# Automatic Generation of Release Notes

Laura Moreno[1], Gabriele Bavota[2], Massimiliano Di Penta[2],
Rocco Oliveto[3], Andrian Marcus[1], Gerardo Canfora[2]
[1]Wayne State University, Detroit, MI, USA
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
lmorenoc@wayne.edu, gbavota@unisannio.it, dipenta@unisannio.it,
rocco.oliveto@unimol.it, amarcus@wayne.edu, canfora@unisannio.it

## ABSTRACT

This paper introduces ARENA (**A**utomatic **RE**lease **N**otes gener**A**tor), an approach for the automatic generation of release notes. ARENA extracts changes from the source code, summarizes them, and integrates them with information from versioning systems and issue trackers. It was designed based on the manual analysis of 1,000 existing release notes. To evaluate the quality of the ARENA release notes, we performed three empirical studies involving a total of 53 participants (45 professional developers and 8 students). The results indicate that the ARENA release notes are very good approximations of those produced by the developers and often include important information that is missing in the manually produced release notes.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, Enhancement, Restructuring, Reverse Engineering, and Reengineering*

## General Terms

Documentation

## Keywords

Release notes, Software documentation, Software evolution

## 1. INTRODUCTION

When a new release of a software project is issued, development teams produce a *release note*, usually included in the release package, or uploaded on the project's website. A release note summarizes the main changes that occurred in the software since the previous release, such as new features, bug fixes, changes to licenses under which the project is released, and, especially when the software is a library used by others, relevant changes at code level.

Producing release notes by hand can be an effort-prone and daunting task. According to a survey that we conducted among open-source and professional developers (see Section 4.2), creating a release note is a difficult and effort-prone activity that can take up to eight hours. Some issue trackers can generate simplified release notes (e.g., the *Atlassian OnDemand release note generator*[1]), but such notes merely list closed issues that developers have manually associated to a release.

This paper proposes ARENA (**A**utomatic **RE**lease **N**otes gener**A**tor), an approach for the automatic generation of release notes. ARENA identifies changes occurred in the commits performed between two releases of a software project, such as structural changes to the code, upgrades of external libraries used by the project, and changes in the licenses. Then, ARENA summarizes the code changes through an approach derived from code summarization [1, 2]. These changes are linked to information that ARENA extracts from commit notes and issue trackers, which is used to describe fixed bugs, new features, and open bugs related to the previous release. Finally, the release note is organized into categories and presented as a hierarchical HTML document, where details on each item can be expanded or collapsed, as needed. It is important to point out that (i) the approach has been designed after manually analyzing 1,000 project release notes to identify what elements they typically contain; and (ii) *ARENA produces detailed release notes, mainly intended to be used by developers and integrators.*

From an engineering standpoint, ARENA leverages existing approaches for summarizing code and for linking code changes to issues; yet, it is novel and unique for two reasons: (i) it generates summaries and descriptions of code changes at different levels of granularity than what was done in the past; and (ii) it combines code analysis, summarization, and mining approaches together to address the problem of release note generation, *for the first time.*

We performed three different empirical studies to evaluate ARENA, having different objectives: (i) evaluating the *completeness* of ARENA release notes with respect to manually created ones; (ii) collecting from developers (internal and external to the projects from which release notes were generated) opinions about the *importance* of the additional information provided by ARENA release notes, which is missing in the existing ones; and (iii) doing an *"in-field" study* with the project leader and developers of an existing software system, to compare their release notes with the ones generated by ARENA.

In summary, this paper makes the following contributions:

1. The ARENA approach to automatically generate re-

[1]`http://tinyurl.com/atlassianRN`

**Table 1: Contents of the 1,000 release notes (CC = Code Components).**

| Content Type | #Rel. Notes | % |
|---|---|---|
| Fixed Bugs | 898 | 90% |
| New Features | 455 | 46% |
| New CC | 428 | 43% |
| Modified CC | 401 | 40% |
| Modified Features | 264 | 26% |
| Refactoring Operations | 209 | 21% |
| Changes to Docum. | 204 | 20% |
| Upgraded Library Dep. | 158 | 16% |
| Deprecated CC | 100 | 10% |
| Deleted CC | 91 | 9% |
| Changes to Config. Files | 84 | 8% |
| Changes to CC Visibility | 70 | 7% |
| Changes to Test Suites | 73 | 7% |
| Known Issues | 64 | 6% |
| Replaced CC | 48 | 5% |
| Architectural Changes | 30 | 3% |
| Changes to Licenses | 18 | 2% |

lease notes.

2. Results of three empirical evaluations aimed at assessing the quality of the generated release notes.
3. Results of a survey analyzing the content of 1,000 release notes from 58 industrial and open source projects.
4. Results of a survey with 22 developers on the difficulty of creating release notes.

**Replication package.** A replication package is available online[2]. It includes: (i) complete results of the survey; (ii) code summarization templates used by ARENA; (iii) all the HTML generated release notes; and (iv) material and working data sets of the three evaluation studies.

**Paper structure.** Section 2 reports results of our initial survey to identify requirements for generating release notes and Section 3 introduces ARENA. Section 4 presents the three evaluation studies and their results, while threats to their validity are discussed in Section 5. Section 6 explains how this work relates to other research. Finally, Section 7 concludes the paper and outlines directions for future work.

## 2. WHAT DO RELEASE NOTES CONTAIN?

To design ARENA, we performed an exploratory study aimed at understanding the structure and content of existing release notes. We manually inspected 1,000 release notes from 58 software projects to analyze and classify their content. The analyzed notes belong to 608 releases of 41 open-source projects from the Apache ecosystem (*e.g.,* Ant, log4j, *etc.*), 382 releases of 14 open-source projects developed by other communities (*e.g.,* jEdit, Selenium, Firefox, *etc.*), and ten releases of three industrial projects (*i.e.,* iOS, Dropbox, and Jama Contour). Different communities produce release notes according to their own guidelines, as industry-wide common standards do not exist.

Release notes are usually presented as a list of items, each one describing some type of change. Table 1 reports the 17 types of changes we identified as usually included in release notes, the number of release notes containing such information, and the corresponding percentage. Note that *Code Components* (denoted by CC in Table 1) may refer to classes, methods, or instance variables.

Bug fixes stand out as the most frequent item included in the release notes (in 898 release notes—90% of our sample). Typically, the information reported is a simple bullet list containing, for each fixed bug, its ID and a very short summary (often the bug's title as stored in the bug tracking system). For example, in the release note of Apache Lucene 4.0.0, the LUCENE-4310 bug fix is reported as follows:

> *LUCENE-4310: MappingCharFilter was failing to match input strings containing non-BMP Unicode characters.*

Other frequently reported changes in release notes are the new features (46%) and new code components (43%). Often these two types of changes are reported together, explaining what new code components were added to implement the new features.

Modified code components (*i.e.,* classes, methods, fields, parameters) are also frequently reported (40%). Note that we include here all cases where the release notes report that a code element has been changed, without specifying how. We do not include here deprecated code components or changes to code components' visibility that are classified separately (see Table 1).

