Requirements Engineering (2018) 23:381–399
https://doi.org/10.1007/s00766-018-0291-4

RE 2017

Discovering undocumented knowledge through visualization of agile software development activities

Case studies on industrial projects using issue tracking system and version control system

Shinobu Saito[1] · **Yukako Iimura**[1] · **Aaron K. Massey**[2] · **Annie I. Antón**[3]

Received: 24 October 2017 / Accepted: 23 March 2018 / Published online: 4 April 2018

Abstract
In agile software development projects, software engineers prioritize implementation over documentation. Is the cost of missing documentation greater than the cost of producing unnecessary or unused documentation? Agile software engineers must still maintain other software artifacts, such as tickets in an issue tracking system or source code committed to a version control system (VCS). Do these artifacts contain useful knowledge? In this paper, we examine undocumented knowledge in a multi-case exploratory case study of industrial agile software development projects. The first is an internal project with 159 source code commits and roughly 8000 lines of code. The second is an external project with 760 source code commits and roughly 50,000 lines of code. We introduce a ticket-commit network chart (TCC) that visually represents time-series commit activities along with filed issue tickets. We also implement a tool to generate the TCC using both commit log and ticket data. Our case study revealed that software engineers committed source code to the VCS without a corresponding issue ticket in a non-trivial minority of instances. If these commits were based on and linked to individual issue tickets, then these "unissued" tickets would have accounted for a non-trivial minority (5–21%) of the knowledge needed for future software modification and operations. End users and requirements engineers also evaluated the contents of these commits. They found that the omission of links to individual tickets had an important impact on future software modification or operation with between 22 and 49% of these instances resulting in undocumented knowledge.

Keywords Agile · Agile development · Agile requirements · Requirements management · Requirements knowledge

1 Introduction

Agile software development is now an established approach to building software systems [14, 19, 22, 27]. The basic concept of agile development practices is that software engineers prioritize implementation activity, customer interaction, and tight iteration cycles over up-front requirements analysis and documentation. In contrast to other software methodologies, such as the traditional waterfall model, they generally put more emphasis on software implementation (i.e., working software) than its documents [4]. In this manner, software developers can respond flexibly to stakeholder needs. Of course, the culture of the organization and team always influence the amount of documentation generated no matter the development approach. However, within this paper, we observe that even when there is a strong culture of using ITS and VCS, the relevance of this kind of

✉ Aaron K. Massey
akmassey@umbc.edu

Shinobu Saito
saito.shinobu@lab.ntt.co.jp

Yukako Iimura
iimura.yukako@lab.ntt.co.jp

Annie I. Antón
aianton@gatech.edu

[1] Software Innovation Center, NTT Corporation, Tokyo, Japan

[2] Department of Information Systems, University of Maryland, Baltimore County, Baltimore, MD, USA

[3] Georgia Institute of Technology School of Interactive Computing, Atlanta, GA, USA

documentation is not always valued as essential or necessary as in more traditional software development approaches.

Agile software development is not typically a documentation-driven process. During agile software development, software engineers prioritize implementation activity to address stakeholder requests. Implemented features may differ from any documentation requirements engineers initially created. To mitigate this, Scrum [22, 25] approaches hold sprint reviews and retrospective meetings wherein the project members (i.e., requirements engineers and software engineers) ensure software documents are properly maintained.

Cockburn describes Agile software development as a cooperative game in which engineers should pay attention to both the current game (i.e., the current project) and the next game (i.e., future projects) [5]. Software projects must maintain knowledge required by "current" and "future" project engineers. Current engineers might need to know the specifications and constraints of the software product. Future engineers might need to know software architecture details or features and the rationale behind development and modification of the product. In agile software projects, we question whether agile software engineers sufficiently document the knowledge required for these stakeholder groups.

This paper details an explanatory multi-case study with two cases conducted to examine two agile software development projects: an internal NTT R&D project [18, 26] and an external, commercial system development project for an NTT customer. NTT is a global service and network provider. NTT Laboratories have more than 3000 researchers both in Japan and abroad. Approximately 1000 researchers are engaged in R&D activities in services and solutions. About 100 software development projects are launched annually to create software products (i.e., prototypes, tools) for services and solutions. Most of these projects are completed within a year making them a seemingly ideal fit for the rapid iterative development style supported by standard agile methods. After the software products are developed and trial evaluations of the project are conducted in the Laboratories, the software products are provided to the NTT Group companies. Next, those companies use or, if necessary, modify the software products for their business activities.

End users and the engineers in the NTT Group companies are first exposed to software products after delivery by NTT Laboratories. When they need to operate or modify these products, they must rely on extant documentation. Knowledge transfer from NTT Laboratories to other companies is a decisive factor for business success. Lack of documentation may cause users and engineers to struggle to use or modify the software. Documentation can take several forms, including software specifications, constraints, architecture, features, and rationales. Each type of documentation may affect future modification or use of the software

The first case of this study was originally reported in our prior work [26]. In this paper, we revise and augment the presentation of that work with extended discussion, additional comparisons to related work, and several new figures and tables. Moreover, we extend and compare our prior research with an entirely new second case as we now discuss. The second case employs a large-scale software system development project outside of NTT. This project is a commercial system development effort, so the nature of the project differs from the first case, which is an R&D project at NTT. Neither end users nor engineers (i.e., requirements engineers and software engineers) are NTT employees. (This was not true for the first case, which was an internal NTT project.) The target system for the second case is not developed from scratch. Because this project comprises a commercial system development effort, almost all engineers are contracted by the organization to which the end users belong. Some engineers were contracted out of the project during the development period. As a result, new engineers joined the project. As in other NTT's projects [24, 25], knowledge transfer from existing members to new members is an important factor for ensuring continuity of development. Without documentation, new development team members will experience difficulty in modifying and/or operating the software systems.

Issue tracking systems (ITSs) and version control systems (VCSs) are commonly used in agile software development projects [10, 20–22, 28, 29]. In this study, we examine the time-series activities of the project engineers (i.e., requirements engineers and software engineers) using both ticket data in an ITS and commit logs in a VCS. Requirements engineers create issue tickets using the ITS. Each ticket describes a task for software engineers. Software engineers implement the product features and commit source code to the VCS based on issue tickets. After the feature has been approved during a daily meeting, the corresponding ticket is closed.

Ideally, all changes to the source code committed in the VCS will correspond to a ticket filed in the ITS. In practice, requirements engineers sometimes describe tasks to software engineers without issuing tickets. This could happen through in-person conversation. Such a scenario often leads to committed source code that does not correspond to a ticket in the ITS. In this paper, we refer to a requirement implemented without a corresponding ticket as an "unissued ticket."[1] As

[1] A commit may be unlinked to an issue ticket at the time it is merged into the repository. This could happen for one of two reasons. First, the issue ticket may exist, and the software engineer simply failed to link the commit to it. Second, the issue ticket may not exist. We refer to both cases as "unlinked commits" and to the second cases as "unissued tickets." Commits should, however, always be linked to

the number of unissued tickets increases, we expect that the knowledge required for future modification or operation may be undocumented. In this context, modification refers to the addition of new features or functions, and operation refers to use of the software system according to its existing goals and features.

In this paper, we describe a ticket-commit network chart (TCC) which links ticket data in the ITS with the commit log data in the VCS. The TCC was previously introduced in our prior work [26]. It provides a visual representation of time-series commit activities, whether they are linked or not linked to the corresponding tickets. The TCC is generated using a tool, which is based on software implementation.

Using the TCC, we seek to answer the following research questions by means of empirical evaluation in agile software development projects in both inside and outside of NTT:

RQ1 How many unissued tickets are created in agile software development projects?
RQ2 How much undocumented knowledge is created by unissued tickets that may be required later for future operation or modification?

We answer RQ1 by measuring how often software engineers modify the source code without tracing the changes to a stakeholder request. Changes that are not traceable to stakeholder requests are often, but not always, problematic in practice. We answer RQ2 by examining how often those changes affect future development. Both questions address the extent to which documentation in standard agile projects is insufficient for future development.

Our first case establishes baseline answers to our two research questions [26]. Our second case provides an important point of comparison. In addition to completely replicating the results of the first case, we are able to demonstrate that these results also hold in a large-scale external software project rather than a smaller, internal software project. The extent to which there are unissued tickets decreased, but the essential findings of the research question remained the same.

The remainder of this paper is organized as follows: Section 2 describes related work with an emphasis on visualization of development activities and knowledge management within agile software development. Section 3 provides an overview of the agile software development projects we examined and evaluated. Section 4 introduces our approach for visualizing agile software development activities. Section 5 presents our empirical analysis procedure and results from the two cases of our study. Section 6 discusses the implications of our findings. Section 7 describes the limitations of this work, and Section 8 summarizes the paper and presents future work.

2 Related research

Two areas of related research provide important background. First, we discuss research related to software development tool support and visualizing software development activities. Second, we introduce research on managing knowledge in agile software development environments.

2.1 Tool support and visualization of software development activities

Visualization of software development is a common challenge in software engineering. Wnuk et al. [33] proposed a feature transition chart (FTC) to visualize scope dynamics within and across multiple projects. The scope of each project is maintained in a feature list. The FTC provides a comprehensive overview of the timing and magnitude of feature transitions among multiple projects. The feature list is similar in style and format to the list of tickets in an ITS we used. However, we focused on commits and corresponding ticket activities. Our TCC provides an overview of the timing of commits, whether they are linked to the corresponding tickets or not.

Lanza et al. [12] developed a real-time visualization tool, called Manhattan, for team activity within software development. This tool, built as an Eclipse plug-in, visualizes projects in the Eclipse workspace using a 3D city metaphor. It depicts a living city where code changes from team members and potential conflicts are animated with different colors and shapes. The preliminary evaluation shows positive reactions in terms of team collaboration. However, their visualization approach notifies engineers of programing activities, whereas our visualization approach notifies engineers of software documentation activities.

