Applying a Rule-Based Natural Language Classifier to Open Source Requirements: a Demonstration of Theory Exploration

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 requirements are 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. Previously, we have demonstrated the Requirements Classifier for Natural Language (RCNL), which aids in NL requirements analysis. Herein, we demonstrate how the RCNL can aid in theory building. From its application to 16 open source projects, we 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. Although the theory is a conjecture, the process of its exploration demonstrates how RCNL can be used to explore theories about open source projects—theory exploration that would otherwise be intractable because of the difficulty in analyzing NL artifacts for requirements properties.

1 Introduction
Pat wants her career to move from programming to designer-analyst at her mid-sized information technology company. She has been encouraged to contribute to an open source project as a means to demonstrate her innovation and design abilities. She wants to select a project that is trending towards success, which will allow her contributions to be widely acknowledged. She wants to select a project that is not just starting or just ending, but is in the early or middle part of its lifecycle. What tools can she apply to help her select the right project?

Project managers face a similar question. Given a project in the early or middle part of its life cycle, should more resources be provided to contribute to the project’s success? Should a new tactic be applied to fix the project? Alternatively, should funding be cut because the project is trending toward failure?

One might be tempted to apply conventional analysis. 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 open source 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 2.3, 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 open source projects transition through three common steps to produce a new themed release: innovate, improve, and deploy[1, 2].

We are exploring techniques aimed to understand what open source developers are doing through analyses of their documents. We apply parsing techniques to understand documents in terms of pre-defined models, such as models of non-functional requirements (NFRs) or distributed collaboration. These analyses may help us to address questions, like Pat’s, about open source projects.

In the work presented herein, we 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. We 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 open source requirements
2. Classify requirements
3. Characterize trends of the classified into requirements factors
4. Correlate the requirements factors with project qualities that may relate to project success

Previously, we have developed tools for steps 1 - 2. This article demonstrates our approach to steps 3 – 4 through an analysis case study. This is exploratory work is designed to test the viability of the approach. In particular, how requirements discovery and classification fit within a qualitative methodology of theory exploration. Moreover, it is
difficult to assess its effectiveness, in part, because open source project success is poorly defined, as are the requirements factors associated with success. (In our study we use number of downloads as a proxy for project success; there are, however, other types of success measures that may be considered as will be described in Section 2.3.)

This article shows how open source projects can be analyzed according to two aggregated temporal requirements qualities to provide some indication of the project success. The two main requirements factors considered are:

1. Requirements development cohesion, which 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, which 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 open source project success, as we illustrate through analysis of 16 open source projects.

Our case-study demonstration of our four-step approach to NL open source analysis led us to 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 (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 projects, but remains a conjecture for more comprehensive analysis. It demonstrates the kinds of theories that we intend to explore using RCNL.

1.1 Documents Open Source Requirements

Requirements management may initially appear absent from open-source software development (OSSD). However, a review of the online communications reveals that OSSD includes many forms of informal, natural language requirements. In particular, 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” [3]. 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 OSSD.

Consider Figure 1, which presents feature requests from the Feature Tracker of the KeePass project on SourceForge. To understand OSSD, researchers need to characterize such communications. Questions arise like the following:

1. Is this a functional or non-functional requirement request?
2. Are there a few distinct requirements contained within the request, one requirement with sub-requirements, one requirement, or is the text social communication or other kinds of text?
3. What kind of requirement request is it? Can it be classified?
4. How is the requirement request modified over time?
5. How does the requirement request relate to other artifacts, such as the software code, test cases, and documentation?

Researchers also want to know how such requests affect quality and other project-level characteristics. For example, are projects more successful when they consider modularity or security requests early in the project?

The research presented here demonstrates a methodology and tool that provides a step towards
automated assistance in these open source requirements discovery problems.

