# Documenting and Exploiting Software Feature Knowledge through Tags

Marcus Seiler Barbara Paech
Institute for Computer Science, Heidelberg University, Germany
E-mail: {seiler|paech}@informatik.uni-heidelberg.de

## Abstract

*Knowledge about features and their relations to detailed requirements or code is important and useful for many software engineering activities such as for performing change impact analysis and tracking feature progress. Documenting feature knowledge is challenging, as companies document features and requirements in issue tracking systems (ITS) and work on code in integrated development environments (IDE). Managing feature knowledge over time is challenging, as features, requirements, and code continuously change. Also, managing the relationships through trace links is challenging, as creating links manually is too time-consuming, and recovering links retrospectively is too error-prone. We developed an approach and tool TAFT to document feature knowledge in ITS and IDE continuously. TAFT uses feature tags to indicate relations between feature descriptions, requirements, work items, and source code. Currently, TAFT comprises a dashboard to track the feature progress, a recommendation system to suggest feature tags for specifications, an inheritor to apply feature tags automatically, and capabilities to navigate in feature knowledge. The tool is integrated into the developers' work environments Jira and Eclipse. In this paper, we present details on the tool support for TAFT, and we report on the results of a case study, which indicates its acceptance.*

## 1. Introduction

Nowadays, many software-developing companies use an issue tracking system (ITS) to support software engineering work [2]. ITS contain several software engineering artifacts like requirements, development tasks (work items), or bug reports. An integrated development environment (IDE) is typically used to work on code that implement artifacts from ITS. Within ITS, requirements are often formulated as requests to modify or to add a specific feature. Since information regarding features is spread across the two sources ITS and IDE, it is hard to document and maintain features and their relations to code. Explicit knowledge about features and their relations to other artifacts, such as detailed requirements or code is not only useful for software evolution [14], but is also important for many software engineering activities including management tasks such as tracking the feature progress for release planning [9], and including development tasks, such as identifying affected artifacts when performing change impact analysis [11]. While the artifacts of feature knowledge are available in ITS and IDE, in practice their relationship is often managed implicitly or incompletely. This makes it difficult to exploit the feature knowledge for development and management tasks.

In previous work [19], we presented an interview study with practitioners on the use of tagging for feature knowledge and first ideas on a lightweight tagging approach to document feature knowledge. In this approach all artifacts relating to a feature are tagged with the feature. The experts from practice found our ideas beneficial. Documenting feature knowledge using tags seems easy at first glance. However, the experts indicated that providing suitable tags is not that easy and managing feature knowledge across different tools, and over time is a challenging task. The tag consistency over time is challenging, as the development of software systems is characterized by continuous change to features, requirements, and code [7]. In this paper, we present tool support to document, maintain, and exploit feature knowledge with tags in ITS and IDE consistently and efficiently. Currently, our tool support focuses on consistency across different tools and on efficiency to apply tags. The tool support comprises a dashboard to track the feature progress, a recommendation system to suggest feature tags for specifications, an inheritor to apply feature tags automatically, and capabilities to navigate in feature knowledge.

We conducted a case study with students in order to evaluate the acceptance of our approach and tool support according to the technology acceptance model (TAM) [6]. The results show that the students found our approach very useful for navigating to code parts during bug fixing. They rated our approach and tool support as easy to use and emphasized that the approach is intuitive to use. Also, they are motivated to use the approach and tool in future. The

DOI reference number: 10.18293/SEKE2019-109

students mentioned the following concerns: cumbersome initial set-up of the dashboard, missing support for viewing metrics from past sprints, and inaccurate recommendations.

The remaining paper is organized as follows: Section 2 introduces the terminology used throughout the paper and our *TAFT* approach. Section 3 describes the details on the tool support. Section 4 describes the case study with its research questions, hypotheses, and results. Section 5 discusses related work. Section 6 finally concludes this paper.

## 2. Background

This section briefly introduces the terms software features and feature knowledge. Then it describes our approach in more detail.