In our prior work [24], we proposed an approach to track requirements evolution using tickets in an ITS. We provided seven rules that describe the identification of requirements evolution events (e.g., refine, decompose, and replace) based on combinations of operations (e.g., add, change, and delete) in the tickets. We defined a requirements evolution chart (REC) to visualize requirements evolution history. We also examined whether the REC supports new requirements engineers conducting an impact analysis [25]. We found that new requirements engineers using the REC could identify artifacts affected

Footnote 1 (continued)

an issue ticket because of the explicit methodological desire to connect one commit with one issue ticket in both cases of our study. Even if a commit could have been linked to an already issued ticket, the fact that it was not remains problematic.

by change requests more accurately and quickly than requirements engineers attempting the same task without the REC. Our approach in that work focused only on the tickets in an ITS for visualization of activities of requirements engineers. Our approach in this research involves both ticket data in ITS and commit data in a VCS. In addition, we visualize software development activities of both requirements engineers and software engineers.

Comparing other multi-case studies with our own can clarify and differentiate our exploratory research design from explanatory work, both of which are common approaches to case-study research. Exploratory research identifies and describes phenomena, whereas explanatory research discovers causal relationships [23, 34]. Dhungana et al. [7] and Thurimella et al. [30] both develop tools using explanatory multi-case studies. Thurimella et al. [30] build plug-ins for requirements tools using case-based research. Their multi-case study is built on the theory-building process [9], and their results show how practitioners can minimize implementation effort when extending an RE tool to manage variability within software product lines. Their work is explanatory. They seek to explain why their approach to variability management is successful using successive cases to enhance their explanation. Similarly, Dhungana et al. [7] also develop a meta-tool through an explanatory multi-case study to improve reuse of assets across multiple product lines. They examine the flexibility, extensibility, and adaptability of their tool through four separate cases. Their work is also explanatory, and they use additional cases to mitigate bias and threats that would otherwise affect the validity of their results. In contrast to both Thurimella et al. and Dhungana et al., our work is exploratory. We seek to describe, not to explain the causes of, undocumented knowledge in agile software development. For example, we are interested in the extent to which undocumented knowledge is present and whether engineers and users view it as potentially problematic. We are not seeking to explain why undocumented knowledge is present or why it is problematic. We use successive cases, therefore, to broaden our exploration rather than to refine an explanation.

The TCC developed and employed in our study serves an entirely different purpose than the tools developed by Dhungana et al. [7] and Thurimella et al. [30]. Their multi-case studies are designed to explain the effectiveness of their tools in addressing their respective problems. Our primary goal is not to establish the effectiveness of the TCC its or efficiency, but to explore undocumented knowledge as a part of the agile software process. Tools similar to the TCC already exist in industry [21]. As researchers, however, we describe the TCC in detail because the practitioners involved in this research study make use of it. In addition, we believe future studies evaluating the TCC would prove both fruitful and interesting.

2.2 Knowledge management for agile software development

Knowledge management in software engineering is crucial because software development is a knowledge-intensive activity. We discuss knowledge management in requirements engineering and in agile software development.

Moe et al. [16] presented their experience in developing and maintaining agile team knowledge, especially virtual teams in two countries. They focused on shared knowledge of tasks and how to carry them out, who knows what on the team, the development process, and team goals. They also discussed developing team knowledge in a global software development project. Herein, we focused exclusively on undocumented knowledge in software development.

Levy and Hazzen [13] discussed how an agile software development team extracts tacit knowledge without extra effort. They indicate that when an agile team tries to introduce and apply knowledge management, overcoming cultural and psychological barriers is important. To improve knowledge extraction and sharing, they discuss several practices (e.g., whole team, collaboration workspace, and stand-up meeting). Previously, we introduced a visualization approach for recovering undocumented knowledge that has similar coordination benefits [26]. This tool is used in both cases of our study.

Maalej and Thurimella [14] provide an introduction to knowledge management in RE, including extensive discussions of identifying requirements knowledge and tool support for requirements knowledge management. In this work, we are primarily interested in whether or not a particular type of requirements knowledge is lost in agile software development teams. For example, are source code commits no longer traceable to the issue tickets (and thus to the requirements) that motivated their own creation? Maalej and Thurimella present several alternative approaches that may be adapted within agile software projects to support this aspect of requirements knowledge management.

Dorairaj et al. [8] investigated knowledge management in agile software development using grounded theory. They identified approximately 20 practices that promote effective knowledge management in agile software development. These practices are categorized through the following knowledge-management processes: knowledge generation, knowledge extraction, knowledge transfer, and knowledge application. They also provide an overview of knowledge-management practices for agile software development. Our approach focuses on a version of their knowledge transfer problem, but the TCC developed herein may provide benefits to other knowledge-management practices they identified.

Thurimella et al. [31] provide guidelines managing knowledge of the reasoning behind software development decisions, expressed as rationales. Their list of rationales

questions (LoRQ) organizes questions according rationale concepts: issue, option, criterion, argument, consequences, and decision. For instance, questions of issue are designed to elicit information when specifying an issue. However, the guidelines and the list are not designed for development projects in which decisions are quick and rapidly changed. Our approach is designed for that environment, as found in agile software development.

Hoda et al. [11] examine documentation patterns in agile software development to identify some of the problems, their proximate causes, and some suggested solutions. For example, some agile teams may follow the "Fake Documentation" pattern in an attempt to create "just enough" documentation to satisfy management. In this case, Hoda et al. suggest timing the minimal amount of time needed to produce enough traditional documentation that the team could coordinate with a non-agile team. This could then serve as a guideline to ensure the "Fake Documentation" pattern is avoided. Our approach does not consider failure patterns for documentation generally. However, we believe our examination of the reasons software engineers commit code to the source code control repository without an associated issue ticket may indicate a new pattern not identified in this prior work.

Stettina and Heijstek [28] examine perceptions of documentation among 79 agile software professionals. Over half of their participants rated documentation as "very important" and said that too little documentation was available in their own projects. Moreover, they did not seem to agree with the agile principle that face-to-face communication is the most efficient method of conveying information. Although we did not survey developer perception of documentation, the results of our retrospective meetings for both cases suggest that a significant portion of the undocumented knowledge in our study would have been problematic. This finding aligns with the perceptions identified by Stettina and Heijstek.

Voigt et al. [32] conducted a follow-on study for the work by Stettina and Heijstek [28]. In that study, they examined documentation and information searches in agile software development through interviews, observation sessions, and online surveys. They also specifically look at issue tickets as a documentation problem. Their findings support those of Stettina and Heijstek. Moreover, they identified documentation gaps as particularly detrimental for agile projects. That is, the "worst case scenario" is when agile developers assume some documentation exists because their process suggests it should and they are unable to find that documentation with a search. This is precisely the problem we are attempting to address herein by creating a tool-supported method to identify gaps in documentation during retrospective meetings.

Mendes et al. [15] examine documentation as a technical debt concern in agile software projects. Their retrospective case study suggests that the costs of undocumented knowledge in this instance were a 47% increase in maintenance costs and an additional 48% of the initial cost of development. Although we do not examine the costs of undocumented knowledge in this study, we find these results particularly motivating. The increase in maintenance costs corresponds closely with our results examining the effect of undocumented knowledge on ongoing software operations.

3 Summary of case-study projects and procedures

3.1 Project description and schedule

3.1.1 First case study

The software product for this study is a prototype of a graphical modeling tool used to draw an enterprise system model. The estimated budget of the complete project is about ten million dollars. The tool was developed to support the engineers as a part of a system development project in NTT Group. This tool is designed to provide functionality specific to NTT Group's system development process.

The development schedule consists of three phases: planning, iteration one, and iteration two with retrospective meetings at the end of each iteration. Phase durations are 3, 4, and 4 weeks, respectively. There are three roles: end user, requirements engineer, and software engineer. Personnel filling each role number four, two, and two, respectively.

3.1.2 Second case study

The project for this study is a large-scale commercial system development. The system was developed for individual agriculture farmers to support their operational agricultural activities: sowing seeds, harvesting, and spraying pest. We selected a single application development project from this larger effort that is designed to provide analysis reports (e.g., weather forecast, satellite image, and pest prediction) to the farmers for planning their working schedules.

The development schedule consists of three phases: planning, iteration one, and iteration two with retrospective meetings at the end of each iteration. Phase durations are 2, 3, and 4 months, respectively. There are three roles: end user (data analyst), requirements engineer, and software engineer. Personnel filling each role number four, three, and five, respectively.

In the planning phase, requirements engineers and end users elicit the software requirements specification together. The specification includes features, user interface, business logic, and so on. Each feature is broken down into a set of implementation tasks for the next iteration phase.

