

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

This phrase "I want" (L2) qualifies the requirement (L4) that it introduces.

Finally, level 5 (L5) concepts are the domain specific classification of the level 4 statements. We have designed two L5 classifiers: (1) standard McCall's quality model, and (2) extended McCall's for OSSD projects.

McCall's model specifies 23 quality criteria for software. These concepts are represented in our L5. In particular, we specify rules for recognizing the 23 quality criteria using the annotations of levels 0 – 4. We aimed to accurately capture the quality model as specified by McCall.

In addition to McCall's classification rules, we specified our own classification rules for the McCall's 23 quality criteria. In particular, these extend the McCall's classification rules to recognize concepts and terms unmentioned in the McCall specification. We call this 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[13]. Second, analysis of NL associated with OSSD projects suggests further keywords and parsing strategies. In particular, we iteratively extended the RCNL parser, and tested it on sample data, until we reached a consistent level of correct classification. Additionally, we made use

of the SensAgent online dictionary for gathering all synonyms who are properly describing the same meaning as the original keyword (www.sensagent.com).

4 Classifier engineering

The RCNL classifier is implemented in GATE [34]. The General Architecture for Text Engineering is developed by the Sheffield Natural Language Processing Group at the University of Sheffield and includes a large community of collaborators and users. Next, we describe a bit of the engineering involved in realizing the RCNL framework in GATE. In particular, we describe rules for tagging text according to the ontology, additional text processing, and the overall text processing activity.

Our parser implements the RCNL ontology to recognize and classify NL requirements. For each level, JAPE rules specify how GATE tags text with concepts of that level. The rules are processed in a pipeline, from level 0 to level 5. The final output tags qualified (L2) requirements (L4) according to their classification (L5). Any text may have multiple tags 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.

4.1 Rule-based tagging

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

GATE provides JAPE (Java Annotation Pattern Engine), a rule-based text-engineering engine that supports Java and regular expressions. GATE also provides an annotation indexing and search engine with an advanced graphical user interface called ANNIC (Annotations in Context). Our analyses 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) describing the pattern to be matched and a right-hand side (RHS) defining the annotation and the features to be created for all the discovered instances of the pattern.

The current implementation of the RCNL classifier consists of about 205 JAPE rules. To illustrate how the RCNL ontology is recognized through JAPE rules, we present simple rules from levels 3 and 5.

4.1.1 A level 3 rule

To illustrate our 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"} |
      {Token.category == "Filename"} |
      {Token.category == "email"} |
      {Token.category == "url"} |
   // ...
      ({L1.category == "Determiner"}
{L1.category == "Noun"})
   ) [1,5]
)
:SubjectFound
-->
 :SubjectFound.L3 = {category =
"Subject"}
```

The LHS part of the rule describes a pattern searching for nouns (as defined in L1), or pronouns (as defined in pre-defined rules in GATE), or filenames, Url's, email, (*etc*. 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`.

4.1.2 A level 5 rule

Here is a (slightly simplified) L5 classification rule.

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

:L5_ComunicativenessFired.Comunicativenes
s = {category = "F5C12"}
```

The LHS matches text annotated as L4 (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 name for factor 5, criteria 12.

4.2 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 introduction phrase and each list item are complete requirements. Often, however, the introductory phrase can be classified as a requirement while the list items are examples or phrases that extend the meaning of the introductory phrase. To address such issues, the L5 tag associated with the introductory phrase is propagated to 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 processing occurs after level 5.

A second auxiliary text processing applies to check requirement containership. It is possible, but rare, that a (L5) requirement is fully contained inside another requirement. When found, a final finishing rule re-annotates requirements to indicate that only the larger 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 rules. These include:

- Tokens covered: Number of tokens in text covered by the text annotated as potential requirements.
- Sentences covered: Number of sentences in text covered by the text annotated as potential requirements.
- Requirements tagged
- Classifications created
- Requirements classified

All rules are controlled by a configuration. Our experiments, described next, enable or disable various rules to determine their contribution to the classifier's performance.

Figure 2 shows the RCNL classifier as implemented in GATE. The screen image shows text tagged as requirement, and two classifications, Operability and StorageEfficiency.