### 2.1. Software Features and Feature Knowledge

Various definitions of the term software feature exist in the literature [12, 4, 3]. We adopted the definition by Bosch [4] and define a feature in the context of this paper as *a functional or non-functional property of a software system.*

Different software engineering artifacts that are documented during specification and implementation relate to a feature. A feature description provides a general specification of the feature. Requirements refine the feature. Work items describe development tasks related to realizing the feature. Code implements (parts of) the feature. We therefore define feature knowledge as *knowledge comprising feature descriptions and all related software engineering artifacts such as requirements, work items, and code, as well as their relations.*

Figure 1 shows an example of feature knowledge from the studied project (cf. Section 4). As shown on the left hand side of Figure 1, a feature description was documented during specification and was refined by a requirement afterwards. The work item describes the implementation task for the requirement. Finally, the right hand side of Figure 1 shows the code implementing the functionality of the requirement.

### 2.2. The Feature Tagging Approach (*TAFT*)

The process of assigning keywords to artifacts is an effective approach to attach additional information to artifacts [20]. We developed a lightweight approach to manage feature knowledge across ITS and IDE [19]. Instead of creating traces between feature knowledge, feature knowledge is tagged with the same keyword. In particular, we use tags for feature descriptions, requirements, work items, and code. The *TAFT* approach works as follows: One tag for each feature of a software is used. The tag summarizes the feature in a short and concise manner. This tagging adheres to the following rules: First, a feature description is tagged with a feature tag if and only if it contains the description of the feature. Second, a requirement is tagged with a feature tag if and only if the requirement refines the feature. Third, a work item is tagged with a feature tag if and only if the described task addresses specification, quality assurance, or implementation of the feature. Finally, source code is tagged with a feature tag if and only if the source code implements (parts of) the feature.

In the example given in Figure 1, the tag `Transportation` is used to summarize the described feature. The feature tag is applied to the feature description, the requirement, and the work item in Figure 1 as they relate to the feature *Transportation*. The second statement of the code listing in Figure 1 shows the feature tag, as the code implements parts of the feature *Transportation*.

Our approach is independent of the development method used. Thus, *TAFT* is applicable for projects using traditional methods such as waterfall, and for projects using a modern development method such as agile. Moreover, we do not make any assumptions about the cardinality of the relations between features and requirements or between features and code. Thus, it is possible to have requirements, work items, and code tagged with multiple features unlike in the example.

## 3. Tool Support for *TAFT*

We developed tool support for *TAFT* in Jira[1] and Eclipse[2], which are common tools for managing software development projects and for working on code, respectively.

Our tool supports the two stakeholders developer and project manager in capturing feature knowledge through labels in Jira and annotations in code. It also supports them in exploiting feature knowledge for development tasks and for management tasks, respectively. We annotate the code with tags instead of tagging commits, as commit messages often contain noise in terms of tangled changes. Tangled changes could result in wrong feature knowledge when tagging a commit [8, 13]. Moreover, code annotations help to understand the code [21] and the cost for creating and maintaining annotations in code is negligible [10]. We rely on Java annotations instead of comment annotations. Unlike comment annotations, Java annotation are language specific, but they retain in the compiled byte code. This is useful for exploiting feature knowledge in (legacy) software even if the source code is not available (anymore).

In the following, we describe the details of the main tool functionality: the feature navigator, the feature dashboard, the feature recommendation and the feature inheritance. The Eclipse plug-in *Feature Navigator* analyzes the source code of a project and scans the code for annotations to support developers. Figure 2b shows a screenshot of the

[1] https://www.atlassian.com/software/jira

[2] https://www.eclipse.org/

**Feature:** Integration of an API to retrieve data from public transportation such as stations and their departure schedules.

**Requirement:** As a user, I want to click on a station in order to view the departure schedule.

**Work item:** Implement service function to retrieve the departure schedule of a station.

```
// Package, imports and further code omitted
@Feature("Transportation")
public class OpnvManager implements IOpnvManager {
   public void queryStation(String stationID) {
   Request request = new Request.Builder().url(new HttpUrl.Builder()
      .scheme("http").host("rnv.the-agent-factory.de")
      .addQueryParameter("stationID", stationID).build();
   new OkHttpClient().newCall(request);
   }
}
```

Figure 1: Example of feature knowledge documented in specifications and in code

*Feature Navigator*. The *Feature Navigator* lists code files implementing a certain feature. The developer can search for features and code files and can directly navigate to a code file once s/he clicked the code in the *Feature Navigator*.

