2017 IEEE 25th International Requirements Engineering Conference

How Much Undocumented Knowledge is there in Agile Software Development?

Case Study on Industrial Project using Issue Tracking System and Version Control System

Shinobu Saito, Yukako Iimura
Software Innovation Center
NTT Corporation
Tokyo, Japan
{saito.shinobu,
iimura.yukako}@lab.ntt.co.jp

Aaron K. Massey
Department of Information Systems
University of Maryland,
Baltimore County,
Baltimore, MD, USA
akmassey@umbc.edu

Annie I. Antón
School of Interactive Computing
Georgia Institute of Technology,
Atlanta, GA, USA
aianton@gatech.edu

***Abstract*—In agile software development projects, software engineers prioritize implementation over documentation to eliminate needless documentation. Is the cost of missing documentation greater than the cost of producing unnecessary or unused documentation? Even without these documents, software engineers maintain other software artifacts, such as tickets in an issue tracking system (ITS) or source code committed to a version control system (VCS). Do these artifacts contain the necessary knowledge? In this paper, we examine undocumented knowledge in an agile software development project at NTT. For our study, we collected 159 commit logs in a VCS and 102 tickets in the ITS from the three-month period of the project. We propose 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 study revealed that in 16% of all commits, software engineers committed source code to the VCS without a corresponding issue ticket in the ITS. Had these commits been based on individual issue tickets, these "unissued" tickets would have accounted for 20% of all tickets. Software users and requirements engineers also evaluated the contents of these commits and found that 42% of the "unissued" tickets were required for software operation and 23% of those were required for requirements modification.**

Keywords-Agile Agile Development; Issue Tracking System; Version Control System; Requirements Knowledge

I. INTRODUCTION

Agile software development is now an established approach to building software systems [10, 14, 18]. 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. They put more emphasis on the software implementation (i.e., working software) than its documents [9]. In this manner, software developers can respond flexibly to stakeholder needs.

In theory, agile software development generates only the minimally necessary software documentation. During 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 [18] 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) [3]. Software projects must maintain knowledge required by "current" software users and "future" project engineers. Users might need to know the specifications and constraints of the software product. Future project members 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 standard agile approaches properly document the knowledge required for these stakeholder groups.

Herein, we describe a study conducted to examine an agile software development project at NTT Laboratories [13]. NTT is a global service and network provider. NTT Laboratories have more than 3,000 researchers both in Japan and abroad. Approximately 1,000 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, etc.) 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.

Software 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, and it can take several

2332-6441/17 $31.00 © 2017 IEEE
DOI 10.1109/RE.2017.33

forms, including: software specifications, constraints, architecture, features, and rationales.

Issue tracking systems (ITS) and version control systems (VCS) are commonly used in agile software development projects [6, 15, 19, 20]. 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 engineer creates issue tickets using the ITS and 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, leading 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." As the number of unissued tickets increases, we expect that the knowledge required for future operation or modification may be undocumented. In this context, operation refers to use of the software system according to its existing goals and features, and modification refers to the addition of new features or functions.

In this paper, we propose a ticket-commit network chart (TCC) which links ticket data in the ITS with the commit log data in the VCS. The chart 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 within NTT's agile software development project:

RQ1: How many unissued tickets are created in NTT's agile software development project?

RQ2: How much undocumented knowledge is created by unissued tickets that may be required later for future operation or modification?

We seek to answer RQ1 to measure 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 seek to answer RQ2 to understand how often those changes affect future development. Both questions address the extent to which documentation in standard agile projects is insufficient for future development.

The remainder of this paper is organized as follows: Section II describes related work with an emphasis on visualization of development activities and knowledge management within agile software development. Section III provides an overview the agile software development project we examined and evaluated. Section IV introduces our approach for visualizing agile software development activities. Section V presents our empirical analysis procedure and results from the case study. Section VI describes the limitations of the case study. Section VII discusses the implications of our findings, and Section VIII summarizes the paper and presents future work.