Explanations of modified features are quite frequent in release notes (26%) and are generally accompanied by the code components that were added/modified/deleted to implement the feature change. An example from the release note of the Google Web Toolkit 2.3.0 (M1) is:

> *Updated GPE's UIBinder editor (i.e., class UIBinder of the Google Plug-in) to provide support for attribute auto-completion based on getter/setters in the owner type.*

Refactoring operations are also included in release notes (21%), generally as simple statements specifying the impacted code components, *e.g.,* "*Refactored the WebDriver-Js*"—from Selenium 2.20.0.

Changes in documentation are present in 20% of the analyzed release notes, although, more often than not, they are rather poorly described with general sentences like "*more complete documentation has been added*" or "*documentation improvements*". We also found: 158 release notes (16%) reporting upgrades in the libraries used by the project (*e.g.,* "*The Deb Ant Task now works with Ant 1.7*"—from Jedit 4.3pre11); 100 (10%), reporting deprecated code components (*e.g.,* "*The requestAccessToAccountsWithType: method of ACAccountStore is now deprecated*"—from iOS 6.0); and 91 (9%), including deleted code components (*e.g.,* "*Removed GWTShell, an obsolete way of starting dev mode*"—from Google Web Toolkit 2.5.1 (RC1)).

Other changes performed in the new release are less common in the analyzed release notes (see Table 1). We must note that rarely summarized types of changes are not necessarily less important than the frequently reported ones. It may simply be the case that some types of changes occur less frequently than others, hence they are reported less. For example, changes to licenses are generally rare and thus, only 18 release notes (2%) contain this information. We do not equate frequency with importance. Future work will answer the *importance* question separately.

Based on the results of this survey and on our assessment of what it can be automatically extracted from available sources of information—*i.e.,* release archives, versioning systems, and issue trackers—we have formulated requirements for what ARENA should include in release notes: (i) a description of fixed bugs, new features, and improvement of existing features, complemented with a description of what was changed in the code; (ii) a summary of other source code

[2] `http://tinyurl.com/qalutua`

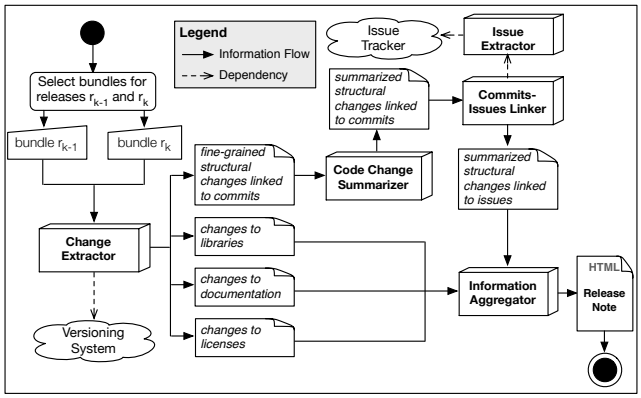


**Figure 1: Overview of the ARENA approach.**

changes, including deprecated methods/classes and refactoring operations; (iii) changes to the libraries used by the system; (iv) changes to licenses and documentation; and (v) open issues. Note that in the current version of ARENA we did not consider: (i) changes to configuration and build files, because there is a plethora of different possible configuration files; and (ii) changes to the high-level system architecture, because existing architecture recovery approaches (*e.g.,* [3, 4, 5]) might require manual effort to produce usable results.

# 3. ARENA OVERVIEW

In a nutshell, ARENA works as depicted in Figure 1. The process to generate release notes for a release $r_k$ is composed of four main steps. First, the *Change Extractor* is used to capture changes performed between releases $r_{k-1}$ and $r_k$. In the second step, the *Code Change Summarizer* describes the fine-grained changes captured by the *Change Extractor*, with the goal of obtaining a higher-level view of what changed in the code. In the third step, the *Commit-Issue Linker* uses the *Issue Extractor* to mine information (*e.g.,* fixed bugs, new features, improvements, *etc.*) from the issue tracker of the project to which $r_k$ belongs. Thus, each fixed bug, implemented new feature, and improvement is linked to the code changes performed between releases $r_{k-1}$ and $r_k$. The summarized structural code changes represent *what* changed in the new release, while the information extracted from the issue tracker explains *why* the changes were performed. Finally, the extracted information is provided as input to the *Information Aggregator*, in charge of organizing it as a hierarchy and creating an HTML version of the release note for $r_k$, where each item can be expanded/collapsed to reveal/hide its details. An example of a release note generated by ARENA can be found at `http://tinyurl.com/oelwef4`.

## 3.1 Code Changes Extraction

ARENA extracts code changes, as well as changes to other entities, from two different sources: the versioning system and the source archives of the releases to be compared.

**Identification of the time interval to be analyzed.** ARENA aims at identifying the subset of commits that pertains to the release for which the release note needs to be generated, say release $r_k$. To identify changes between release $r_{k-1}$ and $r_k$, we consider all commits occurred—in the main trunk of the versioning system—starting from the $r_{k-1}$ release date $t_{k-1}$, until the $r_k$ release date $t_k$. The dates can be approximate, as developers could start working on release $r_{k+1}$ even before $t_k$, *i.e.,* changes committed before $t_k$ could belong to release $r_{k+1}$ and not to release $r_k$. In a real usage scenario, a developer in charge of creating a release note using ARENA could simply provide the best time interval to analyze or, even better, *tag* the commits in the versioning system with the release number.

**Analysis of Code Changes.** Once the commits of interest are identified, ARENA's *Change Extractor* analyzes them using a code analyzer developed in the context of the *Markos* EU project[3]. The code analyzer parses the source code using the *srcML* toolkit [6] and extracts a set of facts concerning the files that have been added, removed, and changed in each commit. Information about the commits performed between release $r_{k-1}$ and $r_k$ is extracted from the versioning system (*e.g., git, svn*). Given the set of files in a commit, the following kinds of changes are identified:

- Files added, removed, and moved between packages;
- Classed added, removed, renamed, or moved between files. To detect moving and renaming, classes (and methods) are encoded by a metric-based fingerprinting [7, 8], which is used to trace the entities;
- Methods added, removed, renamed, or moved;
- Methods changed, *i.e.,* changes in the signature, visibility, source code, or set of thrown exceptions;
- Instance variables added, removed, and visibility changes;
- Deprecated classes and methods.

**Generation of Textual Summaries from Code Changes.** The fine-grained structural changes are provided to the *Code Change Summarizer* to obtain a higher-level view of what changed in the code (see Figure 1). To generate this view, ARENA's *Code Change Summarizer* follows three steps: (i) it hierarchically organizes the code changes; (ii) it selects the changes to be described; and (iii) it generates a set of natural-language sentences for the selected changes.

In the first step, a hierarchy of changed artifacts is built by considering the kind of artifact (*i.e.,* files, classes, methods, or instance variables) affected by each change. In Object-Oriented (OO) software systems, files contain classes, which in turn consist of methods and instance variables. Therefore, changes are grouped based on these relationships, *e.g.,* changed methods and instance variables that belong to the same class are grouped under that class.