The Jira dashboard *Feature Dashboard* supports project managers in tracking the progress of features in a project. Figure 2a shows the *Feature Dashboard*. The *Feature Dashboard* scans the project for feature tags and displays various metrics, e.g., the number of features, the number of requirements (in this case user stories), and the number of code lines implementing a feature. The metrics are calculated based on the labels applied to issues and the annotations applied to code. The dashboard can be configured for a project and the shown metrics can be selected. Multiple instances of the dashboard are possible to have metrics for multiple projects.

Developers and project managers might not document or update feature knowledge regularly, if the effort is too high. As suggested by Robillard et al. [15], we use recommendation systems to reduce the effort for the tool users. The completion of labels when typing parts of a feature tag is a built-in function of Jira. We extended Eclipse's code completion capabilities to complete annotations in code when typing parts of an annotation. Both, the label completion in Jira and the annotation completion in Eclipse represent simple recommendation systems. In addition, we developed a Jira plug-in to *recommend feature tags* for issues. Currently, we recommend existing feature tags based on the issue description using a multi-class Naive Bayes classifier. The feature tags are presented to the user together with the confidence score of the classifier. The user is then able to click on the feature tag to apply it to the actual issue. Figure 2c shows the recommended feature tags (`Transportation`, `RouteFinding`, `Filter`, `Tweets`) with their corresponding confidence scores on the right hand side of the Figure for a user story `Departure Schedule`. In this example, the best matching feature tag is `Transportation` with a confidence score of *74.71%* and is shown at the first position of the list. It is up to the user whether to apply one of the recommended feature tags. The classifier is trained iteratively whenever a user applies one of the recommended feature tags. In addition, we provide a feature tag inheritance plug-in for Jira to further reduce the effort to apply feature tags manually. The inheritance plug-in uses existing relations from Jira to automatically apply feature tags for issues in parent-child relations such as a user story consisting of several work items.

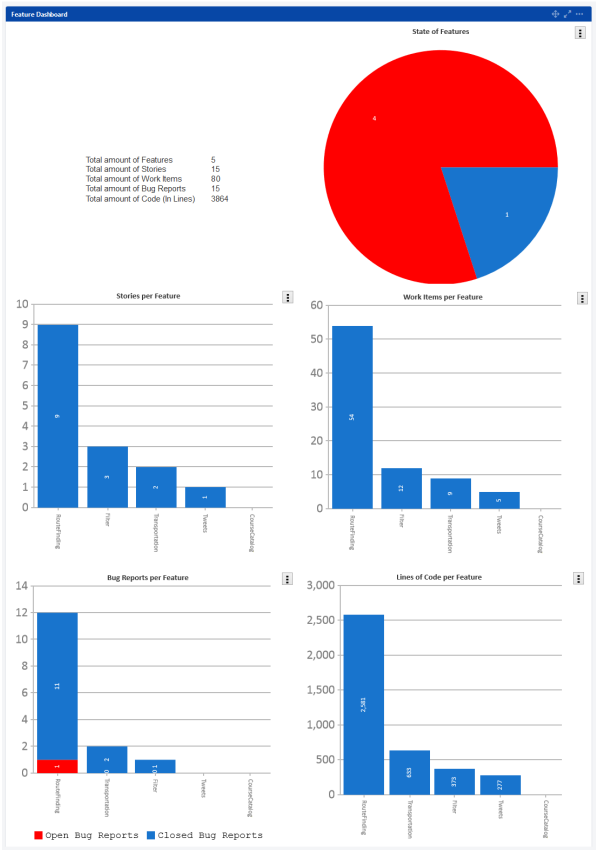
## 4. Evaluation of *TAFT*

We conducted a study with students in order to assess the acceptance of the *TAFT*. In the following, we describe the design of the case study and the applied research method in Section 4.1. Section 4.2 presents and discusses the results. Finally, Section 4.3 discusses threats to validity.

