What Are the Features of This Software? An Exploratory Study

Barbara Paech, Paul Hübner

University of Heidelberg
Institute of Computer Science
Heidelberg, Germany
{paech,huebner}@informatik.uni-heidelberg.de

Thorsten Merten

Bonn-Rhein-Sieg University of Applied Sciences
Dept. of Computer Science
Sankt Augustin, Germany
thorsten.merten@h-brs.de

***Abstract*—Application systems are often advertised with features, and features are used heavily for requirements management. However, often software manufacturers only have incomplete information about the features of their software. The information is distributed over different sources, such as requirements documents, issue trackers, user manuals, and code. In this paper, we research the occurrence of feature information in open source software engineering data. We report on a case study with three open source systems. We analyze what information about features can be found in issue trackers and user documentation. Furthermore, we study the abstraction levels on which the features are described, how feature information is related, and we discuss the possibility to discover such information semi-automatically. To mirror the diversity of software development contexts, we choose open source systems, which are quite different, e.g., in the rigor of issue tracker usage. The results differ accordingly. One main result is that the user documentation did not provide more accurate information than the issue tracker compared to a provided feature list. The results also give hints on how the management of feature relevant information can be supported.**

***Index Terms*—feature; requirements management; mininig software repositories; issue tracker; user documentation**

I. Introduction

In requirements management, features of application software are heavily used to package requirements. At least for the following three purposes: release planning in software product management [1], software product line engineering [2] and requirements feature interaction detection [3]. The corresponding approaches typically assume a dedicated feature and requirements representation. However, in industry features often are managed implicitly. Typically, they are used within a project to develop a part of a software product, but they are not collected in a dedicated document and maintained over time. The paper by Alspaugh and Scacchi shows that open source software (OSS) development projects typically do not have an explicit requirements or feature description and stipulates that this might also be true for many commercial software development projects [4]. In our work, we reported about feature knowledge being implicit in answers to requests for proposals [5]. We and others have reported on a heterogeneous requirements pool being the basis for release planning in industry [1], [6], [7]. Thus, in order to get a feature view of a software product it is often necessary to detect the features from data sources other than requirements or feature documents. Three feature-related information sources are typically available in software projects in industry:

- Bugs and feature requests in an issue tracker
- User documentation
- Code in a version control system

For feature location in code, typically the existence of documentation about features is assumed [8]. As we are interested in deriving the features, we focus on issue trackers (ITS) and user documentation (UD). UD has already been recommended as a substitute for a requirements specification by Dan Berry et al. in [9]. ITS are a well-known source for features, as often issues are explicitly tagged as features. However, feature tagging is not always reliable as has been shown by Herzig et al. in [10]. For example, they found that only 40% to 72% of Bugzilla issues are correctly classified as feature requests and many issues classified as bugs or improvements do actually contain feature requests.

Thus, it is necessary to analyze in more detail what information about features can be found in these sources. Although the Mining Software Repositories[1] community does some work about categorizing features and bug reports, there is no work identifying the individual feature descriptions in the ITS or UD data. The long term-goal of our research is to develop an approach to semi-automatically derive a feature representation from these data sources. As a first step, we present an explorative study analyzing the feature information of three different open source systems. The goal is to explore the kind and quality of feature information in ITS and UD.

The rest of the paper is structured as follows. Section II presents the planning and operation of the case study. Section III presents the results of the study. Related work is discussed in Section IV and an overall summary and outlook on future work is given in Section V.

[1] http://msrconf.org.

 ISBN: 978-1-61208-367-4

II. Case Study Planning and Operation

In this section, we describe the definition and planning of our study, the operation and the threats to validity.

A. Study Definition and Planning

We applied case study research, as this is an exploratory study trying to understand a real-world phenomenon [11]. The main research question is:

What information about software features can be found in the user documentation and issue tracker of a software product, and how well is this suited to derive a feature representation of the software?

This question is detailed into the following research questions:

- RQ1: What feature information can manually be derived from the issue tracker and the user documentation?
- RQ2: What are the commonalities and differences of feature information from UD and ITS and how well does the information fit to the feature list provided by the developers themselves?
- RQ3: How easily could this information be derived semi-automatically?

The study was conducted on open source project data, since it is most easily available. Based on prior experience, the projects were selected so that their combination fulfills the following criteria (see Table I):

- Availability of ITS and UD and of an explicit feature list compiled by the software developers themselves
- Different domains
- Different size of product and ITS and UD data
- Different user groups. We looked for projects with private users and/or professional users.
- Different kinds of ITS. Radiant uses a lightweight ITS (GitHub) with simple tagging possibilities to categorize issues. OFBiz uses an industry standard ITS (Jira) which supports multiple categorization options and status. Mixx uses a heavyweight ITS (LaunchPad) which additionally provides the option of connecting blueprints and user questions to the issues.
- Diversity of the quality of the provided information in the ITS. E.g., Mixxx uses the ITS systematically, whereas Radiant uses it ad-hoc.
- Different completeness of our analysis. The large projects could only be analyzed partially, but are more representative for the situation in industry.

TABLE I. Selection of OSS Projects

	Mixxx	OFBiz	Radiant
Domain	DJ software	ERP	CMS
Size	large	large	small
User group	private	professional	private and professional
ITS	LaunchPad	Jira	GITHub
Use of ITS	systematic	systematic	ad-hoc

B. Study Projects

This section provides characteristics of the three projects utilized for the study. Mixxx is a disk jockey software which implements basic features for managing and playing music and advanced features like a virtual mixer to perform seamless transitions between songs. Radiant is a content management system which implements basic features to create websites or blogs and advanced features like RSS feeds and an extension system to add 3rd party functionality. OFBiz is an enterprise automation software, where we studied the manufacturing resources planning component.