1.2 Discovering and classifying requirements

OSSD requirements take many forms, most of which are represented as natural language (NL) text [3]. For each form, there are many requirements. For example, the KeePass project has 1,496 feature requests, 898 bug reports, and thousands of other various forum posts. Cleland-Huang et. al. found that forums are filled with thousands of requirements, as well as thousands of lines that are not requirements—for example, social communications, code segments, slang, typos, etc. [3, 4]. Thus, requirements discovery is first about delimiting each requirement within its source. Once requirements are identified, then subsequent processing can begin.

Requirements engineering theory specifies measures that can guide the analysis of software development, including classification and tracing. Classification provides an overview of the kinds of requirements present. Reliability, efficiency, integrity, and usability are common categories. Quality models, such as McCall [5], Boehm [6], IEEE [7], and ISO [8], specify qualities and their characteristics, which can be used to specify or recognize requirements. We chose McCall’s software quality model because of being widely accepted in both research and practice communities for very many years.

A quality model, like McCall’s, specifies words and phrases that are indicative of requirements belonging to a classification. For example, a requirement that includes the word faster or slow is indicative of a performance requirement. Using such keywords, the technique of keyword classification uses libraries of keywords, phrases, and grammar fragments to match against delimited text-requirements.

Figure 2 shows the result of applying the Requirements Classifier for Natural Language (RCNL). The highlighted text has been parsed as fragments, recognized as requirements, and classified according to an extended McCall requirements ontology (McCall+) [9]. 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 (number), as well as phrases such as “What is your pets name?” (Note the presence of typos resulting in incorrect grammar—one of the prominent challenges in analyzing NL data from Open Source forum posts).

1.3 Article Overview

Next, this article introduces reference theories that have guided the development of RCNL and its use in open-source project analysis. Section 3 presents the analysis case study demonstration. The final two sections present a discussion and conclusions from this study.

2 Reference Theory

2.1 Requirements in OSSD

Scacchi’s study on OSSD requirements show that informal communication, rather than classical formal modeling, is common [10]. 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 [12]. The NFR classification process proposed has three stages: mining phase, classification phase, and application phase. This
findings are limited to acknowledging the existence of requirements-based perspective [3, 10, 24]. Current associated development processes while maintaining a
A number of studies explored OSSD projects and their component requirements [21-23].

2.3 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 [15]. While adopting this perspective, researchers refer to FR as goals and to NFR as soft goals [16, 17]. 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) [18, 19]. Some variations to it include listing of security concerns under FR, adding supportability under NFR, or specifying sub-categories of these two [20]. Additional requirements classifications adopt an agent-based perspective informing the V-model of requirements and list them as user-stakeholder, system, sub-system, or component requirements [21-23].

2.3 Software Product Quality and Software Development Project Success
A number of studies explored OSSD projects and their associated development processes while maintaining a requirements-based perspective [3, 10, 24]. Current findings are limited to acknowledging the existence of requirements and their associated processes (elicitation, analysis, specification and modeling, validation, and communication) in open-source projects[3]. While requirements processes receive 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 its development lifecycle, receive slightly less attention. One important concept that is worth further investigation is represented by open-source requirements and their impact on project success.

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 [25, 26]. Wiegers’ and Dvir’s findings that software project success is determined by that project’s requirements directly support this perspective [27, 28].

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 [29]. Traditionally, projects were perceived as successful when they met time, budget, and performance goals. Obviously success is more than meeting budget and deadlines. TAM [30, 31] 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 [32]. 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 [33]. 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.

More recently, researchers have emphasized the importance of requirements as 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” [28]. 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 [34]. The relationship between requirements quality and project success is explored by Hooks and Farry [35]. 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 [27, 36]. 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.

2.4 Innovation in Software Development

In software projects, innovation is a prerequisite of success. Much IS research on innovation considers the open-source paradigm [37, 38], organization [39], or IT field as a whole. This leads to longitudinal effects, such as diffusion patterns (e.g., S-curve) [40]. More generally, technological innovation has been considered an exploratory process of integrating previously enumerated design elements [41]. Some have suggested that innovation can be modeled as decade long waves of innovation follow by lapses in innovation [42].