In the second step, ARENA analyzes the hierarchy in a top-down fashion to prioritize and select the code changes to be included in the summaries, in the following way:

1. If a file is associated to class-level changes, high priority is given to the class changes and low priority to the file changes, since in OO programming languages classes are the main decomposition unit, rather than source files.
2. If the change associated to a class is its addition or deletion, high priority is given to it and low priority to any other change in the class (*e.g.,* the addition/deletion of its methods). Otherwise, changes on the visibility or deprecation of the class become of high priority. If the change associated to a class is its modification, high priority is given to the associated changes at instance variable and method level.

[3] `http://www.markosproject.eu`

```
New class SearcherLifetimeManager implementing Closeable.
This boundary class communicates mainly with
AlreadyClosedException, IndexSearcher, and IOException, and
consists mostly of mutator methods. It allows getting
record, handling release, acquiring searcher lifetime
manager, and closing searcher lifetime manager. This class
declares the helper classes SearcherTracker and PruneByAge.
```

**Figure 2: Summary for the *SearcherLifetimeManager* class from Lucene 3.5.**

3. If the change associated to an instance variable or method is its addition, deletion, renaming, or deprecation, the change is given a high priority. Changes to parameters, return types, or exceptions are given a low priority.

Finally, the *Code Change Summarizer* generates a natural language description of the selected changes, presented as a list of paragraphs. For this, ARENA defines a set of templates according to the kind of artifact and kind of change to be described. In this way, one sentence is generated for each change. For example, a deleted file is reported as *File <file_name> has been removed.*

As stated above, the focus of OO systems is on classes. Thus for added classes, ARENA provides more detailed sentences than for other changes, by adapting *JSummarizer* [9], a tool that automatically generates natural-language summaries of classes. Each summary consists of four parts: a general description based on the superclass and interfaces of the class; a description of the role of the class within the system; a description of the class behavior based on the most relevant methods; and a list of the inner classes, if they exist. We adapted *JSummarizer*, by modifying some of the original templates and by improving the filtering process when selecting the information to be included in the class summary. Figure 2 shows part of an automatically generated summary for the *SearcherLifetimeManager* class from Lucene 3.5.

Deleted classes are reported in a similar way to deleted files. Changes regarding the visibility of a class are described by providing the added or removed specifier, *e.g., Class <class_name> is now <added_specifier>*. The description for modified classes consists of sentences reporting added, deleted or modified methods and instance variables. For example, a change in a method's name is reported as: *Method <old_method_name> was renamed as <new_method_name>*.

The generated sentences are concatenated according to the priority previously assigned to the changes. To avoid text redundancies, ARENA groups similar changes in single sentences, *e.g., Methods <method_name$_1$>, ..., and <method_name$_n$> were removed from <class_name>*, rather than list them one at a time.

**Analysis of licensing changes.** To identify license changes, ARENA's *Change Extractor* analyzes the content of the source distribution of $r_{k-1}$ and $r_k$, extracting all source files (*i.e.,*`.java`) and all text files (*i.e.,*`.txt`). Then, it uses the *Ninka* license classifier [10] to identify and classify licenses contained in such files. *Ninka* uses a pattern-matching based approach to identify statements characterizing the various licenses and, given any text file (including source code files) as input, it outputs the license name (*e.g.,* GPL) and its version (*e.g.,* 2.0). Its precision is around 95%.

**Identification of changes in documentation.** This analysis is done on the release archives of $r_{k-1}$ and $r_k$. Although release archives can contain any kind of documentation, we only focus on changes to documentation describing source code entities. ARENA identifies documentation changes using the approach described below:

1. Identify text files, *i.e.,*`.pdf`, `.txt`, `.rdf`, `.doc`, `.docx`, and `.html`, and extract the textual content from them using the *Apache Tika*[4] library.
2. If a text file (say $doc_i$) has been added in $r_k$, then verify whether it contains references to a code file names, class names, and method names. We use a pattern matching approach, similar to the one proposed by Bacchelli *et al.* [11]. If such entities are found in a file, then check whether these files, classes, or methods have been added, removed, or changed in the source code, so that ARENA can generate an explanation of why the documentation was added, *e.g.,* if the added text file contains a reference to class $C_j$ added in $r_k$, ARENA describes the change as "*The file $doc_i$ has been added to the documentation to reflect the addition of class $C_j$*".
3. If a text file (say $doc_i$) has been removed in $r_k$, then check if it contains references to deleted methods/classes/code files. If that is the case, ARENA generates an explanation "*The file $doc_i$ has been deleted from the documentation due to the removal of* <involved_code _components> *from the system*".
4. If a text file has been modified between $r_{k-1}$ and $r_k$, we use a similar approach as above, however we search for references to code entities only in the portions of the text file that were changed.

**Identification of changes in the used libraries.** ARENA's *Change Extractor* analyzes whether: (i) $r_k$ uses new libraries—specifically jars—compared to $r_{k-1}$; (ii) libraries are no longer required; and (iii) libraries have been upgraded to new releases. The analysis is performed in two steps:

1. Parsing the files describing the inter-project dependencies. In Java projects, these are usually *property* files (`libraries.properties` or `deps.properties`) or Maven dependency files (`pom.xml`). The information from such files allows ARENA to detect the libraries used in both releases $r_k$ and $r_{k-1}$. For each library we detect its name and used versions, *e.g., ant*, v. *1.7.1.*
2. Identifying the set of *jars* contained in the two release archives and—by means of regular expression—extracting their name and version from the jar names, *e.g.,*`ant_1.7.1.jar` is mapped to library *ant*, v. *1.7.1.*

With the list of libraries used in both releases, ARENA verifies if: (i) new libraries have been added in $r_k$; (ii) libraries are no longer used in $r_k$; or (iii) new versions of libraries previously used in $r_{k-1}$ are used in $r_k$.

**Identification of refactoring operations.** ARENA's *Change Extractor* also identifies refactorings performed between the releases $r_{k-1}$ and $r_k$, by using two complementary sources of information:

1. Refactorings documented in the commit notes. Although not all refactorings are documented, we claim that the ones documented by developers are important

[4] `http://tika.apache.org`

and deserve to be included in the release notes. Such refactorings are identified by matching regular expressions in commit notes—*e.g.,refact*, *renam*—as done in previous work [12, 13].

2. Class/method renaming and moving (those not already identified by means of their commits using the above heuristic). Such refactoring changes are identified by means of fingerprinting analysis.

In principle, ARENA could describe other kinds of refactorings, which could be identified using tools like *RefFinder* [14]. For the time being, we prefer to keep a light-weight approach, to avoid generating excessively verbose release notes.