Table II provides further details on the projects. For OFBiz, only the manufacturing component and one corresponding provided feature was studied. The LOC of Mixxx comprise only the C++ code (excl. blanks and comments and XML configuration files). The LOC of Radiant comprise only Ruby and (r)html code (excl. blanks and comments). For Mixxx all blueprints and randomly sampled issues (to identify the quality of links between issues and blueprints) were analyzed, for Radiant all issues. In Radiant the status "implemented" was only identified for the feature-relevant issues.

TABLE II. Project Details

	Mixxx	**OFBiz**	**Radiant**
# features in list	22	1	10
Size (LOC)	94117	Not det.	33887
Programming language	C++ (& QT)	Java	Ruby (& Rails)
# issues	2211 + 113 blueprints + 138 user questions	120	348
# issues implemented	1239 + 59 blueprints	94	See text
# issues analyzed	50 + 113 blueprints	all	all
# analyzed issues with feature information	22 + 53 blueprints	19	50
# issues implemented and analyzed with feature information	22 + 53 blueprints	16	43
# subdivisions UD	14 chapters consisting of 69 sections	343	120 pages
# subdivisions UD analyzed	all	36	all
# subdivisions with feature information	62	34	64
# provided features identified in ITS	21	7	12
# provided features identified in UD	24	12	12

C. Study Operation

This section provides a short overview of the indicators we used for feature relevant information and describes how we searched the ITS and UD of the projects. We analyzed the data sources in February 2014. Moreover, we stored all analyzed data locally for a reliable reproduction of our results.

1) Feature indicators: For the ITS we looked for issues which describe a new functionality (F) or quality (Q). The feature has to be already implemented and the issue mentions F and Q or a component of F and Q. It was not always

 ISBN: 978-1-61208-367-4

easy to determine the implementation status of an issue. For Radiant the status was not managed explicitly. Thus, the implementation status was revealed by analyzing the comments of an issue and associated commits. For Mixxx and OFBiz we took the issues with the status "Implemented or Patch Available". We did not find any indication that the status for those issues was set wrongly. However, there might be issues which are implemented but the status is not set accordingly. For the UD we looked at section and page titles containing this kind of information and not only describing the operation of the product. The exact rules and corresponding examples are shown in Table III. We classified the feature information

TABLE III. INDICATORS OF FEATURE RELEVANT INFORMATION

ITS	UD
The issue is implemented AND mentions functionality or quality AND is not related to a bug AND is not only related to refactoring AND the term X or a component X_i of X is explicitly or implicitly mentioned.	*The item describes functionality or quality X and not operation (such as installing or getting help) AND the term X or a component of X is explicitly or implicitly mentioned.*
Radiant (Quality Performance, Component "Radius Parser" of "Radius Template Language"): **"Speed up Radius parser"**	Radiant Page Titles (Quality Performance and Caching) **"Disable caching in a radiant system"**
Radiant (Functionality Asset Management): **"Integrate an asset management solution"**	Radiant Page Title (Functionality Admin UI) **"Altering Tabs in the Admin UI"**
Radiant Implementation Status by comment: **"Seeing as there's a setting for this now, this issue can be closed?"**	Mixxx (Quality was not mentioned)
Mixxx (Quality User Experience, Functionality Vinyl Control): **"Improvements to the overall vinyl control user experience"**	Mixxx Section (Functionality Broadcast): **"Live Broadcasting Preferences"**
Mixxx (Functionality, Component Crates and Playlist): **"Currently Mixxx does not support hierarchies for crates and playlists. This, however, is possible"**	OFBiz (Functionality Routing Task): **"Find Routing Task"**
OFBiz: (Functionality Production Machines) **"cover the case in which many machines are used to complete a production task"**	

according to their abstraction levels. It is well-known that requirements and features are typically described on different abstraction levels. Based on the work of Gorschek et al. [7], we distinguish 3 levels of features:

- Requirements level (called feature level in [7]): the mentioned F comprises several functions or the Q affects several functions
- Function level: F or Q only refer to one function which a user can perform. Implementation details are not mentioned.
- Code level (similar to the component level used in [7], it focuses on the HOW): F or Q only refer to one function which a user can perform. Implementation details are mentioned. For UD the levels were easy to identify. Page or section titles referred generally to requirements, while subpages and subsections referred to functions. Code details were only mentioned in the UD of Radiant, as here the user is required to change classes to setup a certain functionality. Table IV shows examples for issue texts on different abstraction levels.

TABLE IV. EXAMPLES FOR ISSUE ABSTRACTION LEVELS

	Function	Quality
Requirements	**Radiant:** *"Break Radiant into several different extensions"* **OFBiz:** cf. Table III example bottom left.	**Radiant:** Internationalization
Function	**Radiant:** *"Errors when changing your password should be shown"* **Mixxx:** *"Implementation of a traktor library feature to allow professional DJs the smooth migration [...]."* **OFBiz:** *"Improve mrp to support to products which have no orders against them"*	**Radiant:** *"Make it so that pages are only cached for GETs"* **Mixxx:** *"Smooth Waveforms" (relates to a less stuttering display for track visualization).* **OFBiz:** *"There is a need to be able to block viewing info except that info that may pertain to that login"*
Code	**Radiant:** *"Javascript to stop you from navigating away from a page with changes"* **Mixxx:** *"It would be nice to be able to specify multiple <option>s for MIDI controls in XML mapping files."* **OFBiz:** *[...] accepts the partyId as a parameter, but has been commented [...] [however, the] functionality is vital for determining which employees are responsible for rejects*	**Radiant:** *"[Add] Ruby 1.9.x compatibility"* **Mixxx:** *"Distribute Launchpad translations with Mixxx Releases"*

In [9], Berry et al. distinguish typical section types of an UD: the abstractions (objects) of the domain (O) and the use cases (U). We use this distinction to classify the focus of a text. O is used when the feature is directly part of the UI or the software, while U is used when the feature requires some kind of dialogue to be used. U is not applicable to quality features. We also identified relationships between features. They are

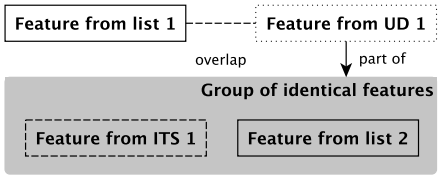


Fig. 1. Legend for the Feature Graphs.

used particularly in visualizations, such as Figure 2 and 4. Based on the information available to us, we determined the following relationships between the identified features (see Figure 1 for a legend of the relationships).