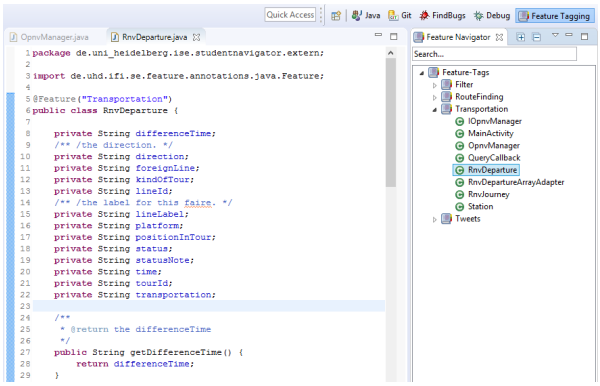
### 4.1. Case Study Design & Research Method

**Study Context:** The study was performed during a development project with six students over a period of six months. The project lasted from October 2017 to March 2018. The students developed an indoor navigation app for Android-based devices for a real customer. Primary users are (other) students who use the app to locate and navigate rooms where lectures take place. Also the app is able to retrieve information from public transportation allowing to display the departure schedule of a nearby station. The customer was a mobile development company. The development method was Scrum-like. In each sprint, one of the students acted as Scrum master and thus was responsible for development planning and communicating with the customer. The customer provided a high-level vision description of the app. The students derived features and refined them during development in agreement with the customer. They used Jira with epics to describe features and with user stories to refine features, and with work items to describe development tasks. The students used Git for Java source code and Eclipse as development environment. They applied our *TAFT* approach and used its tool support during the project. At the beginning of the project, the approach and the basic usage of the tool support were introduced. Also, the students were supported in the initial set-up of the tool support.

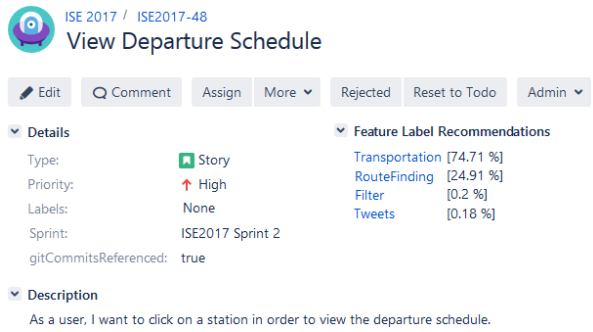
At the end, the project comprised five epics, 17 user stories, 74 work items, and 40 code files. According to our approach, the feature tags were applied to the user stories,




(a) Dashboard to track feature progress




(b) Navigator to find feature in code




(c) Recommendation to suggest feature

Figure 2: Tool support for the *TAFT* approach

the work items, and the code files. We conducted semi-structured interviews with the students to evaluate the acceptance. The questionnaire used during the interview contained open and closed questions.

**Research Questions, Metrics, and Hypotheses:** To evaluate the acceptance of our tool, we build upon the Technology Acceptance Model (TAM) by Davis et al. [6], which models the user acceptance of information technology. In our case, the information technology is the approach and the tool support. TAM uses the variables *perceived ease of use*, a *subjects' intention to use* and *perceived usefulness*. We raise the following research questions for acceptance evaluation:

$RQ_1$ How easy is it to use the approach and tool support?
$RQ_2$ How useful is the approach and tool support?
$RQ_3$ Do the students intend to use the approach and tool support in future?

Regarding the tool support, we are particularly interested in the recommendation and the dashboard. For usefulness, we are particularly interested in progress tracking and feature knowledge relations. We provided a questionnaire to the students with questions corresponding to the research questions.

According to Davis et al. [6] point scales such as Likert scales can be used to measure the variables of TAM. We used a Likert scale with five scale points for asking the students to assess the approach and tool support. The answers to the Likert scale were mapped to an integer as follows: strongly disagree = 1, disagree = 2, neutral (neither agree nor disagree) = 3, agree = 4, and strongly agree = 5.

We use the students' assessments together with their rationale for the ratings as metric for $RQ_1$, $RQ_2$, and $RQ_3$, respectively. Our hypothesis for acceptance is: We expect that the values for the TAM-variables are higher or equal to 3.5. Thus, we expect that most of the responses are in the range between neutral (with a slight tendency to agree) and strongly agree.

## 4.2. Results & Discussion

In the following, we use the answers to the open questions to provide the details for the assessments of the students.