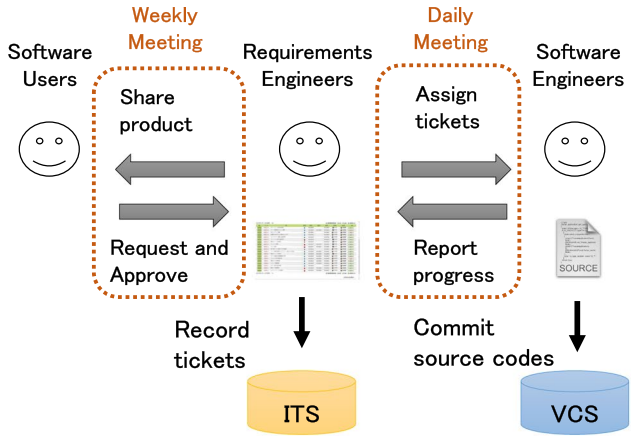

Fig. 1 Actors, their communications, and tools in two iteration phases (Case 1)

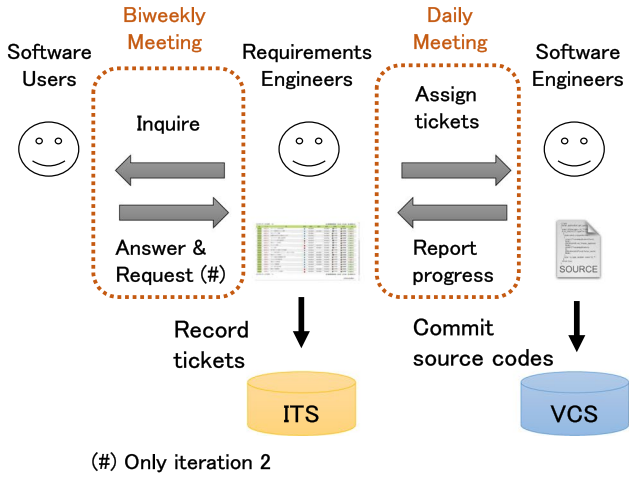

Fig. 2 Actors, their communications, and tools in two iteration phases (Case 2)

3.2 Daily and weekly meetings

3.2.1 First case study

In the iteration phases, as shown in Fig. 1, the project holds two types of meetings: daily and weekly. Requirements engineers and software engineers get together every evening for the daily meeting. Before the meeting, requirements engineers create issue tickets in the ITS. Each ticket reflects one task to implement a natural language requirement. Some requirements directly address bugs identified during weekly meetings. Requirements engineers assign the tickets to the software engineers during the daily meeting. Following the task descriptions of the assigned tickets, software engineers carry out implementation tasks and commit their source code to the VCS. The software engineers provide a progress report to the requirements engineers at the next daily meeting.

This project employs the on-site customer method [3] wherein requirements engineers hold a weekly meeting to share the product with the end users. The users examine and use the product. When they approve of the features developed, the requirements engineers close the corresponding tickets. However, end users often request new features or changes to existing features. In this case, requirements engineers create new issue tickets reflecting these requirements after the meeting. The requirements engineers then assign these new tickets to the software engineers at the next daily meeting.

3.2.2 Second case study

As shown in Fig. 2, the project studied in our second case holds two types of meetings: daily and biweekly. As in the first case, requirements engineers and software engineers get together every evening for the daily meeting. However, requirements engineers and end users hold a biweekly meeting—rather than a weekly meeting—to communicate with each other. As in the first case, each ticket reflects one task to implement a natural language requirement, and some requirements directly address bugs identified during biweekly meetings. During iteration 1, requirements engineers only inquire about features with end users in the meeting. The application was not shared with end users until the end of iteration 1. At the end of iteration 1, the trial version of the application was released to end users. Then, end users operated the trial version and requested changes during the biweekly meeting for iteration 2. As in the first case, requirements engineers created new issue tickets reflecting changes to requirements after the meeting. They assigned these new tickets to the software engineers at the next daily meeting. The team also held retrospective meetings, which are common in agile software development methodologies, to ensure documents and tickets were maintained. An important goal of our work is to improve these meetings by supporting them with the TCC, but in both cases an initial retrospective was held without the TCC for comparison.

3.3 Issue tracking system and version control system

For our two cases, we examined development activities during the two iteration phases using data from our ITS and VCS. Backlog [2] is a web-based application on public cloud services, and it was selected as the ITS for this project. Figure 3 shows a screenshot of the ITS. Each ticket includes nine data items: ticket ID, subject, task description, priority, iteration no., issue date, due date, assignee, and status. Software engineers used Subversion [1] as the VCS. A server

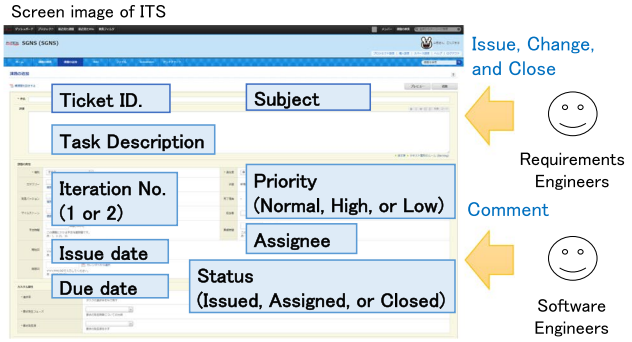

Fig. 3 Recording items in ticket

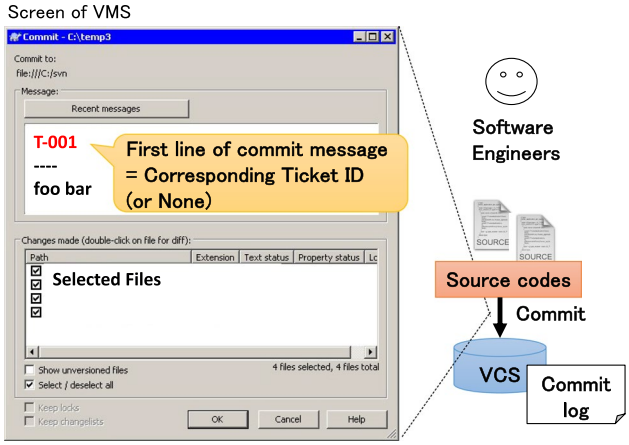

Fig. 4 Entering corresponding ticket ID in commit message

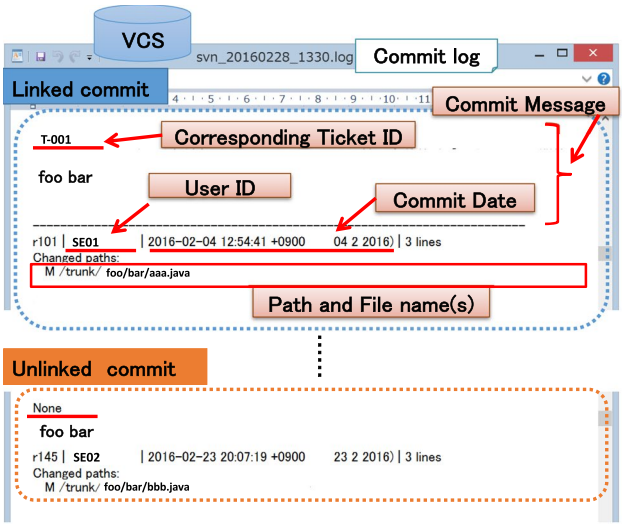

Fig. 5 Commit log (linked commit and unlinked commit)

for the VCS was deployed over a private LAN. During the iteration phases, the software engineers were responsible for committing updated source code to the VCS. Both projects, and thus both cases, employ the same tools.

In these projects, only requirements engineers could create new issue tickets or close existing tickets. Software engineers can refer to or add comments to the task description in the tickets. They may do this to ask questions or make suggestions about the specifications to requirements engineers. When a ticket is newly issued, requirements engineers enter the initial subject, task description, iteration number, and due date. They also select one of three priority levels: "normal," "high," or "low." The ticket ID is automatically set by the ITS, and the status is set to "issued." At the daily meeting, requirements engineers receive a progress report from the software engineers and then decide to whom to assign the new tickets. After assignment, requirements engineers set the name of the assigned software engineer as the assignee in the ticket. The status of the ticket is changed from "issued" to "assigned." Later, when software users approve of the feature as implemented by the software engineer following the task description in the ticket, the requirements engineers set the status to "closed."

4 Visualization of agile development activities

4.1 Linking ticket to commit

During the iteration phases, software engineers implement the software product following the task descriptions of the tickets assigned by the requirements engineers per the following three guidelines for software engineers:

1. When software engineers commit source code to the VCS, they must enter the corresponding ticket ID in the first line of the commit message.
2. Only one ticket ID is permitted per commit message.
3. If no ticket has been issued, the software engineers had no choice but to enter "None" in the first line of the commit message.

Figure 4 shows the dialog box of the commit message. Based on the guidelines, a software engineer might enter ticket ID "T-001" in the first line of the commit message. This ticket ID would link the commit to the corresponding ticket in the ITS. Occasionally, we expect a requirements engineer may assign a task to a software engineer without issuing a ticket because they may be pressed for time. In this case, the software engineer had no choice but to enter "None" in the commit message.

Figure 5 shows a commit log file exported from the VCS. As described above, there are two types of commits: linked and not linked, shown on the upper and lower parts

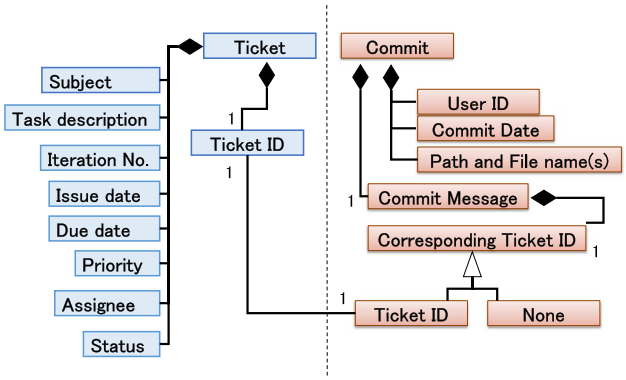

Fig. 6 Meta-model of information on ticket and commit

of Fig. 4, respectively. Each commit includes a commit message, a user ID, a commit date, and the path and file names of files affected by the changes in the commit. In the linked commit, the corresponding ticket ID "T-001" is recorded in the first line of the commit message, and "None" is recorded in the unlinked commit.

Figure 6 illustrates a meta-model that represents the relationship between information on ticket and commit log data. As mentioned in the guidelines, ticket ID, which is set as a value in the corresponding commit message, links the ticket and commit.