- Identical (features of different sources)

- Part of (features of different and same sources)
- Overlapping (features of different sources)

2) Procedure: The second and the third author conducted the actual search, while the first author acted as a reviewer. RQ1 was answered using thematic coding [12]. The second author coded the UD pages and sections and derived a set of codes characterizing the features described. Similarly, the third author coded issues in the ITS. The identified two feature sets were compared with the provided feature list on the project website and with each other. RQ2 was answered by comparing the different feature sets. The answer of RQ3 is based on the experience of the two authors during the manual derivation of the feature information.

D. Threats to Validity

We discuss the threats to validity according to Runeson et al. [13]: *Construct validity:* The authors have not been involved in the development of the sources. Thus, our view of what constitutes a feature of the software is clearly an external one, which might be different from what developers consider a feature of their software. To mitigate this threat, we used the feature list provided by the developers for comparison. *External validity:* The results are not representative for application software in general, as we only looked at three projects. For this exploratory study, we choose very different projects to enlarge the possible insights. *Reliability:* As only one researcher coded the ITS and the UD information, we cannot claim that other researchers would reproduce the coding. However, we used very explicit coding indicators and discussed them explicitly to minimize the bias of the individual coder. Moreover, the performed approach can be adapted to any software development project which provides the required data (ITS, UD and feature List).

III. Results

In the following, we answer the research questions for each project individually. The last subsection summarizes the insights for all projects.

A. Results of Project Radiant CMS

1) Provided feature list: The Radiant website contains a feature overview[2] which depicts 10 features using a name and a short one- or two-sentence description. As these features are listed prominently, we stipulate that they are the most marketing relevant for the developers. Table V shows these features and our classification as F or Q and O or U. The table also shows whether the feature was identified in the ITS or UD. Brackets indicate that the corresponding ITS or UD features are slightly different (see below).

[2]http://radiantcms.org/overview, accessed on August 8, 2014

TABLE V. Radianat Features

Provided feature list	Identified in
Built with Ruby on Rails (Q,O)	ITS
Custom Text Filters (F,U)	-
Flexible Site Structure (Q,O)	-
Intelligent Page Caching (Q,O)	ITS, UD
Layouts (F,O)	(UD)
Licensed under the MIT License (Q,O)	-
Pages (F,O)	ITS
Radius Template Language* (F,O)	ITS, UD
Simple Admin Interface (F,U)	ITS, UD
Snippets (F,O)	(ITS, UD)

* a special macro language (similar to HTML and Ruby).

2) Identification of feature information from UD and ITS (RQ1): The UD is organized in a wiki. The starting page of this wiki is a global table of contents. This table of contents is divided into 11 chapters, 8 of which only deal with administrative issues.

Thus, we identified the three chapters “The Basics”, “How Tos” and “Extensions” as primarily relevant for further analysis. “The Basics” contains seven links to top level UD pages. Except for the links to “FAQs” and “Getting Started”, the links point to pages describing Radiant features as mentioned in the feature list (Pages, Layouts, Snippets, Radius Tags, Customizing the Admin UI). In addition, there are six links to details of the Radius Tag feature and two links to details of the admin UI feature. Each top level UD page contains the intent and summary of the feature, screenshot of the features UI, and detailed descriptions of the feature use.

The “How Tos” chapter contains 29 links to top level UD pages. As visible by the titles, those links point to tutorials describing advanced features. The tutorials include usage examples and reference the basic feature pages. The Radius Template Language is referenced from almost all pages. The “Extensions” chapter starts with 6 pages describing the concept and usage of radiant extensions, followed by a list of 27 common extensions, and 11 pages which describe how to develop an extension for Radiant. According to the indicators of Table III, we identified 64 relevant pages.

The boxes marked with *UD* on the right side in Figure 2 shows the 13 features identified from the UD. *Content delivery* refers to different channels like RSS, *content location* to search in a web page. Most pages are on the function level and many describe layout. The number of pages related to a feature do not signify the importance of that feature. The features listed under “The Basics” can be seen as most essential, however, they are described on 15 pages, only.

The Radiant project uses GitHub as ITS. It is used for different aspects, such as Feature Requests, Bug Reports, Discussions of the development process, Discussions about refactorings and sometimes User Problems and Discussions about Documentation.

GitHub provides optional labels to classify an issue. Since the labels are optional, they are rarely used in the Radiant project. This implies that issues related to features, bugs, or

 ISBN: 978-1-61208-367-4

other aspects of Software Engineering (SWE) are not labeled accordingly by any means.

Therefore, we analyzed each of the 348 issues manually and derived their category (feature, bug, refactoring, other SWE aspects) by analyzing the descriptions and comments.

The boxes marked with *ITS* on the left side in Figure 2 refer to the 11 features identified from the ITS. *Asset management* refers to content different from the pages such as image files. *Development* comprises support for developers, such as a framework, *Frontend* refers to usability features. The issues mostly deal with individual functions, half of them deal with code (cf. Figure 3a). Many issues deal with the *Simple Admin UI*. Here again, the number of issues does not signify the importance of the feature. Furthermore, there are issues with many comments, but, e.g., a very short implementation.