The students stated that the approach is easy to use ($RQ_1$) as it is very intuitive to use. All students found it easy to apply feature tags to issues. A minority stated that applying feature tags in code is more complicated. The students needed more effort to equip some code files with feature tags, as these code files implemented multiple features. This is backed up by the numbers of the project. Each of the 17 issues had exactly one of the tags applied. Of the 40 code files, 37 had at least one of the tags applied. The majority (33 code files) contained one tag. One code file each contained two and three tags. Two code files contained four tags. Altogether, it seems easier to tag specifications instead of code files. The recommendation is rated easy to use and a little less useful compared to the usefulness to relate and track feature knowledge. Overall, the students stated that the recommendation works well and that the recommendation can help to prevent incorrect feature tags. However, the students did not use it very often (only for 14.04% of all issues). A minority reported that wrong recommendations for feature tags decrease the usefulness. One reason could be that the students were presented with all feature tags including those with low confidence values. Therefore, we need to study how recommendation usage can be improved.

Overall, the students found the approach useful ($RQ_2$). The use of the dashboard to track the progress was rated as easy and very useful. The students stated that the dashboard exactly provides the data needed to perform the tracking. However, they missed the functionality to view metrics for past sprints in the dashboard. Also, a minority perceived the initial set-up of the dashboard somewhat cumbersome.

The result for the intention ($RQ_3$) is rather poor compared to the other two variables. The rather low intention to use the dashboard could be due to the fact that this is mainly helpful for the project manager and the students do not see themselves as project managers in the short future. The students justified the assessment for the intention with the dependency on the project size. In smaller projects with a similar scope as this evaluation project, the students would rather not use the approach and the tools, as the application and maintenance of the tags creates overhead for developers and they can remember the feature knowledge by themselves. In very large projects, students indicated that there could be many feature tags that need to be managed. Therefore, they would apply the approach and tool support in those projects. Some students used the feature tags in code to locate code parts that might be affected by bugs. We plan to investigate whether improved support for locating code affected by bugs would raise the motivation for future usage.

Overall, our hypotheses hold and we conclude that our approach is feasible and accepted. We applied our approach in a new project, but it could be also applied to an existing project. The requirements and the code files of an existing project must be equipped with feature tags in retrospect to make our approach work. The effort for this redocumentation is considerable as the relations between all requirements and all code files have to be mapped to the features. However, it can be incrementally done during refactoring or other changes to features.

### 4.3. Threats to Validity

We discuss threats according to Runeson et al. [16].

**Construct Validity:** The construct validity was ensured through data source triangulation by using direct methods (semi-structured interviews with open and closed questionnaires) as well as indirect methods (review of data produced by tool logging). A possible threat is that researcher and interviewee might interpret the questions differently. The threat is mitigated by the format of face-to-face interviews, which enabled the interviewees to ask questions. Also, another researcher checked the questionnaire for applicability and understandability.

**Internal Validity:** The students knew that the researcher had developed the approach and its tool support. The threat was mitigated as the researcher appreciated both positive and negative feedback from the students. The motivation of the students to use the approach and its tool support might be influenced by worries about grades. However, the researcher was not involved in the final grading and the usage had no influence on the grades.

**External Validity:** The documented feature knowledge is specific to this development project and the size of the development project is limited. Moreover, we applied our approach for a new project only. Thus, the results for other projects can be different from the results reported in this study, and the findings cannot be generalized for developers working in industry. However, the project contained situations common to projects in industry, e.g., the elicitation of requirements by the participants, changing requirements due to changed customer needs, as well as communication problems across developers regarding their tasks.

**Reliability:** One researcher did the interviews and assessments to ensure consistency. Other researchers might interpret the results in another direction. The researcher documented the steps during design, data collection, and analysis. In addition, another researcher reviewed the design of the case study and the steps for analysis to increase reliability. This also ensures the reproducibility of the study.

## 5. Related Work

Our approach relates to traceability. *TAFT* provides a coarser-grained traceability as commonly used trace links, as the relations between requirements and code files are established on feature level. In [18], we compared the trace links resulting from tags with other approaches to create traceability links.

There are few approaches which also use tags to document and exploit feature knowledge. Mainly they are from the area of product lines where features are used to manage variability. Thus, they are more heavyweight than our approach. Savage et al. [17] present an Eclipse based tool for locating and tracing feature in code. The underlying approach is different from ours, as they do not directly tag code with its implementing features. Instead, a user manually relates code to feature using an annotation after feature location was performed. Similar to the metrics in our dashboard, they provide a view showing the distribution of features across the code. Ji et al. [10] present an approach to equip code with feature annotations. They simulated the development of a product line of cloned projects using the annotation approach. They found that maintaining such annotations in code is not costly, but useful for maintenance tasks. Andam et al. [1] and Burger et al. [5] both present a standalone tool for locating features in code. Both use comment annotations to document features in code and they use feature location techniques to annotate the code (semi-)automatically. Andam et al. [1] also provide a dashboard for viewing feature metrics. In contrast to ours, their dashboard targets at developers by providing more specialized metrics, e.g. nesting depths of annotations. As the tools focus on feature location, all of them provide capabilities to navigate in feature knowledge documented in code. None of the tools tag specifications nor do they provide recommendations or inheritance of tags.