II. Related Research

A. Visualization of Software Development Activities

Visualization of software development is a common challenge in software engineering. Wnuk et al. [22] 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 at al. [7] developed a real-time visualization tool, called Manhattan, for team activity within software development. This tool, built as an Eclipse plugin, 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 [16], 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 [15]. We found that new requirements engineers using the REC could identify artifacts affected by change requests more accurately and quickly than requirements engineers attempting the same task without the REC. Our approach therein focused only on the tickets in an ITS for visualization of activities of requirements engineers. Our approach in the research presented herein 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.

B. Knowledge Management for Agile Software Development

Knowledge management in software engineering is crucial because software development is a knowledge-intensive activity. Herein, we focus on knowledge management in agile software development.

Moe at al. [11] 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 [8] 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). Herein, we introduce a visualization approach for recovering undocumented knowledge that has similar coordination benefits.

Dorairaj et al. [6] 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. [21] 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.

III. PROJECT OVERVIEW

A. Project Description and Schedule

The software product for this case 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. Phase durations are three weeks, four weeks, and four weeks, respectively. There are three roles: software user, requirements engineer, and software engineer. Personnel filling each role number four, two, and two respectively.

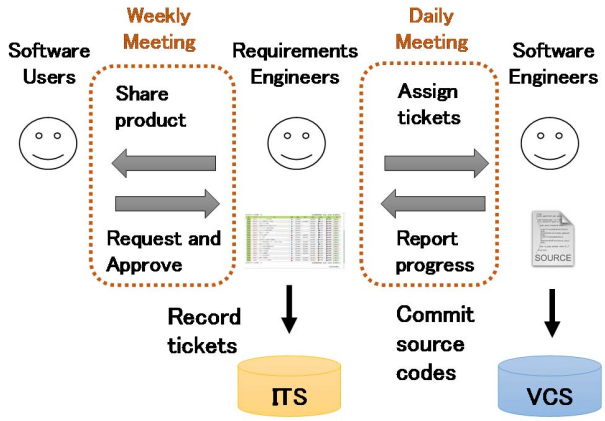

Figure 1. Actors, their communications, and tools in iteration phases

In the first half of the planning phase, requirements engineers elicit the product requirements from the software users. The requirements engineers create user stories for the software product. They also define algorithms for specific features, and design user interface sketches. Each user story is broken down into a set of implementation tasks for the next iteration phase.

B. Daily and Weekly Meetings

In the iteration phases, as shown in the Figure 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. 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 [2] wherein requirements engineers hold a weekly meeting to share the product with the software users. The users examine and use the product. When they approve of the features developed, the requirements engineers close the

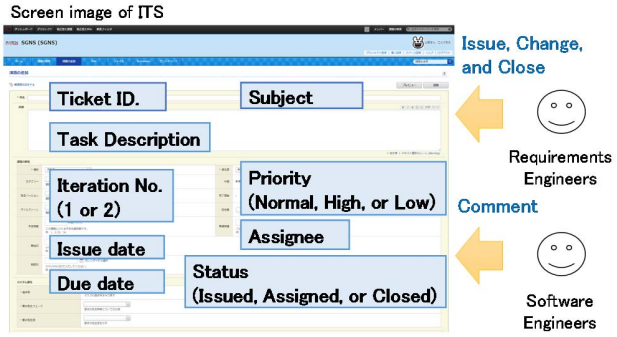

Figure 2. Recording items in ticket

corresponding tickets. However, software 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.

Finally, just before delivery of the software product, requirements engineers and software engineers hold retrospective meetings [18] together to ensure software documents, including tickets, are properly maintained.

C. Issue Tracking System and Version Control System

For our study, we examined the development activities during the two iteration phases using data from our ITS and VCS. Backlog [1] is a web-based application on public cloud services, and it was selected as the ITS for this project. Figure 2 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 [19] as the VCS. A server for the VCS was deployed over a private LAN. During the iteration phases, two software engineers were responsible for committing updated source code to the VCS.