3) Commonalities and differences of UD and ITS and provided features (RQ2): Figure 2 shows the relationships between the identified features. As could be expected, the description of the features in the ITS is quite often on the code level, while the UD features are described on all three levels. Almost a third is on the code level which is unusual for an UD. This is due to the fact that code needs to be changed for some functionalities. However, only 2 features are solely described on the code level. Figure 3 shows that the feature sets have

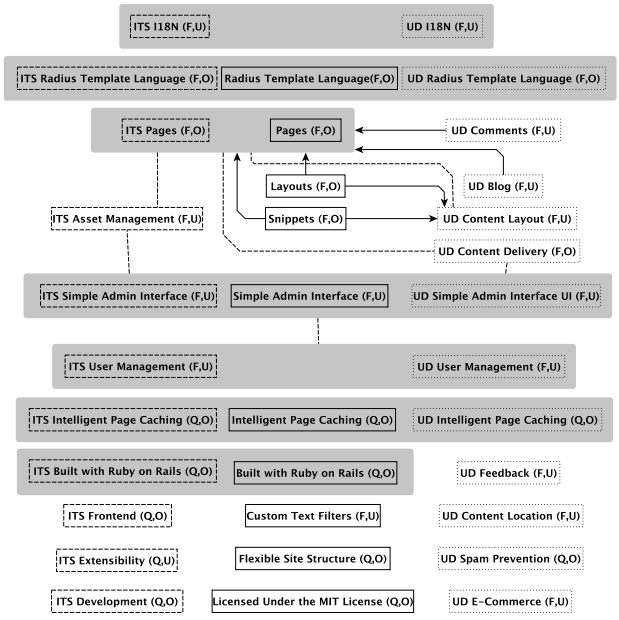
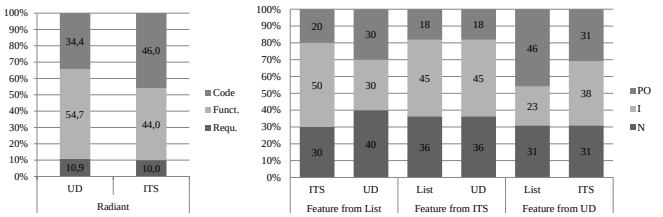


Fig. 2. Radiant Feature Graph (transitive relationships are not shown)

some commonalities, but also differences. Almost half of the ITS features (45%) are identical to the provided features, while only a third of the UD features (31%) is identical (cf. Figure 3b). This might be due to coder differences, but also due to the fact that the UD already provides a structure indicating low-level features which are not mentioned on the marketing level. Issues mention the features without any structure. Thus, the developer and the coder are missing a structure when referring to low-level features. Provided features not identified



(a) Abstraction Level (b) Feature Relationships (N = Not Related, I = Identical, PO = Partial Or Overlapping)

Fig. 3. Radiant Commonalities and Differences of UD, ITS and Feature List

from the ITS (18%) may be due to the fact that the ITS was not used from the beginning of the development. The basic functionality of the software was implemented before the ITS was used. For ITS features not in the provided list (31%), the content could be a reason. While Internalization and Extensibility seem relevant as prominent features, Fronted and Development issues might be too low-level. It is interesting to note that all ITS features which do not have a relation to either the list or the UD are quality-related. All features identified from the UD seem relevant, although 31% of them are not in the provided list. There is no pattern wrt F/Q or O/U in the differences between UD and the other feature sets. The PO-relationships between Pages, Layouts and more fine-grained features, such as Blog or Comments, show that granularity is a challenge. Only UD features have Part-of-relationships. Again, this can be due to the more fine-grained structure of the UD. UD features are more closely related to ITS feature (45% identical) than the provided list, but there are almost as many (36%) non-related features.

4) Automatic identification of feature information (RQ3): Most feature-related information was identified in older issues. 34 feature-related information items could be found in issues #1 to #68. 16 feature information items in #71 to #202 and no feature-related information was found in #203-#384. This suggests that a) older issues should be available for automatic feature extraction and b) it might be best if the ITS is used from the beginning of the development (e.g., design and prototyping phases). In Radiant however, the ITS was only used after a prototype of the software had already been built.

For the ITS we searched for text containing '*should*be*', '*add*, '*would*be*' and '*allow*user*' in the issue title and description. This revealed 27 of the 43 issues with feature related information. However, the search added about the same amount of noise and included refactoring- and bug-related issues. Therefore, the precision of this approach is relatively low. However, depending on the usage of an ITS, it might be possible to extract more precise search terms. A pitfall in automatic analysis are the tags of the Radiant ITS. As in [10], manual categorizations are often wrong. Although tags like *bug*, *design*, or *javascript* are introduced in Radiant, they are not used consistently. Since tags are optional, we found that most issues are not tagged at all. The *bug* tag is only used for

 ISBN: 978-1-61208-367-4

one issue, which is not reliable for any automatic extraction. Further analysis, for example topic analysis [14], is necessary to identify feature labels.

For the automatic identification of features from the UD, the relevant pages need to be identified in a first step. For Radiant the relevant pages were mainly contained in three chapters, which can be identified more efficiently manually than automatically. Additionally, some pages are only related to system operation and do not contain feature information. To some extent, these pages could be identified by searching for operation-related terms such as installation in order to discard these pages. Another input for the identification of relevant pages is the linkage structure. Based on this, it is possible to identify frequently referenced pages which most likely are feature-relevant pages.

The identification of the feature labels could start from the page titles. The nouns contained in the titles can be used as a starting point to create a feature list.