## 6. Conclusion & Future Work

In this paper, we reported on the tool support of our *TAFT* approach and its acceptance in a case study with students. Overall, the results show that the approach and tool support are accepted.

In future work we would like to address the problems experienced by the students. To further ease the application of feature tags, we are also working on recommendations to suggest feature tags in code and on inheriting feature tags for code. In addition, we want to investigate how practitioners think about our tool, e.g., by performing interviews with practitioners.

**Acknowledgement.** We would like to thank all students for their effort in this study.

## References

[1] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron. Florida: Feature location dashboard for extracting and visualizing feature traces. In *11th Int. Work. on Variability Modelling of Software-intensive Systems*, pages 100–107. ACM, 2017.

[2] O. Baysal, R. Holmes, and M. W. Godfrey. Situational Awareness: Personalizing Issue Tracking Systems. In *35th Int. Conf. on Software Engineering*, pages 1185–1188. IEEE, 2013.

[3] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature?: A qualitative study of features in industrial software product lines. In *19th Int. Conf. on Software Product Line*, pages 16–25. ACM, 2015.

[4] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press Books. Addison-Wesley, 2000.

[5] A. Burger and S. Grüner. Finalist2: Feature identification, localization, and tracing tool. In *25th Int. Conf. on Software Analysis, Evolution and Reengineering*, pages 532–537. IEEE, 2018.

[6] F. D. Davis, R. P. Bagozzi, and P. R. Warshaw. User acceptance of computer technology: A comparison of two theoretical models. *Manage. Sci.*, 35(8), Aug. 1989.

[7] M. W. Godfrey and D. M. German. The Past, Present, and Future of Software Evolution. In *Frontiers of Software Maintenance*, pages 129–138. IEEE, 2008.

[8] K. Herzig and A. Zeller. The impact of tangled code changes. In *10th Work. Conf. on Mining Software Repositories*, pages 121–130. IEEE, 2013.

[9] S. Jantunen, L. Lehtola, D. C. Gause, U. R. Dumdum, and R. J. Barnes. The Challenge of Release Planning. In *Int. Work. on Software Product Management*, pages 36–45. IEEE, 2011.

[10] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki. Maintaining feature traceability with embedded annotations. In *19th Int. Conf. on Software Product Line*, pages 61–70. ACM, 2015.

[11] N. Kama. Change Impact Analysis for the Software Development Phase : State-of-the-art. *Journal of Software Engineering and Its Applications*, 7:235–244, 2013.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[13] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *22nd Int. Conf. on Program Comprehension*, pages 262–265. ACM, 2014.

[14] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. Feature-oriented software evolution. In *7th Int. Work. on Variability Modelling of Software-intensive Systems*. ACM, 2013.

[15] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer, 2014.

[16] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, Hoboken, NJ, USA, 1st edition, 2012.

[17] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: feature location and textual tracing tool. In *32nd Int. Conf. on Software Engineering*, pages 255–258. IEEE, 2010.

[18] M. Seiler, P. Hübner, and B. Paech. Comparing traceability through information retrieval, commits, interaction logs, and tags. In *10th Int. Work. on Software and Systems Traceability*. IEEE, 2019. (accepted to be appear).

[19] M. Seiler and B. Paech. Using tags to support feature management across issue tracking systems and version control systems. In *23rd Int. Work. Conf. Requirements Engineering Foundation for Software Quality*, pages 174–180. Springer, 2017.

[20] M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, and M. Muller. How Software Developers Use Tagging to Support Reminding and Refinding. *IEEE Transactions on Software Engineering*, 35:470–483, 2009.

[21] M. Sulír, M. Nosáľ, and J. Porubän. Recording concerns in source code using annotations. *Computer Languages, Systems & Structures*, 46:44–65, nov 2016.