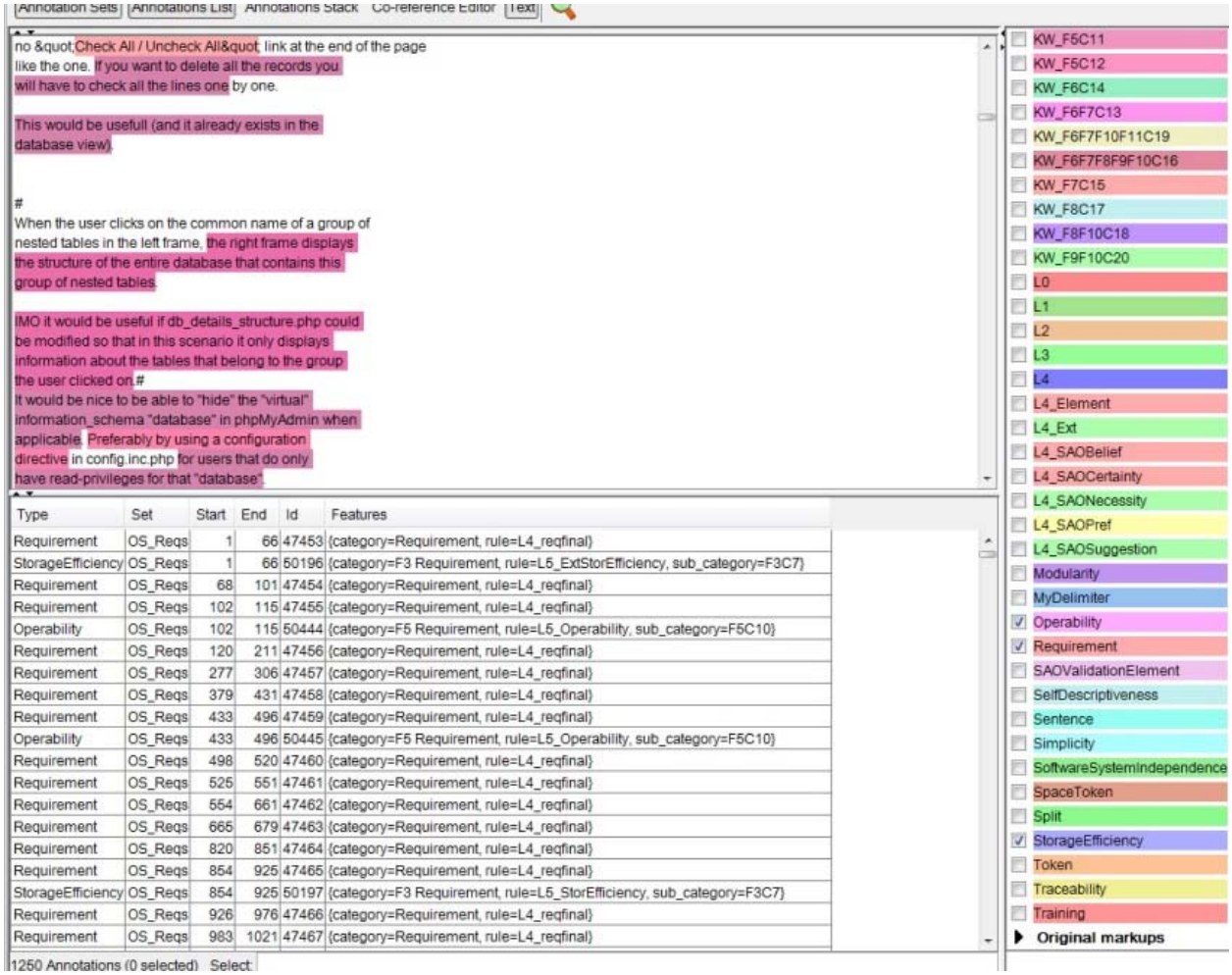

Figure 2 The RCNL classifier as implemented in GATE, showing requirement and two classification tags of text.