4.2 Ticket-commit network chart

We previously introduced our TCC which provides a visual representation of time-series commit activities as described in the VCS and the ITS in our prior work [26]. The TCC represents both commits that are linked to tickets in the ITS and commits that are not linked to tickets in the ITS. We answer the following questions using our TCC:

- What source code was committed but not linked?
- When did unlinked commits occur?
- How many times did unlinked commits occur?

As shown in Fig. 7, the TCC visualizes both life spans of tickets and occurrences of commit activities by displaying colored cells on the spreadsheet. In this figure, the blue cells represent linked commits, and red cells represent commits not linked to issue tickets. The red cells in the second column represent files affected by unlinked commits. We consider blue cells acceptable and red cells unacceptable. The gray cells represent the life span of an issue ticket, which is the period from the issue date to close date on the ticket.

This figure visualizes the time-series commit activities of three source code files (aaa.java, bbb.java, and ccc.java) for ten successive days (from 9/1 to 9/10). The time length of one column corresponds to one day. The path and file names of the files appear in the left two columns. The ticket ID and user ID are the third and fourth

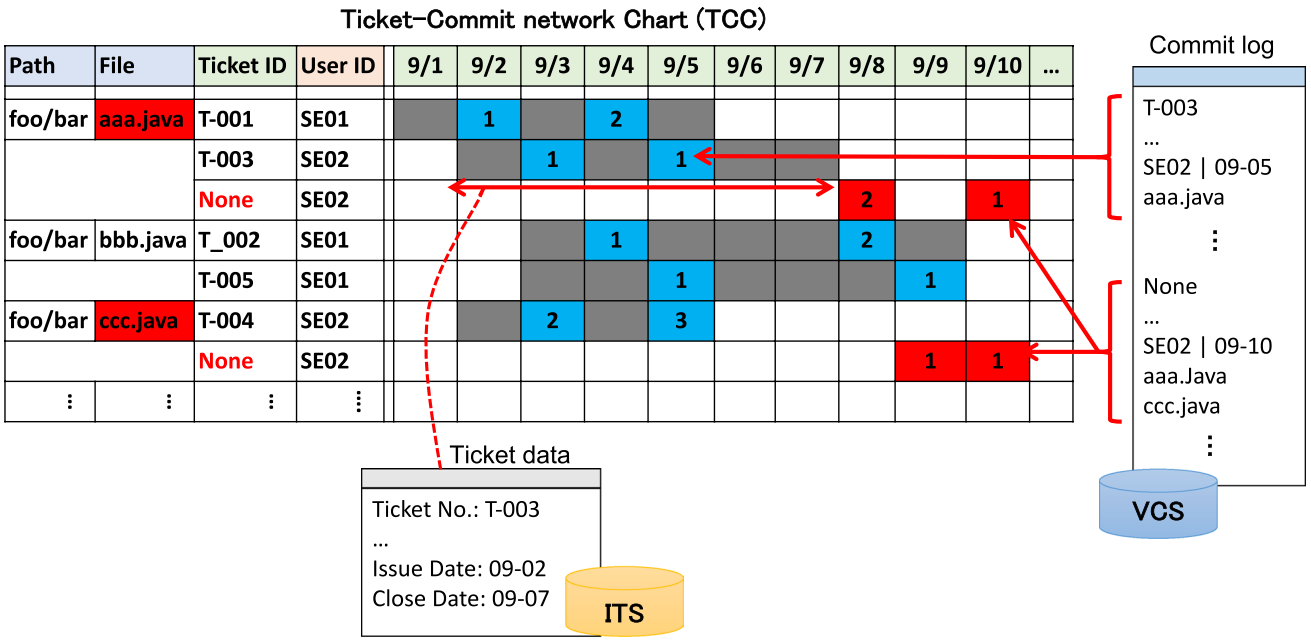

Fig. 7 Ticket commit network chart (TCC)

columns, respectively. The ticket ID is derived from the commit message. The time-series activities of commits are in the sixth and successive columns, as identified by date labels (e.g., 9/1, 9/2).

Consider the fourth line of the figure. Ticket "T_003" was issued on 9/2 and then closed on 9/7. The life span of the ticket was six days. As shown in the bottom part of Fig. 7, the issue date and close date of each ticket of the ITS refer to the life span. In the same (fourth) line, there are two blue cells, which means file "aaa.java" was committed on two days (9/3 and 9/5) by user "SE02." The numbers in the two blue cells also show that SE02 committed the file "aaa.java" one time on both days. If SE02 engineer committed multiple files simultaneously, then multiple blue cells will correspond to the committed files.

In the next line (i.e., the fifth line), the two red cells indicate the occurrences of the commits not linked to the corresponding tickets. "None" and "SE02" appear in the ticket no. and user ID columns, respectively, which means that user SE02 committed the "aaa.java" file on the two days (9/8 and 9/10) without entering the corresponding ticket IDs. Like the blue cells, the numbers in the two red cells show the number of commit times. The two red cells in the fifth line mean that SE02 committed aaa.java without the corresponding ticket ID two times on 9/8 and one time on 9/10. The two red cells in the second column, which correspond to aaa.java and ccc.java, represent the two files are committed without a corresponding ticket ID. In other words, two commits affecting these files (i.e., the fifth and ninth lines) include red cells.

As shown on the right side of Fig. 7, the commit logs in the VCS are denoted as red cells. From the VCS, as shown in Fig. 5, we use four data items: corresponding ticket ID, user ID, commit date, and path and file name(s). Unlike a linked commit, one unlinked commit might color more than one line in the TCC. On the lower right side of Fig. 6, one unlinked commit log includes two files, aaa.java and ccc.java. In addition to the cell on 9/10 in the fifth line that corresponds to aaa.java, the cell on the same day in the ninth line is also red. This line corresponds to ccc.java.

The TCC supports a complete analysis of the commit history for a project. All source code files and all commits are represented. By linking this information with tickets from the ITS, we are also able to completely examine the history of implementing issue tickets as filed. For example, a complex issue ticket may remain open for several days and comprise several commits.

4.3 Software implementation

We implemented a tool to automatically generate the TCC. Figure 8 provides an overview of the TCC generator. From

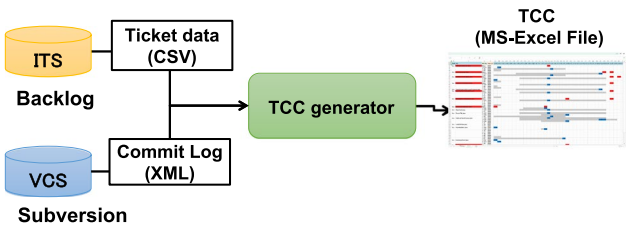

Fig. 8 TCC generator

two inputs, ticket data (CSV file) exported from Backlog [2], and commit log (XML file) exported from Subversion [1], the tool automatically generates the TCC as a spreadsheet in Microsoft Excel. Figure 10 shows a screenshot of the TCC generated using the TCC generator in our study.

5 Case study

To answer the research questions given in Sect. 1, we conducted an exploratory case study with two cases on agile software development projects both inside and outside of NTT Laboratories. Table 1 details relevant artifacts from each project.

Case studies must include each of the following elements: objective, cases, theory, research questions, methods, and selection strategy [23, 34]. The objective of this work is to evaluate the amount and utility of missing documentation in agile software development projects. The two cases are a small, internal project and a larger, external project, both of which use an agile software development methodology. Our theory is that agile software development projects focus more on implementation than documentation and that some of this focus results in undocumented knowledge that would be useful in both future modification of the software and future operation of the software product by end users. Our two research questions can be found in the introduction. Our methodology and case selection strategy are detailed in the following two subsections.

The second case was conducted after the first case. We sought to address two concerns from our first case: (1) The first case was an internal project where the software users were also engineers working for NTT, and (2) the first case was a relatively small project and may not have been representative of a "typical" software project.

Table 1 Artifacts of projects

Phase	Type	Artifacts	Volume	
			Case 1	Case 2
Planning/inception	Documents	User stories	30 pages	34 pages
		Algorithms and user interface sketches	15 pages	576 pages
Iterations 1–2	Source code (Java)	Main programs	5931 LOC	21,899 LOC
		Test programs	2093 LOC	28,861 LOC
	Documents	Tickets	102 tickets	296 tickets
		Source commits	159	760
		User manual	54 pages	78 pages

5.1 Data collection and participants

To select appropriate cases for this study, we sought to work with two agile software development projects. Both projects must be conducted primarily because the resulting product would be used (i.e., not simply so that we could study them). In addition, both projects needed to be comprised primarily of experienced engineers so as to limit the effects of inexperience on our study. Although we have attempted to use the exact same data collection procedures in both cases to mitigate threats to the reliability of comparisons between them, there are several differences as we now discuss.

5.1.1 First case study

Our first case is detailed in our prior work [26]. During the 11-week development period, two requirements engineers created the user stories and design documents (e.g., algorithms and user interface sketches) in the first phase. In the later phases (i.e., two iterations), the requirements engineers issued 102 tickets. They issued 32 tickets at the beginning of iteration 1. They also held weekly meetings seven times with four end users. The remaining 70 tickets resulted from the weekly meetings with users over the course of the two iterations.

Two software engineers implemented the software product in Java following the tasks described in the tickets. The size of the software was approximately 8000 lines of code (LOC), about 6000 LOC main program and 2000 LOC test program. During the two iterations, the total number of commits by the software engineers was 159. The requirements engineers created a user manual in the later part of iteration 2. All project members had over 10 years of experience in software development. Two requirements engineers have previous experience in agile development (6 years for 1 and 8 years for the other). Both software engineers have 2 years of experience in agile development.