B. Results of Project Mixxx

TABLE VI. RADIANT FEATURES

Provided features	Identified in
Advanced Controls (F,U), Dual Decks (F,O)	IST,UD
Decks: Beat Looping. Broad Format Support, Hotcues, Intuitive Pitchbends, EQ and Crossfader Control, Time Stretch and Vinyl Emulation (F,O)	(ITS, UD)
Designer Skins (F,O)	ITS
Free Timecode Vinyl Control (F,U)	ITS,UD
Microphone Input (F,U)	ITS,UD
MIDI Controller Support (F,U)	(ITS,UD)
Powerful Library: Auto DJ, BPM Detection and Sync, Crates and Playlists, Disk Browsing, iTunes Integration (F,U)	ITS,UD
Quad Sampler Decks (F,O)	UD
Recording (F,U)	ITS,UD
ReplayGain Normalization (F,U)	-
Shoutcast Broadcasting (F,U)	(ITS,UD)

1) Provided feature list: Similarly to Radiant, the list of provided features was taken from a website [3] with short marketing descriptions. The feature list contained 20 functional features (see Table VI). Features which are part of a more general feature (e.g. library or deck) are listed in one row.

2) Identification of feature information from UD and ITS (RQ1): The UD is part of a general documentation WIKI which also contains developer documentation and documentation for special users, like artists. We focused on the user manual. The manual contains 14 chapters, 9 of which are feature relevant. These 9 chapters contain 69 sections. Since the chapters contain an introductive text, we assigned feature labels (abstraction level requirement) to 8 of them (one chapter title was "advanced features") and to the 54 sections which satisfied our indicators of TABLE III (abstraction level function). None were on the code level. The boxes marked with *UD* on the right side in Figure 4 show 20 of the

[3]http://mixxx.org, accessed on August 8, 2014

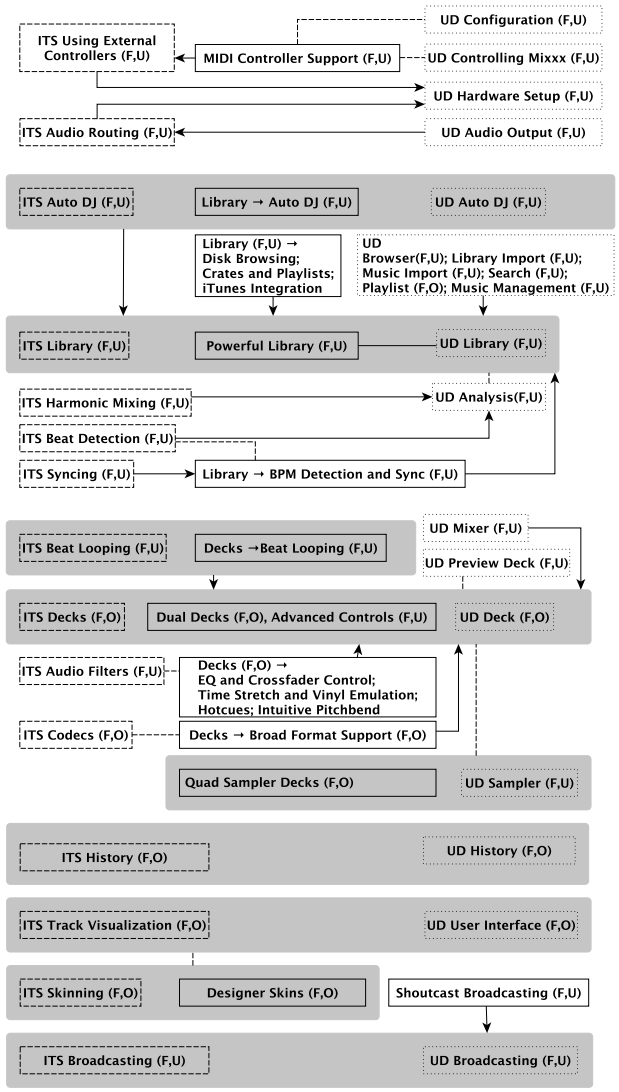


Fig. 4. Mixxx Feature Graph (identical, not related features, and transitive relationships are not shown)

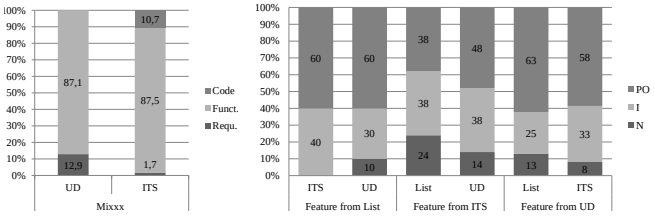
24 identified features and their classification (the features *DJing(F,U), Microphone (F,U), Recording (F,U)* and *Vinyl Control (F,U)* are not linked to the provided features and thus have been omitted in the Figure). The feature *Analysis* refers to the preparation of harmonic mixing, *Controlling Mixxx* allows setting device specific options, and *Vinyl control* allows to use records to control digital playback. All of the identified features describe a functionality. The number of sections or chapters corresponds to the complexity of the features. *Music Management* is the only feature described in detail (7 pages) which is not related directly to a chapter.

The Mixxx Project uses Launchpad as ITS. It contains 2211 issues (including bugs and feature requests), 113 so-called blueprints and 138 questions. The issue classification in bugs and features as made by the developers is very reliable for the issues we analyzed. The blueprints describe refactorings and higher level requirements for features. Blueprints and issues

 ISBN: 978-1-61208-367-4

are often linked, and issues are often linked with the code, but not always. Since blueprints contain more feature-relevant information than issues in the way Launchpad is used in the project, we analyzed all 113 blueprints for feature information. In our analysis, we included 59 blueprints with the status implemented.

The boxes marked with *ITS* on the left side in Figure 4 refer to 15 of the 21 features identified from the ITS (*Development (Q,O), Internationalization (F,O), Microphone Usage (F,U), Playback (F,U), Recording (F,U)* and emphVinyl Control (F,U) are not linked to the provided features and thus have been omitted in the Figure). All of the identified features describe functionality. *Beat Detection* analyzes the speed of a track. *Beat looping* repeats a short part of the track. *Codecs* are different digital formats. As for Radiant, *Development* describes support for the developers. *Skinning* refers to different UI looks. *Syncing* matches the speed of different songs for the mix. Only few blueprints are on the code level.