5 Classifier experiments

Having created a requirements classifier for unstructured and typo-rich NL text of OSSD projects, we applied two kinds of validation methods to assess it. First, we generated metrics from the classification of sampled OSSD project data. Second, we compared the automated classification by RCNL classifier with that by an expert. These two experiments are presented next.

5.1 SourceForge dataset

Like many researchers in OSSD, we selected SourceForge projects for our dataset[11]. SourceForge provides access to over 230,000 OSSD projects and over 2 million registered user's activities, as of February 2009. We 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) [35, 36]. In particular, we processed the May 2008 data from SourceForge.

We narrowed our dataset to substantial projects that actively used requirements. We define this as: (1) having more than three developers, (2) more than 1,000 downloads, and (3) more than 700 feature requests. The result is the 16 projects found in Table 3. The data collected is grouped in 16 text files (with sizes ranging from 229Kb to 2,304Kb), one for each project. For each project, the RCNL classifier analyzed all feature requests to discover requirements and classify those found.

5.2 Experiment configurations

Our analysis uses two configurations to process and evaluate each project from the dataset.

1. Requirements discovery and classification based on extensions to McCall's quality model (§3.2)
2. Requirements discovery and classification based only on McCall's quality model

Table 3 summarizes the results of configuration 1 as McCall+ and configuration 2 as McCall.

Our performance analysis of the RCNL classifier considered two processing steps:

1. Combined levels L0 – L4 averages 1261 tokens per second,
2. L5 averages 600 tokens per second.

This means that analysis of an average project is completed in about six minutes, for processing L0 – L5. Database retrieval of the features and storage of the results is not considered in this number.

Table 2 presents some feature requests from the dataset and their automate classification, according to the RCNL classifier.

Table 2 Example classifications from the dataset.

Feature Request	Classifications
Perhaps this could be made an option in the export screen.	Expandability Operability
It could even be conditionalized, so that the option to export a real Excel file is not available unless this PEAR module is installed.	Expandability Modularity Simplicity
Using this Excel export module should also fix the fact that Excel for Mac and Excel for Windows require differing CSV formats	ExecutionEfficiency Modularity Operability Simplicity StorageEfficiency Traceability

5.3 Data analysis

Table 3 shows the metrics for the automated classification results of the RCNL classifier. The columns provide values for the project name, the number of tokens in the project's feature requests, percent of tokens recognized as part of a requirement, and percent of requirements classified according to the extended McCall model (McCall+) and the standard McCall model.

Consider the Compiere project as an example. The analyzed text has 61,292 tokens. Of those token, 74.3% are recognized by RCNL as being part of a requirements statement. The other 25.7% may be code segments, social tags (e.g., smileys), or other text this is not recognized by RCNL. Of the statements that are recognized as requirements statement, RCNL classified 62.3% using the McCall+ model and 31.2% using the basic McCall model. Some requirements remain unclassified by either model. This occurs when the classifier cannot match the given requirements text with a classification rule.

Table 3 Classification metrics of selected projects.

Project	Tokens	Req's	Classification	
			McCall+	McCall
AWStats	99,549	61.7%	50.5%	22.3%
Compiere	61,292	74.3%	62.3%	31.2%
FileZilla	126,196	73.8%	58.3%	23.4%
Fire	51,164	70.5%	54.0%	22.1%
Float's	54,511	71.5%	55.5%	21.7%
Gallery	199,920	70.4%	56.4%	25.7%
KeePass	118,626	74.3%	62.5%	33.2%
MegaMek	112,121	69.8%	54.7%	26.9%
PCGen	347,167	63.9%	56.9%	28.3%
phpGedView	128,348	70.3%	56.3%	25.1%
phpMyAdmin	136,164	65.7%	61.4%	30.8%
POPFile	116,340	70.6%	53.2%	20.8%
SourceForge	508,894	66.2%	53.5%	24.0%
TikiWiki	109,401	68.6%	54.4%	24.9%
Tortoise	72,064	72.7%	55.9%	25.6%
WinMerge	105,695	72.8%	56.1%	23.2%
Average	**146,716**	**69.8%**	**56.4%**	**25.6%**

The average values are interesting. First, it's clear that our extensions to the standard McCall model did improve classification—56% is much better than 26%. Second, much of the text is recognized as being part of a requirement, with the average of 70%. (It should be noted that feature requests include all kinds of text, including lines of code and conversational text.) Finally, given the recognized requirements, most are classified, with an average of 56%.