## 3.2 Issues Extraction

We use the versioning system to extract various kinds of changes to source code and other system entities, *i.e.,* to explain *what* in the system has been changed. In addition, we rely on the issue tracker to extract change descriptions, *i.e.,* to explain *why* the system has been changed. To this aim, the ARENA *Issue Extractor* (see Figure 1) extracts the following type of issues from the issue tracker:

- *Issues describing bug fixes:* Issues with Type=*"Bug"*, Status=*"Resolved"* or *"Closed"*, and Resolution=*"Fixed"*, with a resolution date included in the $[t_{k-1}, t_k]$ period.
- *Issues describing new features:* Same as above, but considering issues with Type= *"New Feature"*.
- *Issues describing improvements:* Same as for bug fixes, but considering issues with Type= *"Improvement"*.
- *Open issues.* Any issue with Status=*"Open"* and open date in the period $[t_{k-1}, t_k]$.

Note that open issues are collected to present in the release note $r_k$'s *Known Issues*. Based on the fields described above, ARENA has been implemented for the *Jira* issue tracker. However, it can be extended to other issue trackers (*e.g.,Bugzilla*), using the appropriate, available fields. Note that sometimes fields classifying issues as bug fix/new feature/enhancement are not fully reliable [15, 16].

## 3.3 Linking Issue Descriptions to Commits

To link issues to commits we use (and complement) two existing approaches. The first one is the approach by Fischer *et al.* [17], based on regular expressions matching the issue ID in the commit note. The second one is a re-implementation of the *ReLink* approach defined by Wu *et al.* [18], which considers the following constraints: (i) matching the committer/authors with issue tracking contributor name/email; (ii) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than 7 days; and (iii) Vector Space Model (VSM) [19] cosine similarity between the commit note and the last comment referred above greater than 0.7. This threshold has been chosen by manually analyzing the mapping produced by the linking approach on two systems (*i.e.,* Apache Commons Collections and JBoss-AS).

## 3.4 Generating the Release Note

ARENA's *Information Aggregator* is in charge of building the release note as an HTML document. The changes are presented in a hierarchical structure consisting of the categories from the ARENA requirements defined in Section 2 and items summarizing each change.

**Table 2: System releases used in each study.**

| Study | Name | Releases | KLOC | # of commits before release |
|---|---|---|---|---|
| *Study I* | Apache Cayenne | 3.0.2 | 248 | 5,118 |
| | Apache Cayenne | 3.1B2 | 232 | 2,550 |
| | Apache Commons Codec | 1.7 | 17 | 267 |
| | Lucene | 3.5.0 | 184 | 2,869 |
| | Jackson-Core | 2.1.0 | 21 | 170 |
| | Jackson-Core | 2.1.3 | 22 | 31 |
| | Janino | 2.5.16 | 26 | 612 |
| | Janino | 2.6.0 | 31 | 408 |
| *Study II* | Apache Commons Collections | 4.4.0ALPHA1 | 104 | 303 |
| | Lucene | 4.0.0 | 192 | 758 |
| *Study III* | SMOS | 2.0.0 | 23 | 109 |

# 4. EMPIRICAL EVALUATION

The *goal* of our empirical studies is to evaluate ARENA, with the *purpose* of analyzing its capability to generate release notes. The *quality focus* is the completeness, correctness and importance of the content of the generated release notes with respect to original ones, and the perceived usefulness of the information contained in the ARENA release notes. The *perspective* is of researchers, who want to evaluate the effectiveness of automatic approaches for the generation of release notes, and managers and developers, who could consider using ARENA in their own company.

We aim at answering the following research questions:

**$RQ_1$—Completeness:** *How complete are the generated release notes, compared with the original ones?* In other words, our first objective is to check whether ARENA is missing information that is contained in manually-generated release notes.

**$RQ_2$—Importance:** *How important is the content captured by the generated release notes, compared with the original ones?* The aim is assessing developers' perception of the various kinds of items contained in manually and automatically-generated release notes. We are interested in the usefulness of the additional details produced by ARENA, which are missing in the original notes.

To address our research questions, we performed three empirical studies having different settings and involving different kinds of participants. *Study I* aims at assessing the *completeness* of the ARENA release notes with respect to the original ones available on the system Web sites ($\mathbf{RQ}_1$). To ensure a good generalizability of the findings, we conducted this study on eight open source projects. In addition, since the goal of *Study I* does not require high experience and deep knowledge of the application domain (as it will be clearer later), we involved mainly students. *Study II* aims at evaluating the *importance* of the items present in the ARENA release notes and missing in the original ones and *vice versa* ($\mathbf{RQ}_2$). In this case, the task assigned to participants is highly demanding. This is the reason why we conducted the study only on two systems and we involved experts—including original developers of the projects from which the release notes were generated—since it was crucial to have people with experience and knowledge of the system. Finally, *Study III* is an *in-field study* addressing both $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, where we asked the original developers of a system, named SMOS, to evaluate a release note generated by ARENA and to compare it with one produced by the development team leader.

## 4.1 Study I—Completeness

The goal of this study is to assess the completeness of ARENA release notes ($\mathbf{RQ}_1$) on several system releases,

**Table 3: Evaluation provided by the study participants to the items in the original release notes.**

| System | Release | Absent | Less Details | Same Details | More Details |
|---|---|---|---|---|---|
| Apache Cayenne | 3.0.2 | 15% | 5% | 0% | 80% |
| Apache Cayenne | 3.1B2 | 16% | 0% | 0% | 84% |
| Apache Comm. Codec | 1.7 | 13% | 5% | 5% | 77% |
| Apache Lucene | 3.5.0 | 37% | 39% | 8% | 16% |
| Jackson-Core | 2.1.0 | 25% | 8% | 33% | 33% |
| Jackson-Core | 2.1.3 | 0% | 0% | 50% | 50% |
| Janino | 2.5.16 | 0% | 6% | 0% | 94% |
| Janino | 2.6.0 | 12% | 38% | 0% | 50% |
| **Average** | | **14%** | **13%** | **12%** | **61%** |

hence assuring a good external validity, both in terms of projects diversity and features to be included in the releases. The *context* of *Study I* consists of: *objects*, *i.e.,* automatically generated and original release notes from eight releases of five open-source projects (see Table 2); and *subjects* evaluating the release notes, *i.e.,* one B.Sc., five M.Sc., one Ph.D. student, one faculty, and two industrial developers. Before conducting the study, we profiled the participants using a pre-study questionnaire, aimed at collecting information about their programming and industrial experience. To select releases to analyze, we used the following criterion. A release $r_k$ was selected if: (i) the original $r_k$ release note was available, and (ii) the release bundles for $r_k$ and $r_{k-1}$ were available. Additionally, we made sure that items from each change type (see Table 1—except for configuration file and architectural changes) were present in at least one of the eight release notes.

**Design and Planning.** We distributed the release notes to the evaluators, in such a way that each release note was evaluated by two participants. We provided each participant with (i) a pre-study questionnaire; (ii) the generated release note; and (iii) the original release note.

Participants were asked to determine and indicate for each item in the original release note whether: (i) the item appears in the generated release note with roughly the same level of detail; (ii) the item appears in the generated release note but with less details; (iii) the item appears in the generated release note and it has more details; or (iv) the item does not appear in the generated release note. In order to avoid bias in the evaluation, we did not refer to the release notes as "original" or "generated". Instead, we labeled them as *"Release note A"* and *"Release note B"*.