In this project, 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 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".

IV. Visualization of Agile Development Activities

A. 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.

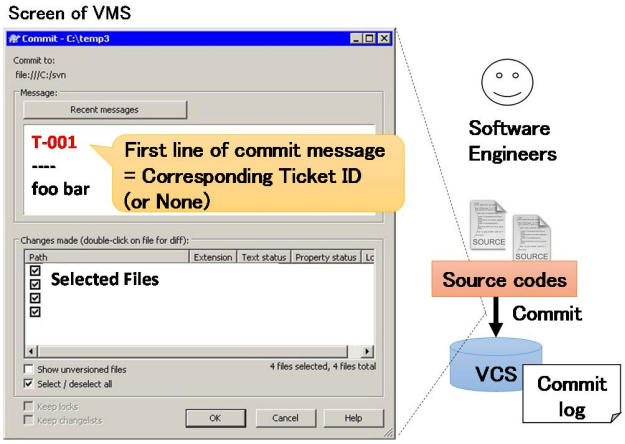

Figure 3. Entering corresponding ticket ID in commit message

Figure 3 shows the dialog box of the commit message. Based on the guidelines, a software engineers 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 4 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 of Figure 4, respectively. Each commit includes a commit message, a user ID, a commit date, and the path and file name(s) 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

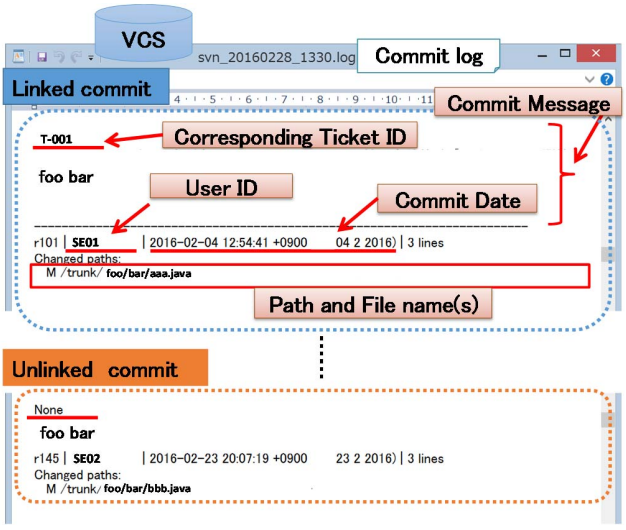

Figure 4. Commit Log (linked commit and unlinked commit)

unlinked commit.

Figure 5 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.

B. *Ticket-Commit Network Chart*

We introduce our TCC which provides a visual representation of time-series commit activities as described in the VCS and the ITS. 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 Figure 6, the TCC visualizes both lifespans 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 lifespan 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 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, etc.).

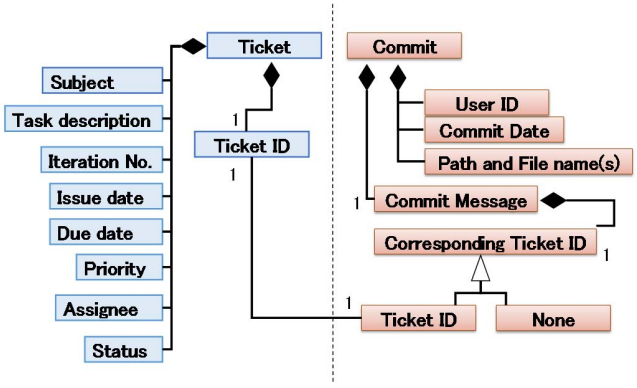

Figure 5. Meta-model of information on ticket and commit

Let's walk through an example. Consider the fourth line of the figure. Ticket "T_003" was issued on 9/2 then closed on 9/7. The lifespan of the ticket was six days. As shown in the bottom part of Figure 6, the issue date and close date of each ticket of the ITS refers to the lifespan. 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 (fifth line), the two red cells indicate the occurrences of the commits not linked to the corresponding tickets. "None" and "SE02" appears 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

