2011 19th IEEE International Conference on Program Comprehension

Collective Code Bookmarks for Program Comprehension

Anja Guzzi*, Lile Hattori§, Michele Lanza§, Martin Pinzger* and Arie van Deursen*
* *Delft University of Technology, The Netherlands*
§ *REVEAL @ Faculty of Informatics, University of Lugano, Switzerland*

***Abstract*—The program comprehension research community has been developing useful tools and techniques to support developers in the time-consuming activity of understanding software artifacts. However, the majority of the tools do not bring collective benefit to the team: After gaining the necessary understanding of an artifact (*e.g.,* using a technique based on visualization, feature localization, architecture reconstruction, *etc.*), developers seldom document what they have learned, thus not sharing their knowledge. We argue that code bookmarking can be effectively used to document a developer's findings, to retrieve this valuable knowledge later on, and to share the findings with other team members.**

We present a tool, called POLLICINO, for collective code bookmarking. To gather requirements for our bookmarking tool, we conducted an online survey and interviewed professional software engineers about their current usage and needs of code bookmarks. We describe our approach and the tool we implemented. To assess the tool's effectiveness, adequacy, and usability, we present an exploratory pre-experimental user study we have performed with 11 participants.

I. INTRODUCTION

Software engineers are often faced with the challenge of understanding a program written by someone else and a long time ago. Due to the lack of proper documentation, program comprehension may take over 60% of the software maintenance effort [5]. Program comprehension methods and techniques can have a significant impact on the overall efficiency of software developers. However, program comprehension is an individualistic and ephemeral activity. Developers create their mental models of the system [16] to perform a task, but seldom document them, *i.e.,* most of the knowledge gained during the comprehension activity resides only in the developer's mind and typically is forgotten once the maintenance task is completed. Several barriers prevent developers from documenting their findings, such as constant interruptions, outdated documents that are hard to update, and preference for face-to-face talks [9].

"Traditional" code bookmarking is a common practice among users of modern Integrated Development Environments (IDEs), such as Visual Studio, Eclipse and IDEA.[1] Examples of bookmarks include task markers (//TODO and //FIXME), tag annotations (@Task), and cross-reference hyperlinks [14]. They have been used to remind developers of unfinished work [14], which serves a short-term purpose; and to guide programmers to perform a task [4], [7], [11]. Most of these bookmarks are embedded in the code, which introduces a trade-off between the benefits of sharing and the drawbacks of having "noise" in the code.

We propose an approach for micro-documentation and knowledge sharing: *collective code bookmarks*. It encourages developers to bookmark artifacts while investigating the source code, and to document their findings with a description associated to a bookmark, which can be shared with other team members or maintained on the developer's workspace for private use. A collective code bookmark is a link from a specific location in a file to one or more descriptions, which may be comments, links to resources, documents, websites, and tags. Extra information can be associated with a bookmark, such as author, creation date, *etc.* Our goal is to support the benefits of collective code bookmarks, and keeping them as an external documentation.

We conducted a survey on the current usage of code bookmarks, as well as interviews with professional software engineers, to gather the requirements for collective code bookmarks and for a supporting tool to encourage developers to document their findings while performing a program comprehension task. Based on this survey, we present an approach for collective code bookmarks with the goal of helping developers to retain and share the knowledge acquired during program comprehension activities.

The contributions of this paper are:

- *The elicitation of requirements for a non-intrusive bookmarking tool that facilitates knowledge sharing.* This elicitation is the outcome of an online survey on current usage of code bookmarks, and several interviews conducted with professional software engineers.
- *An approach to code bookmarking.* We devised our approach as a publicly available tool named POLLICINO. We present our approach and describe the tool.
- *An exploratory pre-experimental user study with 11 participants* to investigate the potential of collective code bookmarks and to evaluate the tool's effectiveness, adequacy, and usability; with promising preliminary results.

Structure of the paper. Section II presents the related work on code bookmarks. We motivate our work in Section III, where we report on requirements for a collective bookmarking tool, elicited from a survey and the interviews with practitioners. In Section IV we describe our approach

[1] See microsoft.com/visualstudio, eclipse.org/, jetbrains.com/idea/

1063-6897/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPC.2011.19

and the tool implementation. Section V presents the design of our experiment, while Section VI reports on its results. Section VII discusses the threats to validity. Finally, we summarize our conclusions in Section VIII.