(a) Abstraction Level (b) Feature Relationships (N = Not Related, I = Identical, PO = Partial or Overlapping)

Fig. 5. Mixxx Commonalities and Differences of UD, ITS, and Feature List

3) Commonalities and differences of UD and ITS and provided features (RQ2): Figure 4 and 5 illustrate the commonalities and differences of the UD, ITS, and provided feature sets. The graph in Figure 4 does not show the identical features AutoDJ, Microphone Input, Recording and Vinyl Control, as well as the 3 features of ITS (Development, Internationalization, Playback) and one feature each of UD (DJing) and List (Replay) which are not related to other features.

There are fewer provided features (a tenth) compared to Radiant (roughly a third) which have not been mentioned by UD or ITS. This was expected from the fact that the ITS blueprints and the UD seem to be well maintained. Similar to Radiant, more features from the provided list were directly identified from the ITS (40%) compared to the 30% of the provided features identified from the UD. However, the difference is smaller. Furthermore, there are more ITS features (24%) which are not related to provided features. Two of them are related UD features. There are many identical features between ITS and UD (between 30 and 40%) and very few non-related ones($< 10\%$). However, there are also many part-of-relationships between ITS features and either UD (48%) or the provided features (38%). And there are even more PO-relationships from UD features to either the ITS (58%) or the provided features (63%). This indicates that even when well-maintained, the granularity in the different sources is different. The distinction between F und Q is not relevant as no Q features were involved. Again, no pattern wrt O/U could be found.

4) Automatic identification of feature information (RQ3): As the Mixxx ITS is maintained very systematically, the possibilities to categorize the status (e.g. implemented, draft, in progress) as well as the issue (e.g. bug, wishlist, blueprint) could be used as indicators for feature relevant information. The identification of the feature labels remains a problem.

The Mixxx UD is a well-structured document separated into chapters and two section levels. 9 of the 14 chapters are feature-relevant, compared to 3 out of 11 for Radiant. Often the chapter and section titles directly contain feature-relevant terms. As for Radiant the relevant chapters can be identified just by manually looking at the chapter titles. For the identification of features on the requirement abstraction level, the chapter titles can be used. The section titles on the first section level can be used for feature identification on the function abstraction level.

C. Results of Project Apache OFBiz

1) Provided feature list: The project OFBiz was studied only partially. As the project is very large a complete analysis was not feasible. The feature page[4] lists features on the requirements level. We decided to look at the manufacturing feature (one component) and the corresponding UD and issues, only.

2) Identification of feature information from UD and ITS(RQ1): The UD for OFBiz is organized as a wiki. However, the wiki only contains more or less empty pages and some basic structures (e.g. sections for role specific documentation, e.g., for managers). Based on this fact, we decided to use the outdated Manager Reference for our UD analysis (last updated in 2004, uploaded to the wiki as PDF attachment between 2006-12 and 2007-01). Based on the experience gained from the previously analyzed projects, we looked at the chapter and section headings, only. The document contains

TABLE VII. OFBiz Manufacturing Features from UD and ITS

Feature UD	Feature ITS
Bill of Materials (F,U)	Data Security (Q,O)
Bill of Mat. Simulation (F,U)	Internationalization (F,O)
Calendar (F,U)	Manage orders (F,U)
Job Shop (F,U)	Manage Products (F,U)
Manufact. Res. Planning (F,U)	Manage Production Machines (F,O)
Manufacturing Rules (F,U)	Manage Production Runs ()
Production Run (F,U)	Resource Planning (F,U)
Reports (F,O)	
Requirement Verification (F,U)	
Routing (F,U)	
Routing Task (F,U)	
Shipment Plans (F,U)	
Status Report (F,U)	

[4]http://OFBiz.apache.org, accessed on August 8, 2014

 ISBN: 978-1-61208-367-4

343 subdivisions organized into 4 hierarchy levels. We decide to remove the sections on the 3rd and 4th hierarchy level, since they only refer to single attributes, e.g. of specific input form values. 8 chapters and 28 sections remained. From this we could identify 13 distinct features (cf. Table VII, left column). 8 of the features are on the requirement abstraction level, based on the chapters. The other 5 features are mostly additional aspects of the requirement abstraction level features, i.e. they have been identified in subsections of the respective chapters. As expected for a manager reference, code details were not mentioned. Also, quality features were not mentioned and only one object of the domain.

The OFBiz Project uses Jira as ITS. It contains 5567 issues, 120 of these issues are related to the manufacturing component which was determined by filtering the ITS. We analyzed all 120 issues of the manufacturing component and identified 7 features (cf. Table VII right column). Security was the only quality aspect identified.

In OFBiz the issue feature descriptions are generally longer (in terms of words) as in the other projects. Requirements, for example are described in detail and often include multiple solution ideas (though not mentioning concrete code), e.g. **"[...] this can be implemented in many ways: a) expanding the concept of Fixed Asset groups [...] b) (more complex) add new association entities to link a task [...]"**. Although this suggests a very accurate handling of the ITS, we found multiple misclassifications of issues (e.g. bugs classified as improvements). In addition, some features were distributed over many issues. E.g. I18N included multiple issues for every single language and one main issue describing the feature and none of these were linked.

TABLE VIII. COMMONALITIES AND DIFFERENCES OF ITS AND UD

Feature ITS (7)	Feature UD (13)	Map
Data Security (Q,O), Internationalization (F,O)		
Manage Orders (F,U)	Bill of Materials (F,U), Bill of Materials Simulation (F,U), Calendar (F,U). Shipment Plans (F,U)	(O)
Manage Products (F,U), Manage Production Machines (F,O)	Manufacturing Rules (F,U)	(O)
Manage Production Runs (F,U)	Production Run (F,U)	I
Ressource Planning (F,U)	Manufacturing Resource Planing (F,U) Job Shop (F,U), Reports (F,O), Requ. Verification (F,U), Routing (F,U), Routing Task (F,U) Status Report (F,U)	I

3) Commonalities and differences of UD and ITS and provided features (RQ2): As the descriptions of the UD were coarse and we mainly looked at the headings, we did not derive a full mapping. Table VIII shows a rough mapping of the features of UD and ITS. Two features are identical and a few have overlaps. However, almost half of the UD features were not mentioned in the issues. This can be explained by the fact that the UD was outdated.