When all the participants completed their evaluation, a Ph.D. student from the University of Sannio analyzed them to verify and arbiter some conflicts in the evaluation of the items present in the original release notes. Out of 144 evaluated items, 43 (30%) exhibited some conflict between the two evaluators. Only five of them (3%) showed strong conflict between the evaluators, e.g., "*the item appears*" vs. "*the item is missing*". The other 38 cases had slight deviations in the evaluation, e.g., "*the item appears with roughly the same level of detail*" vs. "*the item appears but with less details*".

**Participants' background.** Six out of ten evaluators have experience in industry, ranging from one to five years (median 1.5). They reported four to 20 years (median 5.0) of programming experience, of which two to seven are in Java (median 4.5). Seven out of the ten evaluators declared that they routinely check release notes when using a new release.

**Results.** Table 3 summarizes the answers provided by the evaluators when asking about the presence of items from the original release notes in the release notes generated by ARENA. On average, ARENA correctly captures, at different levels of detail, 86% of the items from the original release, missing only 14%. In particular, ARENA provides more details for 61% of the items present in the original release notes, the same level of details for 12%, and less details for 13% of the items. The following is an exemplar situation where an item in the generated release has more details than in the original release. In the release note of Apache Commons Codec 1.7, the item "*CODEC-157 DigestUtils: Add MD2 APIs*" describes the implementation of new APIs. In the ARENA release note, the same item is reported as:

- CODEC-157 DigestUtils: Add MD2 APIs
  - New methods `getMd2Digest()`, `md2(byte)`, `md2(InputStream)`, `md2(String)`, `md2Hex(byte)`, `md2Hex(InputStream)`, and `md2Hex(String)` in `DigestUtils`.
  - New methods `testMd2Hex()`, `testMd2HexLength()`, and `testMd2Length()` in `DigestUtilsTest`.

ARENA reports the addition of new APIs to the `DigestUtils` class and it also explicitly includes: (i) which methods are part of the new APIs; and (ii) the test methods added in `DigestUtilsTest` to test the new APIs.

An outlier case is for Lucene 3.5.0, where ARENA missed 14 (37%) of the items present in the original release note. Upon a closer inspection, we found that eight of the missed items are bug reports fixed in a previous release, yet (for an unknown reason) reported in the release note of Lucene 3.5.0 (*e.g.,* issue *LUCENE-3390*). If we disregard such issues, the percentage of missed items in this release drops to 20%, almost in line with the other release notes.

We analyzed the items that ARENA missed in the other release notes. We found that all the missed items are due to a slight deviation between the time interval analyzed by ARENA and the one comprising the changes considered in the original release note. As explained in Section 3, we make an assumption about the time period of analysis, going from the $r_{k-1}$ release date $t_{k-1}$ until the $r_k$ release date $t_k$. This problem would not occur in a real usage scenario.

**Summary of Study I ($RQ_1$)—Completeness**. The ARENA release notes capture most of the items from the original notes (86% on average)—many missed items can be included by simply adjusting the considered time interval.

## 4.2 Study II—Importance

The goal of *Study II* is to evaluate the importance of the captured and missed items in the ARENA release notes from the perspective of external users/integrators as well as from the perspective of internal developers. The *context* of this study consists of: *objects*, *i.e.,* automatically generated and original release notes from one release of two open-source projects (see Table 2); and *subjects* evaluating the release notes, *i.e.,* 38 professional and open-source developers, including three developers of each object project. One release of Apache Lucene and one release of Apache Commons Collections were selected for the study. The conditions to select the release notes were the same as of *Study I*.

**Design and Planning.** We performed *Study II* by using an online survey. We emailed the survey to several open-source developers registered in the Apache repositories and professional developers from around the world. The questionnaire consisted of two parts on: (i) the participants' background and their experience in using and creating release notes; and (ii) the evaluation of the ARENA release notes and the original release notes for Apache Lucene 4.0.0 and for Apache Commons Collections 4.4.0ALPHA1.

The evaluation of each release note was divided in two

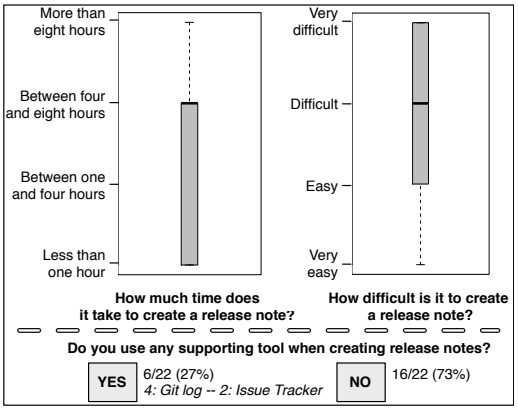


**Figure 3: Difficulty in creating release notes.**

stages. In the first one, participants were asked to indicate for several kinds of items (*e.g., Major Changes*) of the original release note that were missing in the generated release note whether each one was: (i) not at all important; (ii) unimportant; (iii) important; or (iv) very important. A similar process was followed in the second stage, but this time for assessing the importance of some kinds of items of the release note generated by ARENA that were missing in the original release note. In both stages, we pointed out to developers that some items present in a release note and (apparently) missing in the other one might simply be represented in different ways. In the case of Lucene 4.0.0, for example, the items under the *Improvements* category in the ARENA release note are listed under the *Optimizations* category in the original release note.

**Participants' background.** 31 out of 38 evaluators are professional developers, who reported experience in software development ranging from two to 30 years (median 8). The other seven participants are open-source developers, ranging from seven to 25 years of experience in software development (median 13). Three of the participants are developers of Apache Commons Collections, while other three are developers of Apache Lucene. Also, 26 out of the 38 evaluators declared that they use release notes frequently (*i.e.,* more than once a month) or occasionally (*i.e.,* once a month), mainly to check for bug fixes and new features in a software system. 16 developers declared that they check in the release notes of their project's dependencies for compatibility issues and changes that might arise from the new releases.

**Creating release notes.** Only six evaluators (16%) reported never having created release notes. Of the other 32 participants, 22 (58%) reported having created release notes many times (*i.e.,* more than eight times), eight (21%) reported having done it a few times (*i.e.,* three to eight times), and two (5%) declared having created a release note once or twice. Our survey was designed to ask only developers having a high experience in creating release notes (*i.e.,* the 22 cited above) details about their experience when creating release notes. Figure 3 presents the participants' answers on the time and difficulty of creating release notes. Specifically, 64% of the evaluators (*i.e.,* 14 of them) considered this task as difficult or very difficult, while 36% rated it as easy or very easy (median=difficult). The participants reported a median of *between four and eight hours* to create release notes. One of the participants explained that the time needed to create a release note depends on the release, claiming that he worked in the past on *"a major release of a software company product for which the creation of the release note took three days of work."* Finally, only seven evaluators (31%) declared using a supporting tool, such as, issue trackers or version control systems, when creating release notes. Note that ARENA is the only existing automated tool specially designed to support the generation of release notes. Furthermore, ARENA is able to shorten the time devoted to this task, as less than five minutes are needed to generate a release note for a medium-sized system.