5.1.2 Second case study

The total duration of one software application is 9 months, and for this case, all project personnel had previous experience using agile software development. Three requirements engineers created the user stories and design documents. For the design documents, they spent a significant amount of time creating the system screen images of analysis reports. Based on the screen of the existing system, the requirements engineers investigated both needs and problems from end users. Later in the planning phase, the requirements engineers and end users verified and validated the screens created. Then, the requirements engineers issued total 296 tickets during two iterations. The requirements engineers also created a user manual in the later part of iteration 2.

Five software engineers implemented the software application in Java. The size of the software was about 51,000 LOC, which is composed of about 22,000 LOC main program and 29,000 LOC test program. The software engineers committed in total 760 commits during the two iterations.

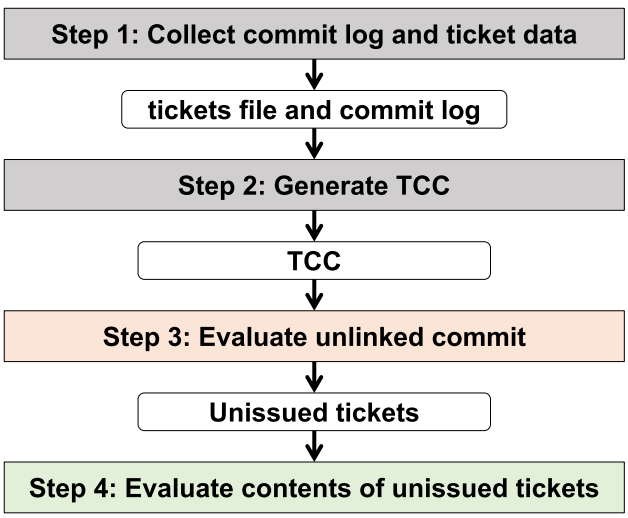

Fig. 9 Analysis procedure (four steps) in case study

Table 2 Evaluation results (first case)

Type of Commits	No. of commits	Step 3		Step 4			
		Corresponding tickets were issued or not (unissued)		Unissued tickets are needed or not			
				For software modification (from REs)		For software operation (from Users)	
Linked to tickets	124	–		–		–	
Not linked to tickets	35	Issued tickets	9	–		–	
		Unissued tickets	26	Needed	6	Needed	11
				Not	20	Not	15

5.2 Analysis procedure

To ensure reliability for our second case, we used exactly the same analysis procedure used in the first case, as we now describe. Figure 9 illustrates the analysis procedure, which contains four steps. In Step 1, we collect commit log and ticket data from the VCS and ITS, respectively. In Step 2, we generate the TCC by using the TCC generator. In Step 3, requirements engineers and software engineers identify and collect the unlinked commits found in the TCC (i.e., red cells).

In Step 4, we conducted a brief interview with the requirements engineers and end users[2] on the project shortly after their final retrospective meetings. First, the requirements engineers collectively determine whether the unlinked commits should be linked to a currently issued ticket or if the ticket remains unissued. Using their response, we answer the first research question (RQ1: How many unissued tickets are created in agile software development projects?). The requirements engineers then collectively evaluate and categorize these unissued tickets in terms of future software modification. Specifically, we asked the requirements engineers to answer the following two interview questions[3] for each unlinked commit:

RE-1 Should this unlinked commit be linked to a currently unissued ticket?
RE-2 Would the recovered ticket be impactful in supporting changes to the source code?

Then, we asked the end users collectively evaluate and categorize the unissued tickets in terms of future software operation using the following interview question for each unissued ticket:

EU-1 Would the recovered tickets be impactful in supporting user interface and usability changes?

The requirements engineers and end users could refer to the contents of the issued tickets and the corresponding source code files as highlighted by the TCC to determine whether these tickets contain the required knowledge. Based on their evaluation results, we answer the second research question (RQ2: How much undocumented knowledge is created by unissued tickets and later required for future operation or modification?).

5.3 Results

5.3.1 First case results

In Step 1, we collected the 102 tickets from the ITS and 158 commit logs from the VCS. During the two iterations, the numbers of linked and unlinked commits were 124 and 35, respectively, as shown in first and second columns in Table 2.

Figure 10 shows the time-series data on the number of commits per day. As shown on the right side of the figure, the number of unlinked commits increased in the later part of the development period. Figure 11 shows the image of the TCC generated for this case in analysis Step 2.

In Step 3, requirements engineers and software engineers examined the activities of the 35 unlinked commits for approximately 4 h. This took place about 3 weeks after the completion of the project.

As shown in the third column of Table 2 (labeled "Step 3"), they identified 26 "unissued" tickets from the contents of 35 unlinked commits. We found that in 16% (26/159) of all commits, software engineers committed source code to the VCS when no corresponding tickets were issued in the ITS.

[2] Note that we did not interview the software engineers because we felt that they would have been biased in commenting on their own source code commit data.

[3] The original interview was conducted in Japanese. Questions RE-1, RE-2, and EU-1 are English translations.

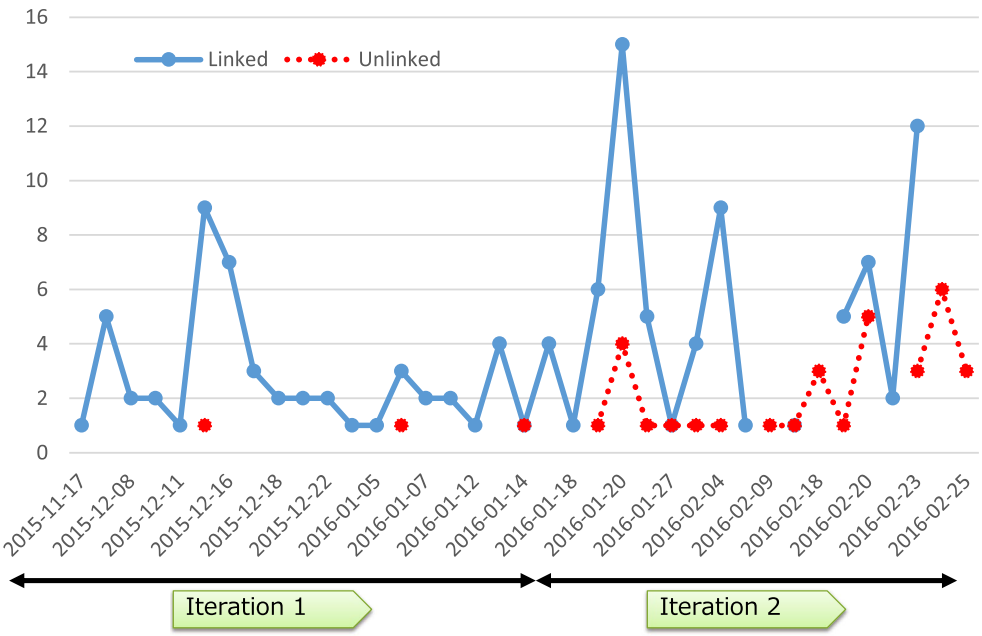

Fig. 10 First case: number of commits per day

We interviewed both the requirements engineers and the software engineers regarding the 26 unissued tickets. We found that the requirements engineers verbally assigned the tasks to the software engineers without issuing tickets because requirements engineers were too busy. Moreover, when the tasks were assigned to the software engineers, they supposed that requirements engineers would issue the corresponding ticket afterward. Therefore, the software engineers had no choice but to enter "None" in the commit message when they committed updated source code to the VCS. However, the tickets were still unissued at the end of the project period.

We also asked both the requirements engineers and the software engineers about the nine commits out of 35 unlinked commits that had issued tickets but remained unlinked. The software engineers responded that the cause of unlinked commits was due to their own error. They failed to enter the corresponding ticket ID in the each "issued" ticket, although the requirements engineers had issued and assigned the tickets to them.

As described above, the requirements engineers created 102 tickets and did not issue 26 tickets during the two iterations. To answer RQ1 from Sect. 1, we show that 20% [26 unissued tickets/(102 issued tickets + 26 unissued tickets)] of all tickets were not issued during the project period.

In Step 4, the requirements engineers recovered 26 unissued tickets, which they identified in the previous step. After the recovery of the unissued tickets, we separately conducted interviews with both the requirements engineers and the end users. In the interviews, they evaluated whether the recovered tickets would have contained knowledge necessary for future development. We must note, however, the degree to which the knowledge would be necessary is subjective. The criteria for "necessary," both in these two projects, but also generally across all software projects are inherently subjective when estimating future development. Traditional methods err on the side of over documenting, whereas agile methods err on the side of under documenting. By asking the requirements engineers on the team, we are able to get the perspective of the engineers designing the system, which we believe to be the most "fair" approach in this circumstance.

Both the requirements engineers and the end users evaluated the contents of the recovered tickets for approximately one hour. The requirements engineers evaluated them from the viewpoint of the need for future software modifications, and the end users evaluated them from the viewpoint of the need for future software operations. The evaluation results are in the two sub-columns of Column 4 in Table 2 (labeled "For software modification (from REs)" and "For software operation (from Users)"). The requirements engineers responded that 23% of the recovered tickets (6/26) were required. The end users responded that 42% of those tickets (11/26) were required. Both groups agreed that four of the 26 unissued tickets contained knowledge that would be required in the future, and 13 of the 26 commits were identified by at least one of the two groups as containing knowledge that would be required in the future.