4) Automatic identification of feature information (RQ3): The OFBiz analysis did not reveal any new insights wrt automatic identification. For the UD we only used the chapter and section titles manually. Thus, this could be also the basis for an automatic identification. As for Radiant, most feature related information was identified in older issues. In the ITS, we found a feature ratio over all issues from 10 to 18% between 2006 and 2009 and only about 3 to 5% between 2009 and 2013.

D. Overall Results

1) Identification of feature information from UD and ITS (RQ1): ITS as well as UD can serve to identify features. The features, however, are described on different abstraction levels [7] (cf. Figure 6). For both, Mixxx, and OFBiz the UD does not contain features on code level and $> 75\%$ on function level. In contrast, the UD of Radiant mainly contains feature information on the code and function levels. In the ITS, feature information is found on all three levels, but, similar to the UD, there are only few features on the requirements level, and the distribution of abstraction levels in the ITS is quite different for each of the projects. Quality features are typically not

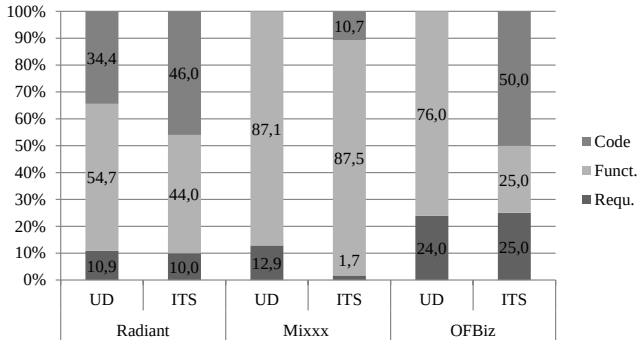


Fig. 6. Abstraction Levels

mentioned in the UD. Radiant and OFBiz UD's mention few quality features and Mixxx's UD none. Most quality features were found in the ITS.

2) Commonalities and differences of UD and ITS and provided features (RQ2): There is a noticeable overlap between the feature information in the ITS, the UD, and the provided feature list. In the Radiant project, roughly a third of the listed features could not be identified by UD or ITS, in the Mixxx project only a tenth could not be identified. Similarly, for Radiant only a third and for Mixxx only a tenth of the features was not related between ITS and UD. This indicates that, both ITS and UD could be used to record feature information systematically.

As the ITS is mainly important for the developers and the UD is targeted to the users, it seemed more likely that the UD better records the listed features [9]. However, it turned out that UD and ITS record the listed features equally well. It is interesting to note that in both projects 30-50% of the

 ISBN: 978-1-61208-367-4

features were identical (between List and UD, List and UD, and ITS and UD). This means that in the case of Mixxx there is a high percentage of overlapping features, while in the case of Radiant there are few overlapping features. Thus, even for a systematically documented project like Mixxx, a feature representation generated from UD or ITS would be different from the marketing feature list. Thus, different feature representations are likely needed for different purposes.

3) Automatic identification of feature information (RQ3): We have preliminary insights for automatic identification. It seems feasible to manually delimit the relevant pages of the UD and to focus on page or section titles to identify feature labels. Also for the ITS, there are first ideas to delimit the relevant issues, but the label identification will require more sophisticated techniques. As neither ITS nor UD were a perfect source, it seems likely that at least both sources must be searched and combined to yield a complete feature set.

Although best practices in RE suggest to describe new features as as-is and to-be situations, we found only one issue which mentions both situations. Generally, the to-be situation was described and the as-is situation was implicit. Furthermore, the quality and use of language, the quality of descriptions as well as categories and links differ from project to project. An automatic identification therefore needs to be "tuned" for each project.

From our experience, we see the following hints for using ITS and UD to record features instead of setting-up a separate feature documentation:

- The Mixxx project shows that feature information can be explicitly managed within an ITS, if it is separated from (but linked to) the usual stream of bugs and change requests. Furthermore, it seems likely that issues in ITS could profit from an abstraction classification or traces to more abstract information.
- Berry et al. recommend structuring an UD into objects, use cases and advanced features [9]. The UD of the projects have some similarities to this structure, but this could be improved. It might be helpful to structure the feature list accordingly.
- Blueprints and issues are much simpler to allocate to software components, because both have a technical nature. Without detailed knowledge of the software architecture, this is almost impossible for UD. Thus, if relationships between features and software components are important, ITS should be used as a source.

IV. Related Work

In this section, we discuss related work which derives feature information from diverse sources manually or semi-automatically.

Ghazarian identified generic classes of software functionality from 15 different requirement specifications in the domain of web-based enterprise systems. The identified classes such as data input or user interface navigation could potentially be useful as indicators of feature information [15]. They also describe that much feature-related information could be found and categorized analyzing only a small amount of issues, respectively only section and paragraph names in the UD. Their classification, however, is very technical and uses low abstraction levels. In contrast, our work classified features for different abstraction levels.

Noll et al. analyze an open source project to identify by whom and where requirements are proposed [16]. They select 13 given features and then trace them. In contrast we first looked at our data sources to manually identify features and then compared them with the given feature list. Thus, we gathered more data about how features are described in detail.