II. Related Work

Code bookmarking can be classified as one form of support for user-defined annotations. Other forms are task markers, tag annotations, and cross-reference hyperlinks [14]. These annotations are relatively common practice among users of modern IDEs, however their design and purpose vary greatly.

Eclipse's user-defined annotations are classified into three categories: (1) tasks – //TODOs, //FIXME, //XXX; (2) problems – reporting invalid states of the system; and (3) bookmarks – for marking locations and having quick access to them. Tasks reside in the source code and have the purpose of reminding (recalling information regarding a specific piece of code) developers, while problems and bookmarks are external links to a line in a source file.

Brühlmann *et al.* proposed a generic annotation tool, called Metanool to capture and retain human knowledge during reverse engineering processes [2]. Metanool supports the association of any type of annotation (*e.g.,* comment, document, UML diagram) to a source code entity. However, it was targeted at facilitating reverse engineering activities instead of supporting active development.

TagSEA combines the notion of marking locations in spatial navigation with social tagging to support reminding and refinding (revisiting a specific part of the code) [14]. The TagSEA annotation has the form of a customized Java annotation, residing in the source code, and thus being shareable across the team. These annotations were designed to be easy for development teams to search, filter, and manage. They also proved to be useful for creating tours for technical presentations that involve interacting with the IDE [4]. TagSEA was evaluated in several (longitudinal) case studies: the findings focus on the sort of tags used and the extent to which tags could be reused.

JTourBus is a similar approach that creates tour guides to help programmers to perform a task or to assist them to get familiar with a sequence of code dependencies [11].

Contrary to TagSEA, our collective bookmarks are designed to reside outside the source code, bringing two benefits: flexible privacy setting – the author of the bookmark can decide whether to share it and with whom; and cleaner code. Furthermore, our evaluation is different, emphasizing code locations over tags, and being carried out with a larger group of participants all conducting the same tasks. Different from JTourBus, our approach does not provide means to create guides, but provides a lightweight approach for developers to micro-document their findings when performing program comprehension activities.

III. Motivation

Previous studies have reported a low use of code bookmarks that do not reside in the source code. In their report on usage data collected from developers using Eclipse, Murphy *et al.* state that only 5 out of 41 developers used the bookmark view [10]. In another survey, Storey *et al.* report that 84% of the respondents never or rarely use bookmarks [13]. Some hypotheses are raised to explain the low adoption, such as *poor visibility* in the IDE, or *poor synchronization* with the code; however no further investigation was performed to understand the reasons for the low adoption.

We conducted an online survey to investigate *why* bookmarks are rarely or never used in modern IDEs. We collected a total of 209 responses from which 71% of the participants were practitioners and 29% academics. The vast majority consisted of experienced developers (60% had more than 6 years of experience, 30% had 3-5 years of experience, the rest had less than 3 years of experience).

From the respondents, 88% report that they never or rarely use code bookmarks (confirming previous results). We furthermore asked them why this is the case. Among those 88%, who never or rarely use bookmarks, 50% answered they did not know that bookmarks existed in their IDE, 25% stated that they do not find them useful, while 9% think creating a bookmark is cumbersome (Figure 1).

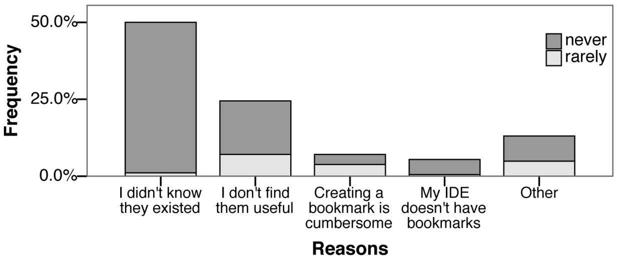


Figure 1: Reasons for not using bookmarks in the IDEs

We also collected qualitative information about how people are using bookmarks. A number of respondents reported that bookmarks are useful for keeping track of points of interest in the code while understanding it (*"I use them while I'm skimming code and trying to get a clue about how to fix something. I therefore bookmark interesting pieces of code to be further inspected"*) and to quickly access them later (*"as a way to return back to the same bit of code a few days later"*). Respondents also reported to use them as a guide while working on a task (*"For marking something of interest, so I don't lose it. shift-F11 in IDEA lists all bookmarks - I tend to create a series of bookmarks for a particular task, so I can find the main bits of the problem."*).