We also categorized the types of task descriptions on activities carried out by software engineers. As shown in the second column of Table 3, the requirements engineers and end users in this case study defined seven categories of changes: external function changes, business logic changes, user interface (system screen) changes, system property changes, bug fixing, source code refactoring, and

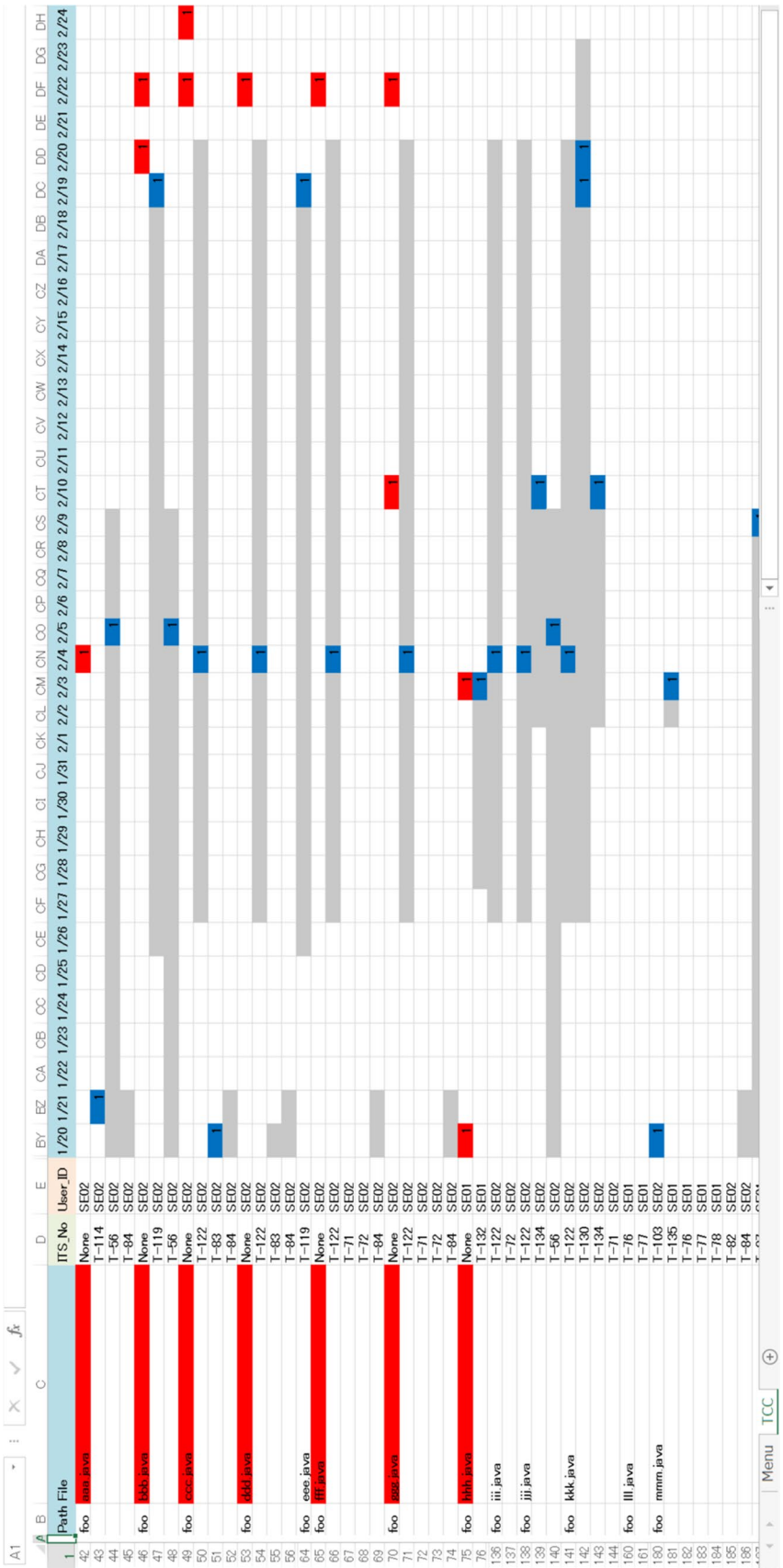

Fig. 11 First case generated TCC

Table 3 Detailed evaluations of unissued tickets (first case)

Type No.	Types of task descriptions	Number of unlinked commits	Number of unissued tickets	Number of tickets necessary for software modification	Number of tickets necessary for software operation	Examples of subjects described in recovered tickets
(1)	External function changes	6	4	1 [25% (=1/4)]	4 [80% (=4/4)]	Add function for displaying calculation result
(2)	Business logic changes	2	2	2 [100% (=2/2)]	2 [100% (=2/2)]	Modify calculation algorithms
(3)	User interface (system screen) changes	9	5	1 [20% (=1/5)]	5 [100% (=5/5)]	Change layout of input forms in system screens
(4)	System property changes	4	3	2 [66% (=2/3)]	0 [0% (=0/3)]	Add/remove parameters in initial file
(5)	Bug fixing	2	0	–	–	–
(6)	Source code refactoring	3	3	0 [0% (=0/3)]	0 [0% (=0/3)]	Create utility class for aggregating common methods
(7)	Development environment change	9	9	0 [0% (=0/9)]	0 [0% (=0/9)]	Rename brunch and tag Eliminate unreachable code (dead code)
	Total	35 unlinked of 159 total	26 [20% (=26/(102+25))]	6 [23% (=6/26)]	11 [42% (=11/26)]	

Table 4 Evaluation results (second case)

Type of commits	No. of commits	Step 3		Step 4			
		Corresponding tickets were issued or not (unissued)		Unissued tickets are needed or not			
				For software modification (from REs)		For software operation (from Users)	
Linked to tickets	712	–		–		–	
Not linked to tickets	48	Issued tickets	7	–		–	
		Unissued tickets	41	Needed	20	Needed	9
				Not	21	Not	32

development environment management. The first three are originally derived from meetings with end users, and the last four are derived from daily developer meetings.

The fifth and sixth columns in the table detail the evaluations conducted by requirements engineers and end users, respectively. The right-most column gives examples of the subjects described in the recovered tickets.

From their detailed evaluations, the end users determined that the knowledge on both external function change (Type 1) and user interface change (Type 3) was necessary, although the requirements engineers did not. However, the requirements engineers evaluated the knowledge on system property change (Type 4) as necessary for forthcoming software modification, although the end users did not think this as necessary. Both actors agreed that knowledge on source code refactoring (Type 6) and development environment change is not necessary for software modification and operation. To answer RQ2 from Sect. 1, our study revealed that 41% of unissued tickets contain knowledge necessary for continued software operation and 22% of unissued tickets contain knowledge necessary for future software modifications. Half of all unissued tickets (13/27) contain knowledge necessary for either continued operation or future modification.

5.3.2 Second case results

We collected the 296 tickets from the ITS, and 159 commit logs from the VCS in Step 1. As shown in two left side columns in Table 4, the numbers of linked and unlinked commits during two iterations were 712 and 48, respectively.

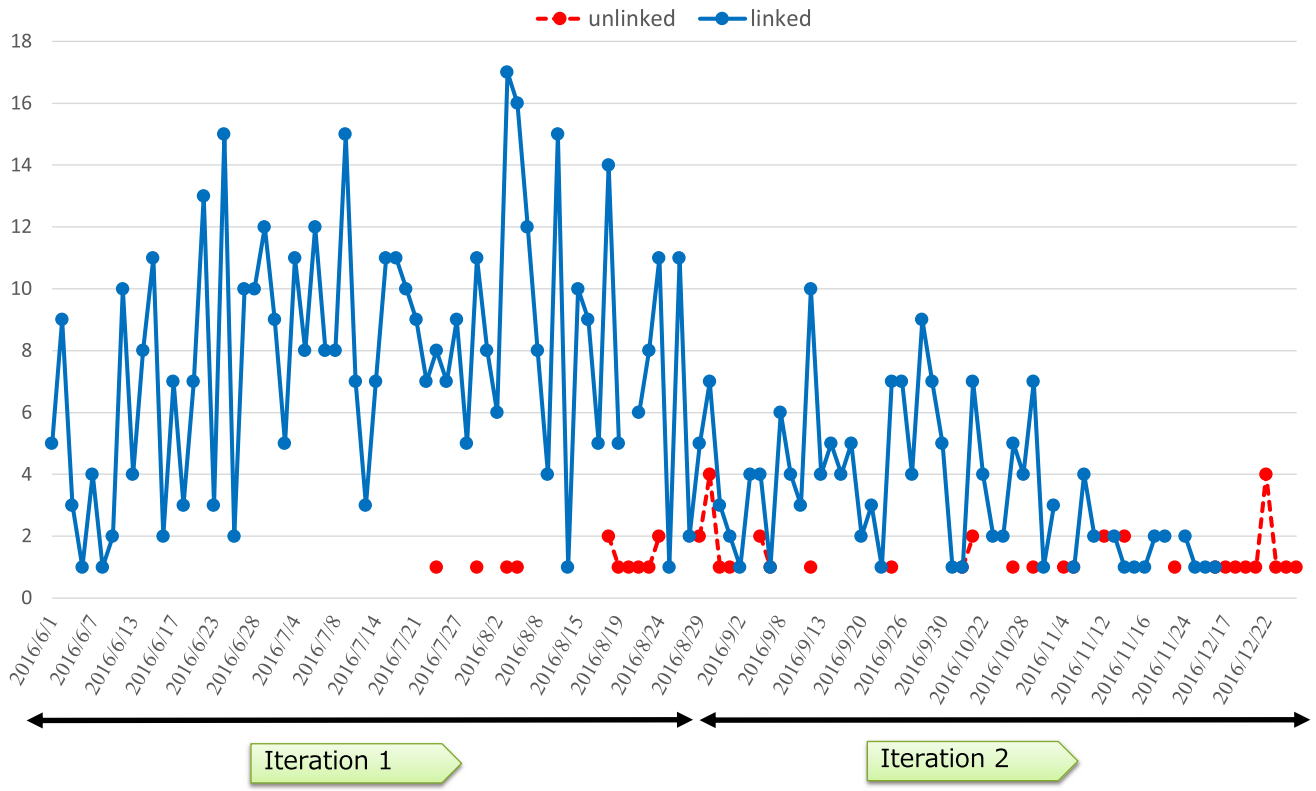

Fig. 12 Second case: number of commits per day

Figure 12 shows the time-series data on the number of commits per day. As shown in the figure, unlinked commits start to appear in the later part of the iteration 1. Moreover, unlinked commits continue to appear during iteration 2, increasing slightly near the end of that iteration. This is not surprising given that when software deadlines loom, documentation is often not viewed as a mission-critical task but rather as a task that slows engineers down from delivering operational software in a timely manner. Regardless, the increase in unlinked commits suggests that by this time the engineers were perhaps spending resources on source code that were, earlier in the iteration, spent on documentation.