5.4 Expert analysis

We compared the RCNL classifier's results with that of a human expert. We divided each project text into small data segments, a few paragraphs long. Next, we randomly selected 20 samples from those segments. The expert, an author of this paper, then classified the requirements within the text. Next, we compared the expert results with those produced by the RCNL classifier. Fortunately, GATE provided a plugin for just this purpose.

The results of the comparison are summarized in Table 4. The first column compares the RCNL requirements tagging of text with that of the expert. Automated tagging was correct for 23 requirements, but missed 7 requirements entirely. In most cases, the requirements tags overlapped, which is expected. A match is *correct* only if both tags begin and end at the exact same character location. *Partially correct* typically occurs when one of the tags is missing a word, whitespace, or punctuation. Given this context, we take a lenient perspective, and consider partially correct responses as correct.

Precision, recall, and F-measure indicate qualities of the classifier. Precision is the percent of classified tags that are correct. Recall is the percent of correct tags that are identified. Because greater precision decreases recall, it is useful to consider F-measure, which computes the harmonic-mean of precision and recall. In our context, recall is most important because we want to find the requirements, after which further processing may occur.

Table 4 presents the precision, recall, and F-measure for text tagging—column one is for requirements recognition, while column two is the aggregation of all McCall model classifications. Overall, the results are encouraging. For example, they are very similar to a study that limited classification to only NFRs[13].

Table 4 Expert comparison measures.

	Requirement	All annotation types
Precision	0.94	0.58
Recall	0.64	0.70
F-Measure	0.76	0.46

6 Discussion

This study contributes to research and practice of OSSD. A systematic method for discovery and classification of requirements in OSSD projects is currently not available. When available, such a method would enable important improvements, such as: (1) better understanding of requirements, their kinds and life-cycles, 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.

The RCNL classifier provides an alternative to existing methods that require substantial input from the researcher (§2). The RCNL classifier runs autonomously. However, it may require new rules to work effectively with new datasets. Although we did develop and test it on the large SourceForge dataset, it may be that other OSS data or traditional software artifacts require changes to the lower

levels of the parser. To adapt the RCNL classifier to another quality model (other than McCall), the level 5 rules must be modified.

Our future work has two main directions. First, we 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 on our datasets. Second, we 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*. We believe such structured data can be used to increase the recognition and classification quality. With access to the structured data, we plan to extend the work to analyze traceability relationships, such as contributions, evolution, and the interrelation between requirements and code.

7 Acknowledgements

We thank the National Science Foundation for their support via NSF grant 0613698.

8 References

[1] A. Mockus, *et al.*, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology,* vol. 11, pp. 309-346, July 2002 2002.

[2] E. v. Hippel and G. v. Krogh, "Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science," *Organization Science,* vol. 14, pp. 209-223, March-April 2003 2003.

[3] K. Crowston, *et al.*, "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.),* 2006.

[4] I. Stamelos, *et al.*, "Code Quality Analysis in Open Source Software Development," *Information Systems Journal,* vol. 12, pp. 43-60, February 2002 2002.

[5] B. Fitzgerald, "The Transformation of Open Source Software," *MIS Quarterly,* vol. 30, pp. 587-598, September 2006 2006.

[6] W. Scacchi, "Understanding Requirements for Open Source Software," in *Design Requirements Engineering – A Multi-disciplinary perspective for the next decade*, K. Lyytinen, *et al.*, Eds., ed: Springer-Verlag, 2009.

[7] J. Cleland-Huang, *et al.*, "Automated support for managing feature requests in open forums," *Communications of the ACM,* vol. 52, pp. 68-74, 2009.

[8] J. A. McCall, *et al.*, *Factors in Software Quality*: NTIS, 1977.

[9] B. W. Boehm, *et al.*, *Characteristics of Software Quality*. New York: North-Holland, 1978.

[10] "IEEE standard for a software quality metrics methodology," *IEEE Std 1061-1998,* 1998.