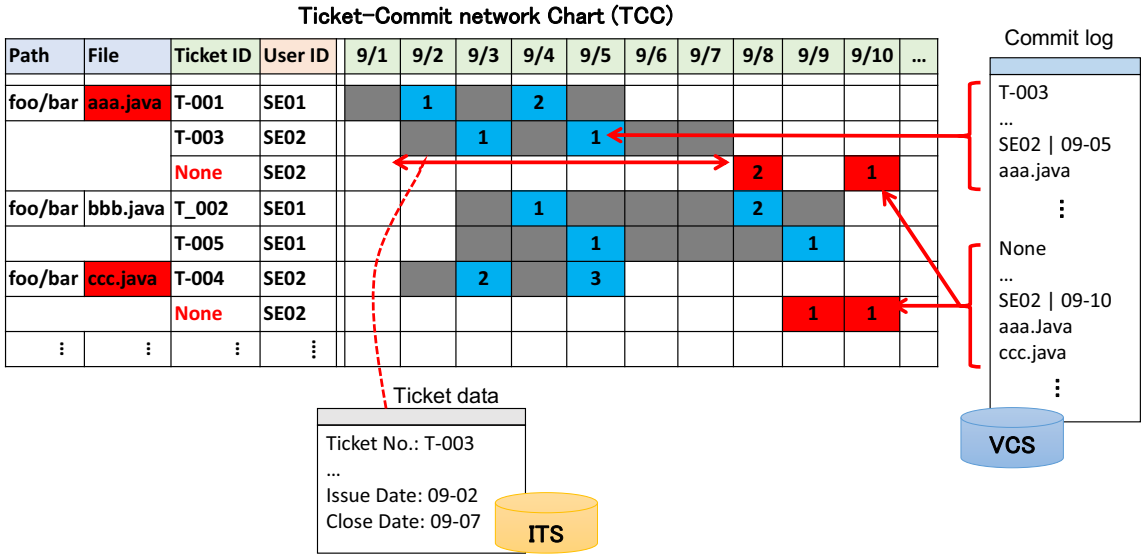

Figure 6. Ticket commit network chart (TCC)

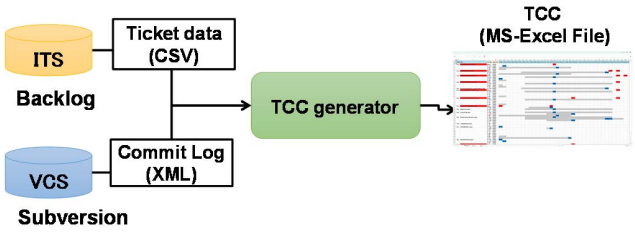

Figure 7. TCC generator

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. fifth and ninth lines) include red cells.

As shown on the right side of Figure 6, the commit logs in the VCS are denoted as red cells. From the VCS, as shown in Figure 4, 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 Figure 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.

C. Software Implementation

We implemented a tool to automatically generate the TCC. Figure 7 provides an overview of the TCC generator. From two inputs, ticket data (CSV file) exported from Backlog, and commit log (XML file) exported from Subversion, 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 the case study.

TABLE I. ARTIFACTS OF PROJECT

Phase	Type	Artifacts	Volume
Planning/ Inception	Documents	User Stories	30 pages
		Algorithms and User Interface Sketches	15 pages
Iterations 1-2	Source code (Java)	Main Programs	5,931 steps
		Test Programs	2,093 steps
	Documents	Tickets	102 tickets
		User Manual	54 pages

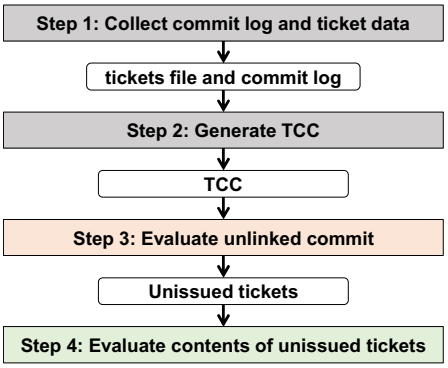

Figure 8. Analysis procedure (four steps) in case study

V. CASE STUDY

To answer the research questions given in Section I, we conducted a case study on an agile software development project at NTT Laboratories.