We also asked the 22 participants with high experience in creating release notes about the kind of content that they usually include in these documents (see Figure 4). *New Features* and *Bug Fixes* are, by far, the most common items in the release notes: most of the participants reported including them often (21 and 20 evaluators, respectively). *Enhanced Features* are also frequently included in the release notes (often by 11 participants and sometimes by ten). Both results confirm the findings of our survey on 1,000 existing release notes presented in Section 2. Other frequently included items are *Deleted and Deprecated Code Components*, *Changes to Licenses*, *Library Upgrades*, and *Known Issues* (median=often), while "sometimes" developers include items related to *Added*, *Replaced* or *Modified Code Components*; and *Refactoring Operations*. On the other hand, evaluators rarely include changes to *Configuration Files*, *Documentation*, *Architecture*, and *Test Suites* in the release notes.

**Results.** Going to the core of this study, the answers provided by the 38 developers on the importance of different kinds of content from the original and the generated release notes are summarized in Figures 5 and 6 for Commons Collections and Lucene, respectively. Among the items present in the original release notes and missed by ARENA, the ones considered important/very important by developers are: *Major Changes* from Commons Collections, summarizing the most important changes in the new release; *API Changes*, *Backward Compatibility*, and *Optimizations* from Lucene. The *Major Changes* section is not present in the ARENA release notes and our future efforts will be oriented to implement an automatic prioritization of changes in the new release, which will allow ARENA to select the most important ones to define the *Major Changes* category. On the other hand, the information present in *API Changes*, *Backward Compatibility*, and *Optimizations* in the original release notes is present in the ARENA release notes, but simply organized differently. For example, the removal of the *SortedVIntList* class is reported in the original release notes in the *Backward Compatibility* category, while ARENA puts it under *Deleted Code Components*. Thus, ARENA is not missing any important information here. As for the other items present in the original release notes and not in the ones generated by ARENA, they are generally classified as unimportant/not important (see Figures 5 and 6).

Regarding the contents included in the ARENA release notes and missing or grouped in different categories in the original release notes, most of them (nine of 11 different kinds of content) were predominantly assessed as important or very important (by 28 developers, in average). The *Improvements* category is considered important/very important by 34 developers for Commons Collections and 33 for Lucene, respectively. While the items contained in this category are present in the *Optimizations* section of the Lucene release note, they are absent in the Commons Collections one. Important or very important were also considered the categories covering *Known Issues* (by 32 developers) and

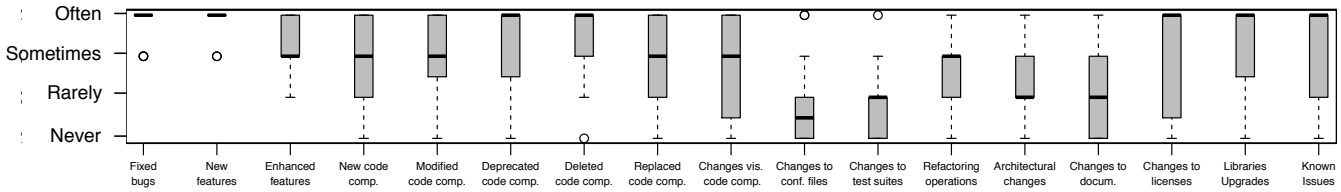


**Figure 4: What kind of content do you include in release notes? (22 developers)**

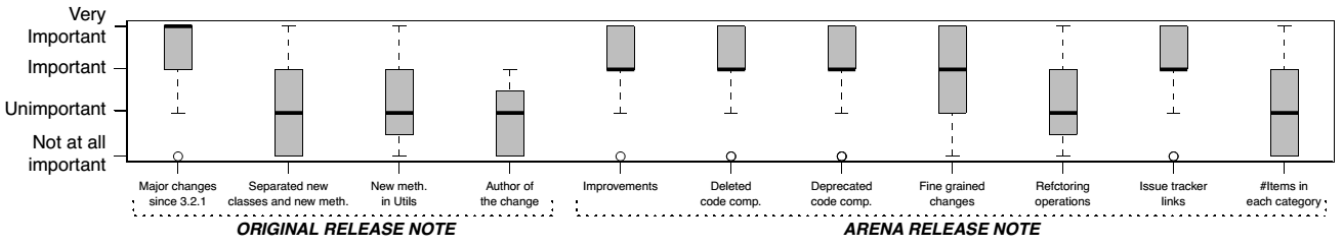


**Figure 5: Importance reported by the evaluators for the content of Commons Collections release notes.**

*Deletion* (29), *Addition* (28), *Deprecation* (29), and *Visibility Changes* (26) *of Code Components*. The evaluators (24 for Commons Collections and 19 for Lucene) also considered important the *fine-grained changes* (*i.e.,* changes at code level) provided by ARENA when describing new features, bug fixes and improvements, and the *links to the change requests* (30) listed under such categories. Note that most of the above categories are absent in the original release notes. For instance, while in the original Lucene release note one deleted class was listed under the *Backward Compatibility* section, ARENA highlights 76 classes that have been deleted in the new release. Surprisingly, *Refactoring Operations* were considered as unimportant in both release notes (by 24 developers for Commons Collections and 25 for Lucene). This might be due to the level of details that ARENA is currently able to provide in this matter (*i.e.,* the refactored source code files, without explaining what exactly was done to them).

**Evaluation made by the original developers.** As mentioned before, six (three each) of the original Lucene and Commons Collections developers took part in our survey. In the case of the contents explicitly provided by the original release note of Commons Collections, its three developers strongly agreed on the importance of the *Major Changes* category and weakly agreed on the importance of showing the *author of the change* (which was marked as important by two developers and unimportant by the other one). In contrast, having different categories describing new classes and new methods (as opposed to ARENA's single *New Code Components* category) was ranked from very unimportant to very important, not allowing us to draw any conclusion. The three developers strongly agreed on the importance of all the characteristics offered by the release note generated by ARENA, except for the *number of items in each category* (*e.g.,* indicating the number of the added code components near the *New Code Components* category), which was considered unimportant by two of the developers and very important by the other one.

In the case of Lucene, its three developers strongly agreed on the importance of the *API Changes* and *Backward Compatibility* categories of the original release note and on the unimportance of the *authors of the changes*. For the other contents of the original release note, there was little agreement. However, once again, the three developers strongly agreed on the importance of all the contents included in the ARENA release note, except for the *Refactoring Operations* category, which was marked as important by two of the developers and as unimportant by the other one. Most of the developers' responses go in line with the responses of all the other evaluators presented above.