Here, we are more interested in how a small open-source team innovates. Thus, the team and the individual are 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 [41]. However, assuming the team has limited resources (i.e., members) they alternate between innovation and other project activities [43]. 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. In our theory development, 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 NFRs.

3 SourceForge Experiments

3.1 Methodology

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 NLP text found in the Feature Request forum of each project on SourceForge.

2. Classify requirements
   Use RCNL to classify requirements. RCNL uses an extended McCall’s model of 23 requirement qualities (aka non-functional requirements) [5].

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 the Introduction section) are computed. Finally, their trends of consecutive windows (e.g., \( \Delta RDC, \Delta RTF \)) are computed.

4. Correlate the requirements factors with project qualities that may relate to project success
   Finally, the derived factors are graphed, correlated, and otherwise compared as part of the exploration of relationships.

When we began the project, we had some initial hypotheses of constructs, such as RDC and RTF. Through the proceeding methodology, we derived the relationships that form our theoretical conjecture.

3.2 Data Collection and Analysis

Like many researchers in OSSD, we selected SourceForge projects for our dataset [11]. At the time of our data collection, March 2011, SourceForge provided access to over 324,000 OSSD projects and over 3.4 million registered user’s activities. 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) [44, 45]. In particular, we processed the February 2010 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 I. 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 [9, 46], which simplifies comparison and ensures validation of the requirements classification.

The analysis of OSSD projects lifecycle requires a time-based analysis of available data (aka 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. Naturally, a project’s duration is not a multiple of 6-months. Consequently, the length of the last data
window in each project is between 3 and 9 months. This is useful in the analysis of individual requirements discovered but has the potential to skew the results and bias their interpretation in a project-level evolution analysis. Consequently, we drop last 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.

3.3 Requirements development cohesion

Figure 3 shows a stack graph of NFR variance for 14 projects over 12 6-month windows, with Average as line (Top, scaled right).

Figure 3 shows that some projects show waves of $\sigma_{\text{NFR}}$, revealing cycles of innovation followed by consolidation.

Figure 4 shows a (solid) line graph of KeePass’s $\sigma_{\text{NFR}}$ for 12 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 NFR’s that are central to new product features.

The closing of feature requests marks the inclusion of new features in a product. In Figure 4, the Closed (dashed) line graph 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 $\sigma_{\text{NFR}}$ wave peaks. Theory suggests that, a successful innovation effort ($\sigma_{\text{NFR}}$ 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 $\sigma_{\text{NFR}}$ decreases.

These relationships between $\sigma_{\text{NFR}}$ and Closed seem to hold (roughly) in Figure 4. Checking for correlation between the $\sigma_{\text{NFR}}$ and Closed values using Pearson’s correlation coefficient gives us $\rho_{\sigma,\text{Closed}} = -0.42$, 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 4 also shows the number of downloads, as a (dashed-dotted) line graph. Just as waves of innovation ($\sigma_{\text{NFR}}$) 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 $\rho_{\text{Closed,Downloads}} = 0.43$, 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 I shows the correlations for the eight projects that had sufficient data (e.g., feature-closed statistics) for analysis. The column heading are defined as follows:

- \( \rho_{s, \text{closed}} \): Pearson’s correlation coefficient between \( \sigma_{\text{NFR}} \) and the number of Closed features
- \( \rho_{\text{closed}, \text{downloads}} \): 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)
- Weekly Downloads: The number of weekly downloads (from SourceForge). 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.

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 I \( \rho_{s, \text{closed}} \) and \( \rho_{\text{closed}, \text{downloads}} \) values suggest that this theory may be worth more exploration. More 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.

Consider AwStats from Table I. The value of \( \rho_{\text{closed,downloads}} = -0.40 \) seems to present a counterexample. Consider Figure 5, which graphs NFR variance, Closed features, and Downloads for AwStats. Notice that there are relatively few closed features after point 10. In comparing the waves of innovation, indicated by NFR variance (StdDev), with the wave of close 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 \( \rho_{\text{closed,downloads}} \) correlations of Compiere and WinMerge may be explained in similar fashion.