A. Data Collection and Participants

Table I lists the artifacts of the project, which were created during the 11-week development period. In this case, 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 software 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 program of line size of the software was approximately 8,000 lines of code (LOC), about 6,000 LOC main program and 2,000 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 (six years for one and eight years for the other). Both software engineers have two years of experience in agile development.

B. Analysis Procedure

Figure 8 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 examine the unlinked commits found in the TCC (i.e., red cells). The goal of this process is to identify any unissued tickets. After identification, we answer the first research question (RQ1: How many unissued tickets are created in NTT's agile software development project?).

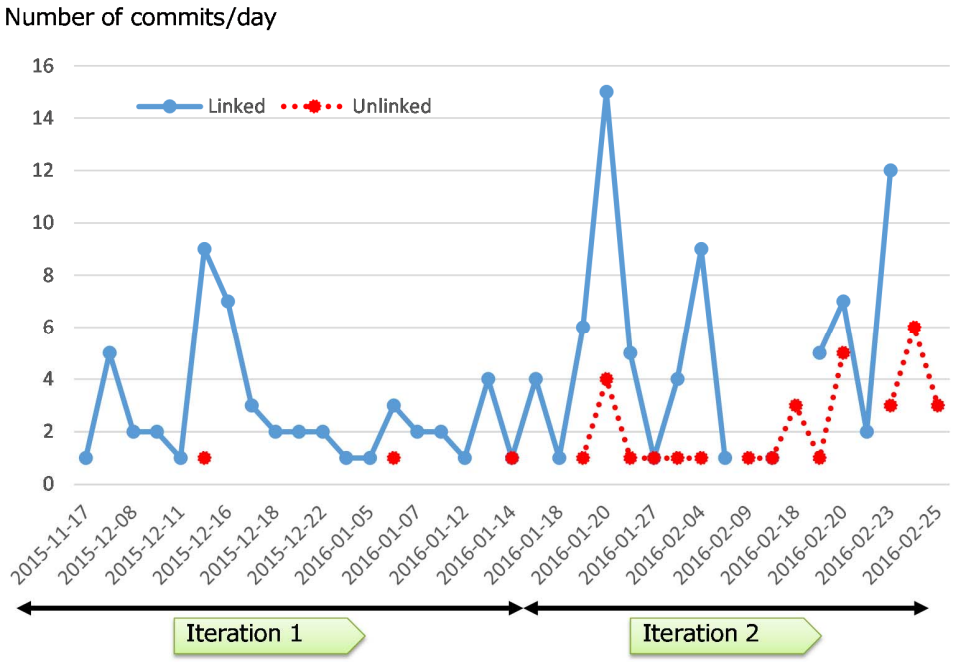

Figure 9. Time series data on number of commits per day

In the final step, the requirements engineers try to recover the unissued tickets. The requirements engineers and software users evaluate the contents of the issued tickets and check whether these tickets contain the required knowledge. The requirements engineers evaluate these recovered tickets in terms of future software modification, and the software users evaluate these tickets in terms of future software operation. Based on their evaluation results, we can answer the second research question (RQ2: How much undocumented knowledge is created by unissued tickets and later required for future operation or modification?).

C. Results

1) Visualization of Agile Software Development

In step 1, we collected the 102 tickets from the ITS, and 159 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 II.

Figure 9 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.

In Step 2, we generated the TCC using the TCC generator from those data. Figure 10 shows the image of the TCC generated for this case study.