**Qualitative feedback**. We also allowed developers to comment on the ARENA release notes in a free text box at the end of the survey. Evaluators provided positive feedback about the release notes generated by ARENA. A representative one is: *"In general, they are very readable. I think they are aimed at engineers more than at non-engineers. [...] for something that is consumed as an API, such as an open source library or framework, I think these kind of notes are ideal."* Another developer commented *"If it's fully automated (I'm not sure) ARENA is a great tool."* Note that some of the contents provided by the generated release notes were considered unimportant in some cases. One of the reasons behind such assessments was that *"Every item needs a description to be useful. For example, the Added Components section needs a description of each component, and the Modified Components section needs a description of what is the meaning of the modification."* This happens for the commits that cannot be linked to any bug fix, improvement, or new feature request in the issue tracker system. Since ARENA is meant to support the creation of release notes, they can be augmented by the users according to their needs.

**Summary of Study II ($RQ_2$)—Importance**. Most information included in the ARENA release notes is considered important or very important by developers. Also, most information considered as important in the original release notes is captured by ARENA (although sometimes in a different fashion), with the exception of the *Major changes* category that we plan to include by prioritizing changes.

## 4.3 Study III—"In-field" Evaluation

The goal of *Study III* is to allow project experts: (i) to evaluate the generated release note on its own, including their perceived usefulness; and (ii) to compare the generated release note with one produced manually by their team leader. The *context* of this study consists of one release of the SMOS system and five (out of seven) members of the original development team of this system. SMOS is a software developed for schools and supports the communications between the school and the students' parents. The first release of SMOS (*i.e.,* SMOS 1.0) was developed in 2009 by

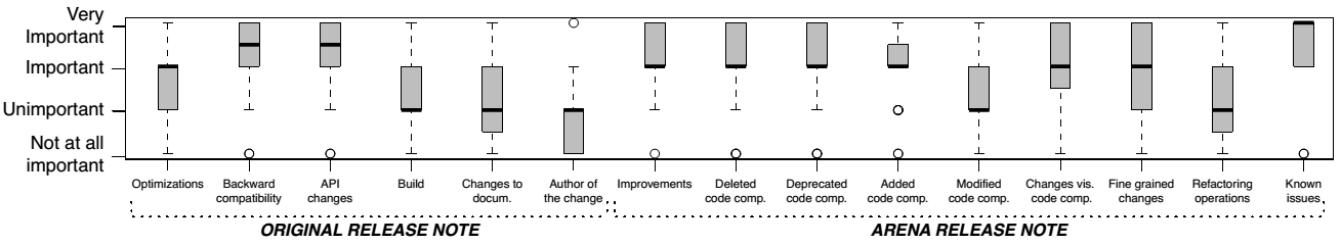


**Figure 6: Importance reported by the evaluators for the content of Lucene release notes.**

M.Sc. students at the University of Salerno during their industrial internship. Its code, composed by almost 23 KLOC, is hosted in a *Git* repository and has been subject to several changes over time. It led to the second release (*i.e.,* SMOS 2.0) nearly two years after release 1.0. The developers involved in our study worked on the evolution of SMOS during these two years and, since then, they have been working for more than three years in industry.

**Design and Planning.** To have a baseline for comparison, we asked the leader of the original SMOS development team to generate the release note for version 2.0 of the system. During the definition of the release note, the leader had access to: (i) the source code of the two releases; (ii) the list of changes (as extracted from the versioning system) performed between the two releases; and (iii) information from the issue tracker system containing the change requests implemented in the time period between SMOS 1.0 and 2.0. Finally, we asked him to report how much time he spent on producing the release note.

We conducted this study in two stages. In the first stage, the developers (excluding the project leader) evaluated the release note generated by ARENA based only on their knowledge of the system. As in *Study I*, we did not refer to this one as an automatically generated note (but as *Release note A*). We asked the developers to judge four statements with the possible answers on a 4-points Likert scale (*strongly agree*, *weakly agree*, *weakly disagree*, *strongly disagree*). We also asked them to judge the extent to which the generated release note would be useful for developers in the context of a maintenance task and what kind of additional information should be included in the release note. In the second stage, we asked the SMOS developers to evaluate—using the same questionnaire—the release note manually produced by the team leader, calling it *Release note B*.

**Results.** The team leader took 82 minutes to manually summarize the 109 changes that occurred between releases 1.0 and 2.0. This resulted in a release note with 11 items, each one grouping a subset of related changes. For example, one of the items in this release note was:

> *Several changes have been applied in the system to implement the Singleton design pattern in the classes accessing the SMOS database. Among the most important changes, all constructors of classes in the* `storage` *package should now not be used in favor of the new methods* `getInstance()`*, returning the single existing instance of the class. This resulted in several changes to all methods in the system using classes in* `storage`*.*

In the meantime, the remaining four developers evaluated the release note generated by ARENA. The results are reported in Table 4 (see the numbers *not* in parenthesis).

Developers *strongly agreed* or *weakly agreed* on the fact that the ARENA release note contains all the information needed to understand the changes performed between the two SMOS releases. In particular, the only developer answering *weakly agree* (id 1 in Table 4) explained that *"the release note contains all what is needed. However, it would be great to have a further level of granularity showing exactly what changed inside methods"*. In other words, this developer would like to see, on demand, the source code lines modified in each changed code entity. While this information is not present in ARENA's release note, it would be rather easy to implement. All other developers answered with a *strongly agree* and one of them explained her score with the following motivation: *"I got a clear idea of what changed in SMOS 2.0. Also, I noticed as I was not aware about some of the reported changes"*.

All developers *strongly agreed* on the correctness of the information reported in the release note generated by ARENA (see Table 4): *"after checking in the source code I think all the reported information is correct."* Also, they *strongly disagreed* about the presence of redundancy in the information reported in the release note. In particular, one of them explained that *"information is well organized and the hierarchical view allows visualizing exactly what you want, with no redundancy."* Finally, all developers *weakly agreed* or *strongly agreed* on the usefulness of the ARENA release note for a developer in charge of evolving the software system, for example, *"the release note is very useful to get a quick idea of what changed in the system and why."* The only developer answering *weakly agree* commented: *"developers are certainly able to get a clear idea about what changed in the system. But they may still need to look in source code for details."* This developer was the one asking for a line of code granularity level.

In the second part of this study, we asked the same four developers to evaluate (by using the same questionnaire) the release note manually-generated by their team leader. Table 4 reports these results in parenthesis. In this case, three developers *weakly agreed* on the completeness of the release note, while one *weakly disagreed*. As comparison, on the completeness of the release note generated by ARENA three developers *strongly agreed* and one *weakly agreed*. The developer answering *weakly disagree* motivated her choice explaining that *"the level of granularity in this release note is much higher as compared to the previous one. Thus, it is difficult to get a clear idea of what changed in the system. Also, information about the updated libraries is missing"*. When talking about "the granularity" of the release note, the developer refers to the fact that changes are not always reported at method level as it happened in the ARENA release note. Three developers *strongly agreed* about the correctness of the information reported in the manually-generated release note, while one of them answered *weakly agree*, reporting an error present in the release note: *"the method* `daysBetweenDates` *has been deprecated, not deleted"*. Indeed, the manually-generated release note contained such a mistake, avoided by ARENA which reported: *Method daysBetween(Date,Date)*

**Table 4: Evaluation provided by four original developers to the release note generated by ARENA for SMOS. In parenthesis, the evaluation provided to the manually-generated release note.**

| Claim | Subject ID | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| The release note contains all the information needed to understand what changed between the old and the new release | 3 (2) | 4 (3) | 4 (3) | 4 (3) |
| All the information reported in the release note is correct | 4 (3) | 4 (4) | 4 (4) | 4 (4) |
| There is redundancy in the information reported in the release note | 1 (1) | 1 (1) | 1 (1) | 1 (1) |
| The release note is useful to a developer in charge of evolving the software system | 3 (3) | 4 (3) | 4 (3) | 4 (3) |
| 1=strongly disagree, 2=weakly disagree, 3=weakly agree, 4=strongly agree | | | | |

*in Utility has been deprecated*. Note that the error was rather subtle, as only one of the developers was able to spot it.