To further investigate the potential of our approach, we conducted interviews with four practitioners with different roles: (R1) a software developer with 6 years of experience who was not aware of code bookmarks; (R2) a software developer with 8 years of experience who was aware of code bookmarks; (R3) an open-source developer 10 years of experience; and (R4) a software architect and team leader with 25 years of experience.

R1 reported that his team has the culture of placing regular comments and task markers in the code to remind, refind, and annotate decisions. This is a problem when the code goes to quality assessment: *"In the system we work we see thousands of TODOs, and we never do anything about it. (...) Every six months the manager freaks out because of the quality control, and we have to clean up the entire code."*

R2 actively uses code bookmarks, mainly to understand a legacy system that his team maintains. He reported that system size and lack of documentation are the main reasons for using bookmarks: *"The code is huge and I really need them as breadcrumbs, in particular when I'm digging into code. (...) I'm changing code that was written by 5-6 people in 10 years with no architecture, no design, (...) and I must have a way to track all the jumps that I'm doing."* One of his main complaints was the lack of share-ability of bookmarks: *"(...) and I actually tell him (teammate) where to add his bookmarks (...) but since up to now there's no way of sharing them, most information remains just for me."* Since R2 is actively using code bookmarks, we asked him what features he would like to have, and his answer was: *"I want to share my bookmark's description, and I'd like to add resources to it. (...) Keep it simple."* Other suggestions were to offer grouping options, and sequencing by the user's choice.

With R3 and R4 we investigated when collective bookmarks would be useful. R3 thought they could be very useful to assist the developers of his team, who are spread across different locations. R4 identified the following situations: when a developer is passing the responsibility of a part of the code to another person; to maintain traceability between UML diagrams and the code; in the beginning of development, to annotate the most important methods of the API.

Summary. Based on the survey and the interviews, we have identified the following *requirements* for collective code bookmarks and for a tool that supports them.

Be share-able. Current bookmarks are either private or can be shared through the source code, imposing a restriction on how a developer can share it. The feedback collected suggests that developers want to be able to share their bookmarks, but doing it so through the code clutters the source code.

Have a description. Code bookmarks should be used as a means for micro-documenting a finding. Having a textual description of the finding is essential for reminding it later.

Support grouping and sequencing. Developers would like to be able to associate auxiliary information to bookmarks to help organizing them.

Be platform independent. There are numerous languages and IDEs, and having a bookmark model for each combination restricts the social benefit of bookmarks. There should be a standardized model that can be used in any language/IDE.

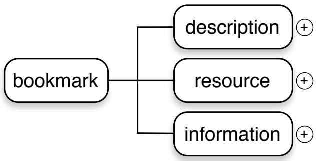


Figure 2: Meta-model for collective code bookmarks

Be simple to use. The action of bookmarking a location in the code should be non-intrusive, intuitive and almost effortless. Otherwise the gain from the information registered is hindered by the effort to register it.

IV. The Pollicino Approach

A. Collective Code Bookmarks

Two requirements for collective code bookmarks are to be shareable and platform independent. Thus, we redefine code bookmarks by proposing a meta-model, in the form of a standardized and extensible XML schema. Using an XML schema to encode the above information allows code bookmarks to be shared as XML files, making them portable to any IDE and attachable to any source file.