In Step 2, the TCC was generated from the two data by our TCC generator. However, because the TCC generated in the second case is much larger and more detailed than that of the first case study, it has been omitted.

In Step 3, both the requirements engineers and the software engineers examined the activities of the 48 unlinked commits for approximately half day. This took place about one month after the completion of the development of the application.

From the contents of 48 unlinked commits, 41 "unissued" tickets were identified as shown in the third column of Table 4. As a result, we found that in 5% (41/760) of all commits, five software engineers committed source code to the VCS without entering corresponding ticket IDs. The requirements engineers created 296 tickets and did not issue 41 tickets during the two iterations. To answer RQ1 from Sect. 1, our study revealed that 12% [41 unissued tickets/(296 issued tickets + 41 unissued tickets)] of all tickets were not issued during the project period of the second case.

In Step 4, the requirements engineers recovered 41 unissued tickets identified in Step 3. After the recovery of the unissued tickets, as in the first case, those "recovered" tickets were evaluated from the viewpoints of the needs for future development, namely future software modification and operation. The evaluation results are shown in the fourth and fifth columns in Table 4. To answer RQ2 from Sect. 1, our results indicate that 49% of the recovered tickets (20/41) were required for future software modification and 22% of those tickets (9/41) were required for future software operation.

As in the first case, we detailed the evaluations as shown in Table 5. We also used the same seven types of task descriptions used in the first case. As shown in the fourth column, the implementation tasks of unissued tickets mapped to four categories: Types 3, 4, 6, and 7.

The knowledge on user interface change (Type 3) was needed for both software modification and operation. From the viewpoint of software operation, the task descriptions of source code refactoring (Type 6) were not needed. On the

Table 5 Detailed evaluations of unissued tickets (Second Case)

Type no.	Types of task descriptions	Number of unlinked commits	Number of unissued tickets	Number of tickets necessary for software modification	Number of tickets necessary for software operation
(1)	External function changes	0	–	–	–
(2)	Business logic changes	0	–	–	–
(3)	User interface (system screen) changes	10	9	9 [20% (=9/9)]	9 [100% (=9/9)]
(4)	System property changes	22	16	0[0% (=0/16)]	0 [0% (=0/16)]
(5)	Bug fixing	0	–	–	–
(6)	Source code refactoring	11	11	11 [100% (=11/11)]	0 [0% (=0/3)]
(7)	Development environment change	5	5	0 [0% (=0/5)]	0 [0% (=0/5)]
	Total	48 unlinked of 760 total	41 [12% (=41/(296+41)]	20 [49% (=20/41)]	9 [22% (=9/41)]

other hand, the engineers valued the knowledge as necessary for future software modification. The engineers determined the knowledge on system property changes (Type 4) was not required. The tasks on development environment change (Type 7) were not necessary for software modification and operation.

6 Discussion

In this section, we discuss the implications of our findings and its possible applications for practitioners.

6.1 Identifying undocumented knowledge

Based on the results of the case studies, we found two reasons software engineers committed source code to the VCS without linking the commit to a ticket ID. The first is that the software engineers simply failed to enter the corresponding ticket ID in the commit message. The second reason is that requirements engineers did not issue a corresponding ticket for the requested feature. We argue that unissued tickets (i.e., the second reason) may cause serious problems. As we mentioned before, in both cases, requirements engineers and software engineers hold a retrospective meeting to maintain the software document just before delivering the software product at the end of the second iteration. In that meeting, the engineers using the TCC found about 22% of the unissued tickets in the first case and 49% of the unissued tickets in the second case to be critical for future software modification (i.e., adding new features). Similarly, approximately 41% of the unissued tickets in the first case and 22% of the unissued tickets in the second case were found to be crucial documentation for future software operation (i.e., maintaining current features). We believe use of our TCC contributes to detecting this type of undocumented knowledge.

We do not claim that requirements engineers should always issue tickets that record all software engineer's tasks in a project. If so, documentation (i.e., describing and managing tickets) will become a heavy burden on requirements engineers and software engineers in an agile environment. The project might not benefit from the advantages of agile software development approaches (e.g., flexibility, quick feedback). We suggest that it is better that project members examine their activities using our TCC or any similar visualization tool during their retrospective meetings. In these meetings, they could examine the unissued tickets required for future software modifications or operations and document any relevant undocumented knowledge.

6.2 Understanding unrecorded activities

In agile software development projects, the number of tickets issued in past projects has been used as reference information of cost estimation for new projects [6]. In the first case, we found 34 unissued tickets and recovered 27 of them. Before that, the requirements engineers had already issued 102 tickets. As a result, 129 tickets (102+27) were finally documented. To answer the first research question, we found that about 21% of all tickets (27/129) had not been recorded in the ITS. Similarly, in the second case, we found that about 12% [41/(296+41)] of all tickets had not been issued in the ITS.

If the original numbers of tickets (i.e., 102 tickets for the first case and 296 tickets for the second) were used as reference information for estimating future software modification costs, then there might be differences in actual and estimated costs of more than 20 and 12% for the first case and the second case, respectively.

To accurately estimate software development costs and avoid costs identified by Mendes et al. [14], we should understand the actual activities of past projects, which we aim to use as reference information. Our approach supports

understanding actual work activities to estimate software development cost for forthcoming projects.

6.3 Timing of visualizations and evaluations

Timing might be a decisive factor for enhancing the effectiveness of our visualization approach. In our case studies, actors in both projects evaluated and responded regarding the unissued tickets within 1 month (i.e., 3 weeks for the first case and 1 month for the second case) after completion of the development. During the evaluations, their memory of the project remained fresh in their minds. If the evaluation had been much later, it would have been more difficult for them to identify and recover the unissued tickets.

Once the project is completed, most project members (i.e., requirements engineers and software engineers) will be assigned to a new project. If there are no "original" project members, detecting unissued tickets and documenting previously undocumented knowledge may be impossible, even if a TCC is generated.

6.4 The effect of undocumented knowledge

Although we cannot completely characterize the effect of undocumented knowledge based on these two cases in our case study, our results suggest that undocumented knowledge in agile software projects is an important long-term challenge. We believe this is particularly important for undocumented knowledge from the second case, which involved nearly six times the number of commits as the first case. In both of our cases, the unissued tickets had an outsized impact in terms of their effect on future software modification and operations. Between roughly one-fifth and two-fifths of the unissued tickets resulted in a situation where future development could have been stalled unnecessarily.

If our findings could be generalized as a result of additional case studies, then perhaps the most important result would be that we can better understand the actual trade-off being made when switching from traditional methods to agile methods. Agile software development proponents believe that documentation is too heavily prioritized in traditional approaches [4]. Advocates of traditional approaches favor documentation as a means of preventing possible confusion [15, 22]. In a perfect software development methodology, engineers would only create documentation that would definitely be referenced in the future, an impossibility in the real world which requires trade-offs made on imperfect information.

Exploratory research is needed for traditional software methodologies to understand how much unused documentation is created. This needlessly created documentation is the other side of the trade-off. Such a study would produce the most direct comparison to our finding that between roughly 20 and 40% of unissued tickets result in a need for additional documentation. If both aspects can be better quantified, then both agile and traditional methods could be improved. In addition, we may be able to provide better guidance to software engineers about what must be documented and where documentation may not be as useful.

The second case had a lower percentage of unissued tickets despite having a far larger number of commits. Perhaps this is a result of all team members having prior experience with agile software methods. It may also be a result of this project being developed for external stakeholders rather than internal stakeholders as in the first case. Unfortunately, we were unable to justify a deeper examination of this. However, we believe it suggests that experience and context play significant roles in the impact of adopting agile methods on documentation.

7 Case-study limitations

When designing any case study, including exploratory case studies, care should be taken to mitigate threats to validity. This paper describes an exploratory case study seeking to determine: (1) the extent to which undocumented knowledge exists in agile software development projects and (2) the extent to which this may cause problems for future development. We evaluate this case study using both quantitative data (e.g., unlinked tickets) and qualitative data (e.g., determinations regarding the future utility of an unlinked ticket). These data represent our units of observation, which should not be confused with our unit of analysis: documented knowledge.

Some may question whether this work is explanatory rather than exploratory in nature. Runeson and Höst describe exploratory work as "finding out what is happening, seeking new insights and generating ideas and hypotheses for new research" [23]. In contrast, our goal in this work is to determine whether undocumented knowledge exists in agile software development and whether it may affect future development. Runeson and Höst describe explanatory research as "seeking an explanation of a situation or a problem, mostly but not necessary in the form of a causal relationship" [23]. Because we are not investigating why undocumented knowledge occurs or attempting to establish and define any related causal relationships, this work remains exploratory in nature. Runeson and Höst also characterize case studies, rather than experiments, as typically exploratory in nature, as commonly using both quantitative and qualitative data, and as using explicit research questions from the outset [23]. The presence of research questions, quantitative data, or qualitative data is fully consistent with exploratory research and does not indicate, either on their own or collectively, that this is explanatory work.