**Summary of Study III.** The SMOS developers judged the ARENA release note as more complete and precise than the one created by the team leader ($\mathbf{RQ}_1$). Moreover, the extra information included in the generated release note makes it to be considered more useful than the manual one ($\mathbf{RQ}_2$).

# 5. THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. In this work such threats mainly concern how the generated release notes were evaluated. For *Study I* and *Study II*, we tried to limit the subjectiveness in the answers by asking respondents to compare the contents of the generated release note with that of the actual one. In *Study I* we assigned each release note to two independent evaluators. For *Study III*, although our main aim was to collect qualitative insights, we still tried to collect objective information by (i) involving multiple evaluators, and (ii) using an appropriate questionnaire with answers in a Likert scale, complemented with additional comments.

Threats to *internal validity* concern factors that could have influenced our results. In *Study I* and *Study III* we have tried to limit the evaluators' bias by not telling them upfront which were the original and automatically generated release notes. Another possible threat is that the participants in *Study I* and *Study II* have different level of knowledge of the object projects. We must note that some of the respondents in *Study II* were developers of the object projects and their answers were in line with the answers of the other participants. None of the participants in *Study I* were developers/contributors of the object projects. In *Study III* we asked developers to perform the comparative evaluation only after having provided a first assessment of the automatically generated release note. This allowed us to gain both an absolute and a relative assessment.

Threats to *external validity* concern the generalization of our findings. In terms of *evaluators*, the paper reports results concerning the evaluation of release notes from the perspective of potential end-users/integrators (*Study I* and *Study II*) and of developers/maintainers (*Study II* and *Study III*). In terms of *objects*, across all studies, release notes from 11 different releases were generated and evaluated.

# 6. RELATED WORK

To the best of our knowledge, no previous work has focused on automatically extracting and describing the system-level changes occurring between two subsequent versions of a software system. Some research has been conducted to summarize changes occurring at a smaller granularity. Buse and Weimer proposed *DeltaDoc* [20] a technique to generate a human-readable text describing the behavioral changes caused by modifications to the body of a method. Such an approach, however, does not capture why the changes were performed. In this sense, Rastkar and Murphy [21] proposed a machine learning-based technique to extract the content related to a code change from a set of documents (*e.g.,* bug reports or commit messages). Differently from our approach, these summaries focus on describing fine-grained changes at method level. ARENA is meant to summarize sets of structural changes at system level and, where possible, relate them to their motivation. The automatic generation of release notes relies on extracting change information from software repositories and issue tracking systems. More research work has been done in this area, especially in the context of software evolution [22]. Related to our approach is the work on traceability link recovery between the issues reported in issue trackers and changes in versioning systems [17, 23, 24, 25]. While ARENA employs similar techniques to extract change information, none of these approaches attempted to produce a natural language description of the code changes linked to the reports.

Concerning summarization of other software artifacts, different approaches have been proposed to automatically summarize bug reports [26, 27, 28]. The focus of such approaches is on identifying and extracting the most relevant sentences of bug reports by using supervised learning techniques [26], network analysis [27, 28], and information retrieval [28]. These summaries are meant to reduce the content of bug reports. In a different way, the bug report summaries included in the ARENA release notes are meant to describe the changes occurred in the code of the software systems in response to such reports. At source code level, the automatic summarization research has focused on OO artifacts, such as, classes and methods [1, 2, 29]. ARENA uses the approach proposed by Moreno *et al.* [2] for generating descriptions of the added classes in the new version of the software.

# 7. CONCLUSION AND FUTURE WORK

We introduced ARENA, a technique that combines in a unique way source code analysis and summarization techniques with information mined from software repositories to automatically generate complete release notes. Three empirical studies were aimed at answering two research questions and concluded that: (i) the ARENA release notes provide important content that is not explicit or is missing in original release notes, as considered by professional and open-source developers; (ii) the ARENA release notes include more complete and precise information than the original ones; and (iii) the extra information included by ARENA makes its release notes to be considered more useful. Based on this work, the research on release note generation moves into a distinct arena, where new research questions can be investigated, such as: *What is the most important information to include in the release notes and how to classify it? How should release notes be presented to users?* These are just two items on our research agenda.

# 8. REFERENCES


[1] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52.

[2] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proceedings of the IEEE International Conference on Program Comprehension*, ser. ICPC '13. IEEE, 2013, pp. 23–32.

[3] R. Koschke, "Atomic architectural component recovery for program understanding and evolution," in *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*. IEEE Computer Society, 2002, pp. 478–481.

[4] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[5] A. Hassan and R. Holt, "Architecture recovery of web applications," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 2002, pp. 349–359.

[6] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-based lightweight C++ fact extractor," in *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 134–143.

[7] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September 2004, Kyoto, Japan*. IEEE Computer Society, 2004, pp. 31–40.

[8] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[9] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *Proceedings of the IEEE International Conference on Program Comprehension, Formal Tool Demonstration*, ser. ICPC '13. IEEE, 2013, pp. 230–232.

[10] D. M. Germán, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010, pp. 437–446.

[11] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 375–384.

[12] J. Ratzinger, T. Sigmund, and H. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008, Leipzig, Germany, May 10-11, 2008*. ACM, 2008, pp. 35–38.

[13] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: an empirical study," *Empirical Software Engineering*, 2012.

[14] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10.

[15] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, 2008, p. 23.

[16] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 392–401.

[17] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 2003, pp. 23–.

[18] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 15–25.

[19] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[20] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 33–42.

[21] S. Rastkar and G. C. Murphy, "Why did this code change?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1193–1196.

[22] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 77–131, Mar. 2007.

[23] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proceedings of 21st IEEE International Conference on Software Maintenance*. Budapest,

Hungary: IEEE CS Press, 2005, pp. 525–535.

[24] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 97–106.

[25] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE '11. New York, NY, USA: ACM, 2011, pp. 31–37.

[26] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 505–514.

[27] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Riva del Garda, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 430–439.

[28] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 11:1–11:11.

[29] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of 17th IEEE Working Conference on Reverese Engineering*. Beverly, MA: IEEE CS Press, 2010, pp. 35–44.