The following approaches use text mining to derive feature-like information from requirements specifications. Thus, their data sources are much more elaborate than the feature descriptions in ITS. Although the quality of UD and of the requirements specifications could be comparable, requirements specifications are rather structured than UD. Thus, results from both document sources are not quite comparable. These approaches can be used as a starting point for our future work on a semi-automatic feature-derivation approach. Gacitua et al. provide an approach for identifying abstractions from text documents which outperforms the usual information-retrieval methods [17]. In [18], Boutkova and Houdek describe an approach applied in industry to derive features from requirements documents based on a list of nouns. In the study it provided helpful input for experts.

Kuhn et al. recover topics from source code [19] and note that some extracted clusters represent software features. Since the features are extracted from source code, they are on the functional level (for example *handling text buffers*). In contrast, this paper is interested in feature descriptions on different abstraction levels.

V. Conclusion and Lessons Learned

The exploratory study of the OSS projects has shown commonalities and differences of feature information in UD and ITS and provided feature lists. The results are promising in the sense that ITS and UD provided relevant features. The results also show that deriving a complete feature set semi-automatically will be very difficult and will depend on the project.

In the projects we studied, the feature information within the project was consistent and some feature descriptions formed patterns (e.g. headings in the UD often denoted features). However, most of these patterns were not transferable to other projects.

During our research, we found that feature information is contained in only few issues of an ITS. Due to an ITS's nature, other issues like bugs are also tracked. Although many ITS provide the option to categorize issues manually as feature or bug, the quality of those manual categories depends largely on the project. Therefore, an analysis of the natural language is

 ISBN: 978-1-61208-367-4

also needed to identify all feature information. Furthermore, the feature information can be scattered all over the issue and can be found in title, description or even comments (although title and description are most common). Therefore, only a very small part of the natural language in an issue contains the feature description apart from rationale, solution ideas, social interaction and so on.

For UD, the starting point to detect and extract feature data semi-automatically is the structure of the respective documents. The analysis of the projects in this study showed that different structure levels in UD map to different feature abstraction levels. Our first experiments in the direction of using the UD headings and applying filter operation to remove common conjunctions seems promising. Moreover, certain UD parts, like administrative instructions, can be omitted for feature derivation since they do not contain feature-relevant data. To create a feature list from ITS and UD data automatically, a structured way of storing feature information would be necessary.

In future work, we will develop semi-automatic feature identification algorithms as well as guidelines for maintaining feature information in UD and ITS. They will be applied in industry to explore whether the identified features can be successfully used in release management.

ACKNOWLEDGMENT

This work is partly funded by the Bonn-Rhein-Sieg University Graduate Institute. We thank the Open Source community for the freely available data that was used in this research.

REFERENCES

[1] S. Fricker and S. Schumacher, "Release planning with feature trees: industrial case," in *Requirements Engineering: Foundation for Software Quality. 18th International Working Conference, REFSQ 2012*, vol. LNCS 7195. Essen, Germany: Springer Berlin Heidelberg, 2012, pp. 288–305.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feasibility Study Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. November, 1990.

[3] P. Shaker, J. M. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *2012 20th IEEE International Requirements Engineering Conference (RE)*. IEEE, Sep. 2012, pp. 151–160.

[4] T. a. Alspaugh and W. Scacchi, "Ongoing software development without classical requirements," in *2013 21st IEEE International Requirements Engineering Conference (RE)*. Rio de Janeiro, Brazil: Ieee, Jul. 2013, pp. 165–174.

[5] B. Paech, R. Heinrich, G. Zorn-Pauli, A. Jung, and S. Tadjiky, "Answering a Request for Proposal Challenges and Proposed Solutions," in *18th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 12)*, vol. LNCS 7195. Essen, Germany: Springer, 2012, pp. 16–29.

[6] G. Zorn-Pauli, B. Paech, T. Beck, and H. Karey, "Analyzing An Industrial Strategic Release Planning Process A Case Study At Roche Diagnostics," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, vol. LNCS 7830. Springer, 2013, pp. 269–284.

[7] T. Gorschek and C. Wohlin, "Requirements Abstraction Model," *Requirements Engineering Journal*, vol. 11, no. 1, pp. 79–101, Nov. 2006.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code. A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[9] D. M. Berry, K. Daudjee, I. Fainchtein, J. Dong, M. A. Nelson, T. Nelson, and L. Ou, "User s Manual as a Requirements Specification : Case Studies," *Requirements Engineering*, vol. 9, no. 1, pp. 67–82, 2004.

[10] K. Herzig, S. Just, and A. Zeller, "Its Not a Bug, Its a Feature: How Misclassification Impacts Bug Prediction," in *Proceedings of the 2013 International Conference on Software Engineering (ISCE)*. IEEE Press, 2013, pp. 392–401.

[11] R. K. Yin, *Case Study Research: Design and Methods*, 5th ed., ser. Applied Social Research Methods. SAGE Publications, Inc., 2013.

[12] C. Robson, *Real World Research*, 3rd ed. Wiley, 2011.

[13] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Dec. 2009.

[14] D. M. Blei, "Probabilistic topic models," *Communications of the ACM*, vol. 55, no. 4, p. 77, Apr. 2012.

[15] A. Ghazarian, "Characterization of functional software requirements space: The law of requirements taxonomic growth," in *2012 20th IEEE International Requirements Engineering Conference (RE)*. Chicago, Illinois, USA: IEEE, Sep. 2012, pp. 241–250.

[16] J. Noll and W.-M. Liu, "Requirements elicitation in open source software development: a case study," in *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development - FLOSS '10*. New York, New York, USA: ACM Press, 2010, pp. 35–40.

[17] R. Gacitua, P. Sawyer, and V. Gervasi, "Relevance-based abstraction identification: technique and evaluation," *Requirements Engineering*, vol. 16, no. 3, pp. 251–265, Jun. 2011.

[18] E. Boutkova and F. Houdek, "Semi-automatic identification of features in requirement specifications," in *2011 IEEE 19th International Requirements Engineering Conference*. Trento, Italy: Ieee, Aug. 2011, pp. 313–318.

[19] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

 ISBN: 978-1-61208-367-4