2) Answers to Research Questions

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

As shown in the third column of Table II, 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.

We interviewed both groups of engineers regarding 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 afterwards. 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 unissued tickets left undone to the last time of the project period.

TABLE II. EVALUATION RESULTS

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

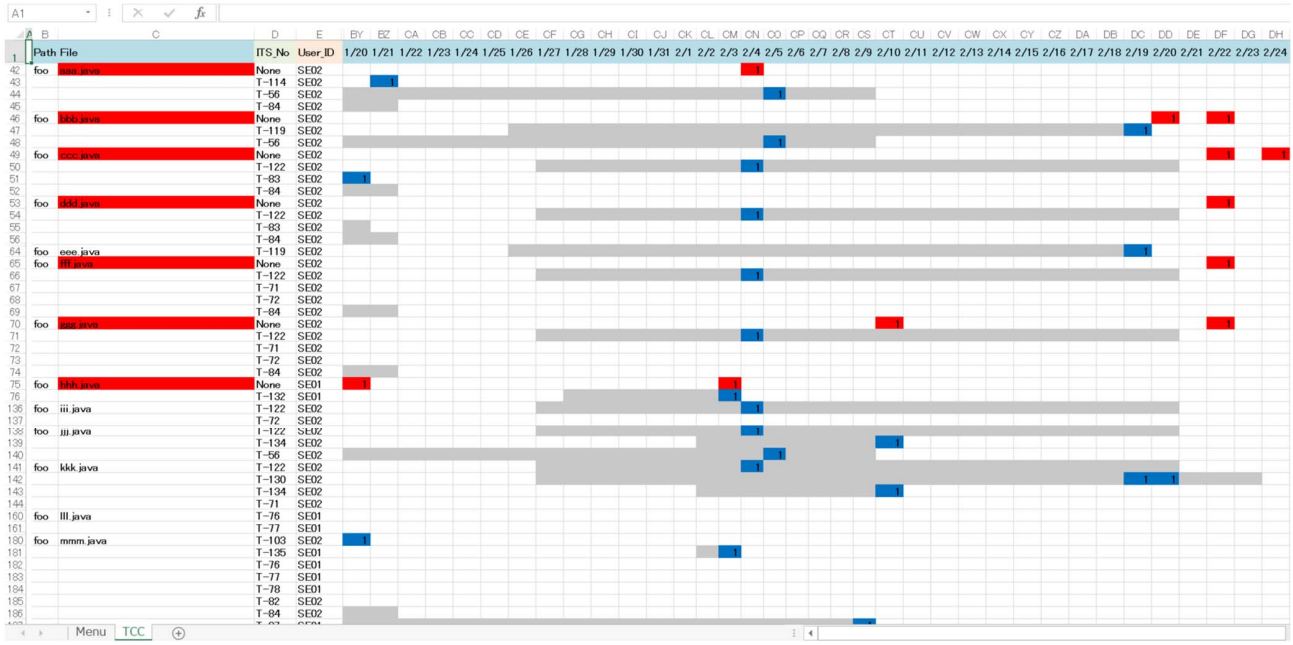

Figure 10. TCC generated in the case study

We also interviewed both groups regarding the 35 unlinked commits. The software engineers responded that nine of the unlinked commits should have been linked, but they failed to enter the corresponding ticket ID for the 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 Section 1, our study revealed 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 requirement 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 groups of engineers. In the interviews, they evaluated whether the contents of the recovered tickets were necessary knowledge for future development.

Both groups of engineers 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 software users evaluated them from the viewpoint of the

TABLE III. DETAILED EVALUATIONS OF UNISSUED TICKETS

Type No.	Types of task descriptions	No. of Not linked Commits	No. of unissued tickets	No. of tickets necessary for software modification	No. of tickets necessary for software operation	Examples of subjects described in recovered tickets
(1)	External function changes	6	4	1 [25% (=1/4)]	4 [100% (=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	26 [20% (=26/(102+25)]	6 [23% (=6/26)]	11 [42% (=11/26)]	

need for future software operations. The evaluation results are in the fourth and fifth columns in Table II. The requirements engineers responded that 23% of the recovered tickets (= 6/26) were required. The software users responded that 42% of those tickets (= 11/26) were required.