[11] H. Lintula, *et al.*, "Exploring the Maintenance Process through the Defect Management in the Open Source Projects - Four Case Studies," in *Proceedings of the International Conference on Software Engineering Advances (ICSEA'06)*, Como, Italy, 2006.

[12] W. Scacchi, "Understanding the Requirements for Developing Open Source Software Systems," *IEEE Proceedings - Software,* vol. 149, pp. 24-39, February 2002 2002.

[13] J. Cleland-Huang, *et al.*, "The Detection and Classification of Non-Functional Requirements with Application to Early Aspects," in *14th IEEE International Requirements Engineering Conference (RE'06)*, 2006, pp. 39-48.

[14] A. Sampaio, *et al.*, "Mining Aspects in Requirements," presented at the Early Aspects 2005: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, Chicago, Illinois, 2005.

[15] A. Fantechi and E. Spinicci, "A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents," in *Workshop em Engenharia de Requisitos (WER2005)*, Porto, Portugal, 2005, pp. 245-256.

[16] L. Bass, *et al.*, *Software Architecture in Practice*. Reading, MA: Addison Wesley, 1998.

[17] J. Mylopoulos, *et al.*, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM,* vol. 42, pp. 31-37, January 1999 1999.

[18] A. vanLamsweerde, "Goal-Orientation in Requirements Engineering," in *Requirements Engineering - From System Goals to UML Models to Software Specifications*, ed: Wiley, 2007.

[19] L. Chung, *et al.*, *Non-functional Requirements in Software Engineering*: Springer, 2000.

[20] A. Moreira, *et al.*, "Crosscutting Quality Attributes for Requirements Engineering," in *Proceedings of the Software Engineering and Knowledge Engineering Conference (SEKE)*, Ischia, Italy, 2002.

[21] R. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[22] M. Broy and T. Stauner, "Requirements Engineering for Embedded Systems," *Informationstechnik und Technische Informatik,* vol. 41, pp. 7-11, 1999.

[23] E. Hull, *et al.*, *Requirements Engineering*: Springer, 2005.

[24] M. Weber and J. Weisbrod, "Requirements Engineering in Automotive Development - Experiences and Challenges," in *Proceedings of the 2002 IEEE Joint International Conference on Requirements Engineering (RE'02)*, 2002.

[25] A. Cockburn, "Goals and Use Cases," *Journal of Object–Oriented Programming,* Sept.-Oct. 1997 1997.

[26] A. Durán, *et al.*, "Expressing Customer Requirements Using Natural Language Requirements Templates and Patterns," in *Proceedings of IMACS/IEEE CSCC'99*, Athens, 1999.

[27] E. Gamma, *et al.*, *Design patterns : Elements of reusable object - oriented software*. Reading, MA: Addison-Wesley, 1995.

[28] J. Rumbaugh, "Getting started: Using use cases to capture requirements," *Journal of Object–Oriented*

Programming, September 1994 1994.

[29] A. Durán*, et al.*, "A Requirements Elicitation Approach Based in Templates and Patterns," in *Workshop de Engenharia de Requisitos*, 1999.

[30] S. Konrad and B. H. C. Cheng, "Requirements Patterns for Embedded Systems," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*, 2002.

[31] C. Denger*, et al.*, "Higher Quality Requirements Specifications through Natural Language Patterns," in *Proceedings of the IEEE International Conference on Software—Science, Technology & Engineering (SwSTE'03)*, 2003.

[32] A. R. Hevner*, et al.*, "Design Science in Information Systems Research," *MIS Quarterly,* vol. 28, p. 75, Mar 2004.

[33] D. Wimalasuriya and D. Dou, "Ontology-Based Information Extraction: An Introduction and a Survey of Current Approaches," 2009.

[34] H. Cunningham*, et al.*, "GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications," in *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, Philadelphia, 2002.

[35] Y. Gao*, et al.*, "A Research Collaboratory for Open Source Software Research," in *Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007), International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS 2007)*, Minneapolis, MN, 2007.

[36] G. Madey. The SourceForge Research Data Archive (SRDA) [Online]. Available: http://zerlot.cse.nd.edu/