Construct validity addresses the degree to which a case study aligns with the theoretical concepts used. Three ways to reinforce construct validity are using multiple sources of reliable evidence, establishing a chain of evidence, and having key informants review draft case-study reports [34]. We used two projects in the case study. We collected 158 and 760 commit logs, and 102 and 296 tickets from two iterations from projects of the first case and the second case, respectively. The length of iterations of the first case is 2 months. For the second case, this is 7 months. Three different types of actors (i.e., requirements engineers, software engineers, and end users) were engaged in both projects. To establish a chain of evidence, we used the supporting tools (i.e., ITS and VCS) to maintain a record of all data of our studies. Finally, other engineers of NTT Group's Agile Professional Center [17] reviewed our draft case-study report.

Our interview at the end of each case produced another important limitation for construct validity. We asked requirements engineers to identify whether unlinked commits would be needed for future modification of the software and end users to identify whether those same unlinked commits would be needed to maintain future operation of the software. Essentially, we filled out a "yes" or "no" for each unlinked commit with each group to their respective questions. We did not provide heuristics, checklists, or guidelines to either group to help them make this determination. Therefore, the participants in this interview may have had differing subjective expectations related to answering their questions.

In this exploratory case study, we make no causal inferences. That is, we do not attempt to definitively determine why the tickets were unlinked; we simply seek to discover how many were unlinked and qualitatively characterize them as impactful for future development or use. Therefore, internal validity is not a concern.

External validity is the ability of a case study's findings to generalize to broader populations [34]. A possible threat to external validity is the fact that we analyzed only two projects. In addition, we do not examine the consequences of these gaps. It is possible that, although our interview subjects felt the unlinked commits were problematic, other groups would not find similar unlinked commits to be problematic for future modification or operation of the software. That said, our visualization approach (i.e., TCC) is not domain specific and would work for any project using the commit log procedure described herein and used widely in industry. We used an ITS and VCS in the approach, both of which are commonly used in agile software development projects. We also used the standard data formats of both ITS and VCS. Our TCC generator does not require changes to these data formats. We believe these facts reinforce the external validity of our case.

Reliability is the ability to repeat a study and observe similar results [34]. In this work, we use data collection and analysis procedures that are as similar as possible in both cases to mitigate the threat of reliability and ensure comparisons between the two cases remain valid. A complete discussion of the differences can be found in Sects. 5.1 and 5.2. This decision does limit our ability to explain new phenomena. For example, we could have altered our procedures in the second case based on things we learned in the first case to better explain our results. To do so, we would need to have examined another smaller project more similar to the first case. This would also have changed the nature of our study from exploratory to explanatory. However, we chose instead to extend our exploratory goals from an internally used, smaller project (i.e., the first case) to an externally used, larger project (i.e., the second case). To further reinforce our study's reliability, the ITS and VCS used for the case studies have no specific features. Both are open-source software. We also developed a tool that generates the TCC. This tool enabled the automatic carrying out of Step 2 of the cases. By using open-source software and the tool, other researchers and study participants will be able to follow the steps of the case study rigorously.

8 Summary and future work

Agile software development is popular and widely used, but the effects of an agile approach to documentation are not well understood. In this paper, we examine undocumented knowledge—information that could have been documented, but was not—in two industrial cases using agile software methods.

Our study revealed that software engineers committed source code to the VCS without entering the corresponding ticket IDs in 17 and 5% of all commits in the first case and the second case, respectively. The unissued tickets accounted for 21% of all tickets in the first case and 12% of those in the second case. The end users and requirements engineers evaluated the contents of the unissued tickets, and we found that 42% of these tickets in the first case and 22% of those in the second case were required for end users in future software operation. Moreover, 22% of the unissued tickets in the first case and 49% of those in the second case were required for engineers in future software modification. These results suggest that undocumented knowledge in agile software development is a non-trivial problem that will compound over time.

We plan to design and develop a tool to predict the occurrence of unissued tickets through real-time monitoring and visualization of the activities of the ITS and VCS. The goal of this tool is to prevent undocumented knowledge due to human error and mitigate the effort expended during retrospective meetings to uncover these errors.

Acknowledgements The authors are grateful to Messrs. Takashi Hoshino, Keiichiro Horikawa, Daisuke Hamuro, and Masayuki Inoue of NTT, Tetsuo Kobashi, Masatoshi Hiraoka and Hironori Shibayama of NTT DATA, Dr. Noriaki Izumi and Motoi Yamane at Piecemeal Technology for their assistance in the case study. We also wish to thank the anonymous referees of both the Requirements Engineering Conference and this special issue of the Requirements Engineering Journal for their feedback.

References

1. Apache Subversion. Enterprise-class centralized version control for the masses. https://subversion.apache.org/. Accessed 03 Jan 2017
2. Backlog. Online project management tool for developers. https://backlog.com/. Accessed 03 Jan 2017
3. Beck K, Cynthia A (2005) Extreme programming explained: embrace change. Addison-Wesley Professional, Boston
4. Beck K, Grenning J, Martin RC, Beedle M, Highsmith J, Mellor S, van Bennekum A, Hunt A, Schwaber K, Cockburn A, Jefferies R, Sutherland J, Cunningham W, Kern J, Thomas D, Fowler M, Marick B (2001) Manifesto for agile software development. http://agilemanifesto.org/. Accessed 10 Jan 2017
5. Cockburn A (2006) Agile software development: the cooperative game, 2nd edn. Addison-Wesley Professional, Boston
6. Cohn M (2005) Agile estimating and planning. Prentice Hall, Upper Saddle River
7. Dhungana D, Grünbacher P, Rabiser R (2011) The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Autom Softw Eng 18(1):77–114
8. Dorairaj S, Noble J, Malik P (2012) Knowledge management in distributed agile software development. In: The Agile conference, pp 64–73
9. Eisenhardt KM (1989) Building theories from case study research. Acad Manag Rev 14(4):532–550
10. GitHub. The world's leading software development platform. https://github.com/. Accessed 03 Jan 2017
11. Hoda R, Noble J, Marshall S (2010) How much is just enough?: Some documentation patterns on agile projects. In: Proceedings of the 15th European conference on pattern languages of programs, New York, NY, USA, pp 13:1–13:13
12. Lanza M, D'Ambros M, Bacchelli A, Hattori L, Rigotti F (2013) Manhattan: supporting real-time visual team activity awareness. In: Proceedings of the 21st international conference on program comprehension, pp 207–210
13. Levy M, Hazzan O (2009) Knowledge management in practice: the case of agile software development. IN: Proceedings of the 2009 ICSE workshops (CHASE'09), pp 60–65
14. Maalej W, Thurimella AK (2015) Managing requirements knowledge. Springer, Berlin
15. Martin RC (2003) Agile software development: principles, patterns, and practices. Prentice Hall, Upper Saddle River
16. Mendes TS, de F. Farias MA, Mendonça M, Soares HF, Kalinowski M, Spínola RO (2016) Impacts of agile requirements documentation debt on software projects: a retrospective study. In: Proceedings of the 31st annual ACM symposium on applied computing, pp 1290–1295
17. Moe NB, Faegri TE, Cruzes DS, Faugstad JE (2016) Enabling knowledge sharing in agile virtual teams. In: Proceedings of the 11th international conference on global software engineering, pp 29–34
18. NTT DATA. NTT DATA: Global IT Innovator. http://www.nttdata.com/global/en/. Accessed 10 Jan 2017
19. NTT R&D. NTT Research & Development. http://www.ntt.co.jp/RD/e/index.html. Accessed 10 Jan 2017
20. Palmer SR, Felsing JM (2002) A practical guide to feature driven development. Prentice Hall, Upper Saddle River
21. Redmine. Flexible project management. https://redmine.org/. Accessed 03 Jan 2017
22. Rubin KS (2012) Essential Scrum: a practical guide to the most common agile process. Addison-Wesley Professional, Boston
23. Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. EmpirSoftw Eng 14(2):131–165
24. Saito S, Iimura Y, Takahashi K, Massey AK, Antón AI (2014) Tracking Requirements evolution by using issue tickets: a case study of a document management and approval system. In: 36th international conference on software engineering, pp 245–254
25. Saito S, Iimura Y, Tashiro H, Massey AK, Antón AI (2016) Visualizing the effects of requirements evolution. In: 38th International conference on software engineering, pp 152–161
26. Saito S, Iimura Y, Massey AK, Antón AI (2017) How much undocumented knowledge is there in agile software development?: Case study on industrial project using issue tracking system and version control system. In: Proceedings of the 25th international conference on requirements engineering, pp 194–203
27. Schwaber K, Beedle M (2001) Agile software development with Scrum. Prentice Hall, Upper Saddle River
28. Stettina CJ, Heijstek W (2011) Necessary and neglected?: An empirical study of internal documentation in agile software development teams. In: Proceedings of the 29th ACM international conference on design of communication, pp 159–166
29. Trac. Integrated SCM & project management. https://trac.edgewall.org/. Accessed 03 Jan 2017
30. Thurimella AK, Bruegge B, Janzen D (2017) Variability Plug-Ins for Requirements Tools: a Case-Based Theory Building Approach. IEEE Syst J 11(4):1935–1946
31. Thurimella AK, Schubanz M, Pleuss A, Botterweck G (2017) Guidelines for managing requirement rationales. IEEE Softw 34(1):82–90
32. Voigt S, von Garrel J, Müller J, Wirth D (2016) A study of documentation in agile software projects. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement, pp 4:1–4:6
33. Wnuk K, Regnell B, Karlsson L (2009) Feature transition charts for visualization of cross-project scope evolution in large-scale requirements engineering for product lines. In: Proceedings of the 4th international workshop on requirements engineering visualization, pp 11–20
34. Yin RK (2003) Case study research: design and methods, vol 5, 3rd edn. Sage, Thousand Oaks