We also categorized the types of task descriptions on activities carried out by software engineers. As shown in the second column of Table III, the requirements engineers and software engineers 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 development environment management. The first three are originally derived from meetings with software 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 software users respectively. The right-most column gives examples of the subjects described in the recovered tickets.

From their detailed evaluations, the software 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 software 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 Section 1, our study revealed that 42% of unissued tickets contain knowledge necessary for continued software operation. Our study also revealed that 23% of unissued tickets contain knowledge necessary for future software modifications. Half of all unissued tickets (13/26) contain knowledge necessary for either continued operation or future modification.

VI. Discussion

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

A. Identifying Undocumented Knowledge

Based on the results of the case study, we found two reasons software engineers committed source codes 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, requirements engineers and software engineers hold a retrospective meeting to maintain the software document just before delivering the software product. Over 40% of knowledge for software operations was undocumented before they evaluated the TCC generated within the case study. The generated TCC helped the requirements engineers recover the unissued tickets. Our approach contributes to detecting this type of undocumented knowledge.

We do not claim that requirements engineers should issue tickets that record all software engineer's tasks in a project. If so, documentation (i.e., describing/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 (i.e., requirement engineers and software engineers) examine their activities using our TCC during their review/retrospective meetings. In these meetings, they could examine the unissued tickets required for future software modifications or operations and document any relevant undocumented knowledge.

B. 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 [4]. In our case study, we found 34 unissued tickets and recovered 26 of them. Before that, the requirements engineers had already issued 102 tickets. As a result, 128 tickets (= 102+26) were finally documented. As an answer for the first research question, this means that about 20% of all tickets (= 26/128) had not been recorded in the ITS. If the original number of tickets (i.e., 102 tickets) is used as reference information for cost estimation for a future software modification, then there might be more than a difference of 20% in the amount between actual and estimated costs. To accurately estimate software development cost, 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.

C. Timing of Visualizations and Evaluations

Timing might be a decisive factor for enhancing the effectiveness of our visualization approach. In our case study, actors in the project evaluated and responded regarding the unissued tickets three weeks after completion of the development. During the evaluation, 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 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.

VII. Case Study Limitations

When designing any case study, care should be taken to mitigate threats to validity.

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 [23]. In this case study, we collected 159 commit logs and 102 tickets from two iterations over a total of two months with three different types of actors (requirements engineers, software engineers, and software users). However, we used only one source from one agile software development project. 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 study. Finally, other engineers of NTT Group's Agile Professional Center [12] reviewed our draft case-study report.

In this case study, we make no causal inferences, so internal validity is not a concern.

External validity is the ability of a case study's findings to generalize to broader populations [23]. A possible threat to external validity is the fact that we only analyzed one project. However, our visualization approach (i.e., TCC) is not domain specific and would work for any project using the commit log procedure described herein. 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 [23]. To reinforce our study's reliability, the ITS and VCS used for the case study 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 case. By using Open Source Software and the tool, other researchers and case-study participants will be able to follow the steps of the case study rigorously.

VIII. Conclusion and Future work

We proposed a TCC for visual representation of time-series source code commit activities. Using the TCC, we could determine how many commits are linked or not linked to the corresponding tickets. The TCC could help in identifying these unissued tickets. We conducted a case study to examine requirements documentation and maintenance in an industrial agile software development project. Our study revealed that in 16% of all commits, software engineers committed source code to the VCS when no ticket corresponding to the changes was issued in the ITS. The unissued tickets accounted for 20% of all tickets. The software users and requirements engineers evaluated the contents of the unissued tickets, and we found that 42% of these tickets were required for software users in future software operation and 23% were required for future software modification.