Consider mapping \( \rho_{\text{closed,downloads}} \) onto three values:
- Low = \( \rho_{\text{closed,downloads}} < -0.15 \)
- Medium = \(-0.15 \leq \rho_{\text{closed,downloads}} < 0.15 \)
- High = \( \rho_{\text{closed,downloads}} \geq 0.15 \)

Projects with \( \rho_{\text{closed,downloads}} = \text{High} \) are consistent with the \( \sigma_{\text{NFR}} \) innovation wave theory. The others may have other factors that prevent innovative features from increasing downloads. Using the attributes of Table I as inputs, we applied decision tree data-mining to derive the following classification rules:

<table>
<thead>
<tr>
<th>System</th>
<th>( \rho_{s, \text{closed}} )</th>
<th>( \rho_{\text{closed,downloads}} )</th>
<th>Features Solved</th>
<th>Patches Solved</th>
<th>Closed/Reqs</th>
<th>Weekly Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>awstats</td>
<td>-0.20</td>
<td>-0.40</td>
<td>31%</td>
<td>63%</td>
<td>0.047</td>
<td>714,553</td>
</tr>
<tr>
<td>compiере</td>
<td>-0.08</td>
<td>-0.18</td>
<td>53%</td>
<td>97%</td>
<td>0.107</td>
<td>114,068</td>
</tr>
<tr>
<td>filezilla</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>57,516</td>
</tr>
<tr>
<td>fire</td>
<td>-</td>
<td>-</td>
<td>80%</td>
<td>0%</td>
<td>-</td>
<td>31,148</td>
</tr>
<tr>
<td>floats</td>
<td>-</td>
<td>-</td>
<td>28%</td>
<td>100%</td>
<td>-</td>
<td>15,214</td>
</tr>
<tr>
<td>gallery</td>
<td>-</td>
<td>-</td>
<td>73%</td>
<td>76%</td>
<td>0.136</td>
<td>5,163</td>
</tr>
<tr>
<td>keepass</td>
<td>-0.40</td>
<td>0.43</td>
<td>79%</td>
<td>99%</td>
<td>0.140</td>
<td>4,169</td>
</tr>
<tr>
<td>megamek</td>
<td>-0.10</td>
<td>-0.01</td>
<td>82%</td>
<td>98%</td>
<td>0.069</td>
<td>2,073</td>
</tr>
<tr>
<td>pcgen</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1,829</td>
</tr>
<tr>
<td>phpmyadmin</td>
<td>-0.40</td>
<td>0.09</td>
<td>49%</td>
<td>89%</td>
<td>0.107</td>
<td>1,550</td>
</tr>
<tr>
<td>popfile</td>
<td>0.26</td>
<td>0.17</td>
<td>77%</td>
<td>92%</td>
<td>0.157</td>
<td>522</td>
</tr>
<tr>
<td>sourceforge</td>
<td>-</td>
<td>-</td>
<td>88%</td>
<td>97%</td>
<td>-</td>
<td>766</td>
</tr>
<tr>
<td>tikiwiki</td>
<td>-</td>
<td>-</td>
<td>23%</td>
<td>61%</td>
<td>-</td>
<td>269</td>
</tr>
<tr>
<td>tortoise</td>
<td>-</td>
<td>-</td>
<td>61%</td>
<td>95%</td>
<td>-</td>
<td>216</td>
</tr>
<tr>
<td>winmerge</td>
<td>0.06</td>
<td>-0.19</td>
<td>61%</td>
<td>97%</td>
<td>0.161</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 5 NFR variance (StdDev) and Closed features (scaled left) with Downloads (scaled right) for 6-month windows of AwStats.
1. If $\rho_{\text{closed}} > -0.26$, then $\rho_{\text{closed,downloads}} = \text{Low}$
2. If $\rho_{\text{closed}} \leq -0.26$, and...
   a. Closed/Reqs > 0.107 then $\rho_{\text{closed,downloads}} = \text{High}$
   b. Closed/Reqs $\leq 0.107$ then $\rho_{\text{closed,downloads}} = \text{Medium}$

These rules cover the 7 projects (having $\rho_{\text{closed}}$) with only 1 misclassification. The rules support the theory in that $\rho_{\text{closed}}$ affects $\rho_{\text{closed,downloads}}$ the most. Additionally, these rules suggest that Closed/Reqs affects $\rho_{\text{closed,downloads}}$. 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 a Closed/Reqs ratio—too many requirements are being considered relative to the number of features being closed. This suggests that the many requirements ideas being discussed are reducing the effort to close features.