We plan to conduct a similar case study examining requirements document maintenance in a large-scale software development project. We will also design and develop a tool to predict the occurrence of unissued tickets by monitoring the activities of the ITS and VCS. The goal of that tool is to prevent undocumented knowledge due to human error.

IX. Acknowledgments

The authors are grateful to Messrs. Takashi Hoshino, Keiichiro Horikawa, and Masayuki Inoue of NTT, Masatoshi Hiraoka and Daisuke Hamuro of NTT DATA, Noriaki Izumi and Motoi Yamane at Piecemeal Technology for their assistance in the case study.

References

[1] Backlog, http://backlogtool.com/?lang=1

[2] K. Beck and A. Cynthia. *Extreme programming explained : embrace change*. Boston, MA: Addison-Wesley, 2005.

[3] A. Cockburn. Agile Software Development: The Cooperative Game, Addison-Wesley Professional, 2006, 2nd edition.

[4] M. Cohn, Agile Estimating and Planning, Prentice Hall, 2005.

[5] S. Dorairaj, J. Noble, and P. Malik, "Knowledge Management in Distributed Agile Software Development," 2012 Agile Conference, pp. 64-73.

[6] GitHub, https://github.com/

[7] M. Lanza, M. D'Ambros, A. Bacchelli, L. Hattori, and F. Rigotti, "Manhattan: Supporting Real-Time Visual Team Activity Awareness," 21st Intl Conf. on Program Comprehension, 2013, pp. 207-210.

[8] M. Levy and O. Hazzan, "Knowledge management in practice: The case of agile software development," CHASE '09 Proc. of the 2009 ICSE Workshop, pp. 60-65.

[9] Manifesto for Agile Software Development, http://agilemanifesto.org/

[10] R. C. Martin. *Agile software development : principles, patterns, and practices*. Upper Saddle River, N.J: Prentice Hall, 2003.

[11] N. B. Moe, T. E Faegri, and D. S. Cruzes, and J. E. Faugstad, "Enabling Knowledge Sharing in Agile Virtual Teams," 11th Intl. Conf on Global Software Engineering, 2016, pp. 29-33.

[12] NTT DATA, http://www.nttdata.com/global/en/

[13] NTT R&D, http://www.ntt.co.jp/RD/e/index.html

[14] S. R. PALMER and J. M. Felsing, A Practical Guide to Feature-Driven Development, Prentice Hall, 2002.

[15] Redmine, http://www.redmine.org/

[16] S. Saito, Y. Iimura, K. Takahashi, A. K. Massey and A.I. Anton. "Tracking Requirements Evolution by Using Issue Tickets: A Case Study of a Document Management and Approval System," 36th Intl Conf. of Software Engineering, 2014, pp. 245-254.

[17] S. Saito, Y. Iimura, H. Tashiro, A. K. Massey, and A I. Anton, "Visualizing the Effects of Requirements Evolution," 38th Intl. Conf. of Software Engineering, 2016, pp. 152-161.

[18] K. Schwaber and M. Beedle. Agile Software Development with Scrum. Prentice Hall, 2001.

[19] Subversion, https://subversion.apache.org/

[20] The Trac Project, http://trac.edgewall.org/

[21] A. K. Thurimella, M. Schubanz, A. Pleuss, and G. Botterweck, "Guidelines for Managing Requirements Rationales," IEEE Software, Vol. 34, No. 1, pp. 82-90, Jan.-Feb. 2017.

[22] K. Wnuk, B. Regnell, and L. Karlsson, "Feature transition charts for visualization of cross-project scope evolution in large-scale requirements engineering for product lines," 4th Intl. Workshop of Requirements Engineering Visualization, 2009, pp. 11-20.

[23] R. K. Yin, Case Study Research: Design and Methods, in Applied Social Research Methods Series, Vol. 5, 2003, 3rd edition