3.4 Requirements traceability focus

Traceability plays a 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. During testing, developers will trace from tests 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.

Our analysis reveals that open source has a greater emphasis on operability than traceability. Both Figure 6 and Figure 7 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 8 shows this distinction more clearly by graphing the ratio of operability/traceability for KeePass, AwStats, Compiere, and WinMerge. (In this study, RTF 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 two projects, Compiere and WinMerge, similarly have points where their ratio raises above their average. Thus, the three projects that have $\rho_{\text{closed,downloads}} < 0$ (and thus seem inconsistent with the $\sigma_{\text{NFR}}$ innovation theory) all have spikes in their operability/traceability ratio. When this distinction is considered, the data set is consistent with the theory.

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:

- $\rho_{\text{closed}}$ is weakly positive, indicating difficulty in converting innovations ($\sigma_{\text{NFR}}$) into closed features
- Closed/Reqs is low (with Closed↓), indicating more emphasis on requirements than on implementing them
- operability/traceability has spikes (with operability↓ & traceability↑), indicating that traceability, and thus understanding the development, has become an issue
- $\rho_{\text{closed,downloads}} < 0$ (with Closed↓ & Downloads↓), 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 ($\sigma_{\text{NFR}}$) into the delivered product (Closed↑ & Downloads↑).

4 Discussion

The previous section summarizes our exploratory analysis of 16 open-source projects using NL requirements parsing and NFR classification. We began this analysis to show the kinds of analysis made possible through automated requirements discovery and classification, as provided by RCNL. Because of this analysis, we have come to posit our simple $\sigma_{\text{NFR}}$ innovation theory, which conjectures a sequential, wave-like process from requirements innovation ($\sigma_{\text{NFR}}$) to closing features to increased downloads. RCNL can aid in such theory formation.

The $\sigma_{\text{NFR}}$ innovation theory remains a conjecture until more data can be analyzed and more formal modeling of the time-shifted process correlations can be had. Additionally, underlying assumptions should be validated. For example, it should be validated that increase $\sigma_{\text{NFR}}$ activity results in increased innovation, rather than simply more randomized requirements. Likewise, it should be validated that increased operability/traceability spikes (with operability↓ & traceability↑) is indicative of developers having trouble converting feature request into closed features. Such detailed validation may require a grounded theory approach to analyzing the meaning of the underlying artifacts. In the meantime, however, this case study demonstrates how RCNL can aid analysis of requirements properties that are difficult to uncover, except through arduous manual analysis.

Returning to Pat’s questions from the Introduction section, we would have to recommend that Pat forgo invitations to work on AwStats, Compiere, and WinMerge. Given our dataset, Pat should consider working on KeePass PhpGedGiew or PhpMyAdmin because, according to the posited theory, these projects can transform abstract requirements innovations into the delivered product.

5 Conclusions

This article demonstrates how a NL requirements parsing and NFR 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 life cycle. The approach assumes four common steps:

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

The resulting correlations provide a basis to explore how open source developers do their work. More substantial statistical analysis will be required to validate theories.

This article presented a case study demonstrating the use of RCNL is exploring relationships among requirements. The demonstration posited a theory that innovations expressed as NFRs 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. The theory remains a conjecture for now. However, the use of RCNL to aid in theory exploration seems worthwhile. Our future research is aimed to expand the capabilities of RCNL and facilitate its integration with common qualitative and quantitative tools (e.g., SPSS) to simplify requirements research on open source projects.

6 References