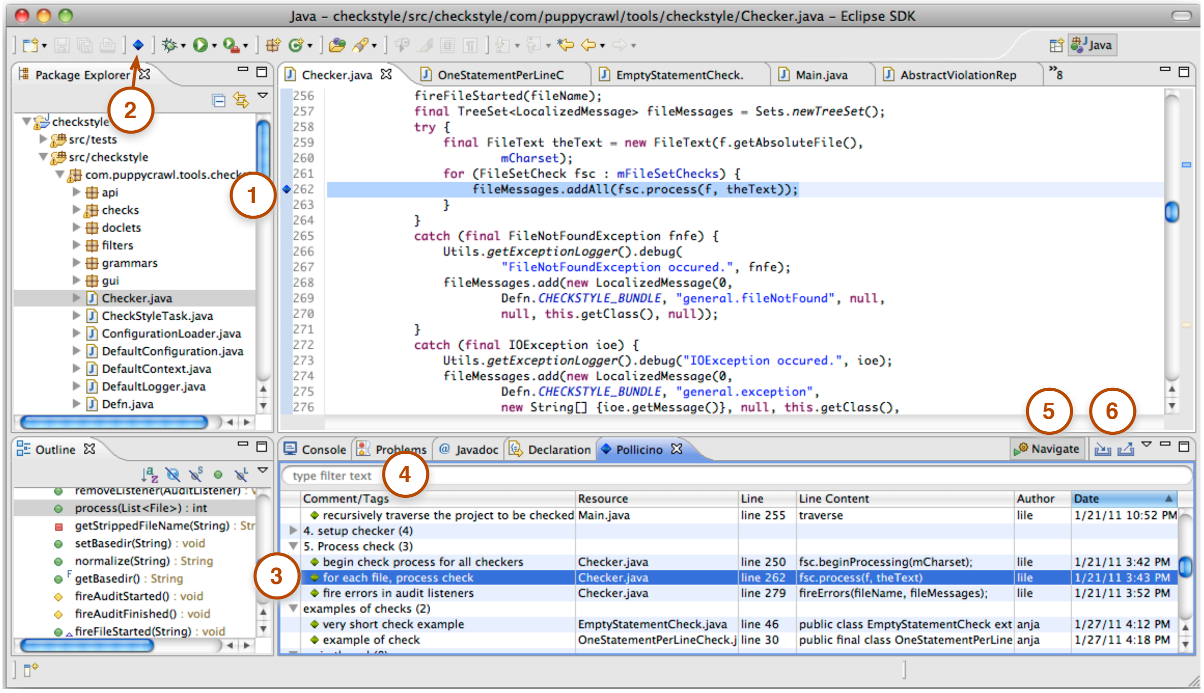
Figure 2 illustrates our meta-model, composed of three parts. The *description* can be a text message, a link to internal/external resources, or tags. Any type of description can be added (ranking attributes, pictures, movies, etc.). The *resource* contains information about the location of the bookmark in a file: project – the name of the project; location – the full path of the file within a project; line number – the line number where the bookmarks should be placed. The three attributes are mandatory to properly locate the bookmark in a file. Finally, *information* contains an extensible list of optional attributes (such as author, creation time, group, or a file revision number).

B. Pollicino

To assess the feasibility and usefulness of collective code bookmarks, we have implemented POLLICINO[2] (see Figure 3), an Eclipse plug-in that allows developers to place bookmarks as a way to micro-document their findings, and share them with others. In the following, we present the main features of POLLICINO.

Bookmarking the code. To bookmark a code snippet, the user can mark it and right-click on the editor or on the left ruler, and choose "Add Stone" in the popup menu. The user then enters the minimum information (*e.g.,* a description) after which POLLICINO inserts a blue marker into the left ruler at the corresponding line (Point 1). Bookmarks can be added in any type of file (*e.g.,* source code, text file, XML file, build file, *etc.*). Some participants of the survey reported

[2] Pollicino (Little Thumbling in Italian) is inspired by the fairy tale Hop-o'-My-Thumb, in which a boy uses pebbles to find his way back home.



```
            fireFileStarted(fileName);
            final TreeSet<LocalizedMessage> fileMessages = Sets.newTreeSet();
            try {
                final FileText theText = new FileText(f.getAbsoluteFile(),
                        mCharset);
                for (FileSetCheck fsc : mFileSetChecks) {
                    fileMessages.addAll(fsc.process(f, theText));
                }
            }
            catch (final FileNotFoundException fnfe) {
                Utils.getExceptionLogger().debug(
                        "FileNotFoundException occured.", fnfe);
                fileMessages.add(new LocalizedMessage(0,
                        Defn.CHECKSTYLE_BUNDLE, "general.fileNotFound", null,
                        null, this.getClass(), null));
            }
            catch (final IOException ioe) {
                Utils.getExceptionLogger().debug("IOException occured.", ioe);
                fileMessages.add(new LocalizedMessage(0,
                        Defn.CHECKSTYLE_BUNDLE, "general.exception",
                        new String[] {ioe.getMessage()}, null, this.getClass(),
```

Comment/Tags	Resource	Line	Line Content	Author	Date
◆ recursively traverse the project to be checked	Main.java	line 255	traverse	lile	1/21/11 10:52 PM
▶ 4. setup checker (4)					
▼ 5. Process check (3)					
◆ begin check process for all checkers	Checker.java	line 250	fsc.beginProcessing(mCharset);	lile	1/21/11 3:42 PM
◆ for each file, process check	Checker.java	line 262	fsc.process(f, theText)	lile	1/21/11 3:43 PM
◆ fire errors in audit listeners	Checker.java	line 279	fireErrors(fileName, fileMessages);	lile	1/21/11 3:52 PM
▼ examples of checks (2)					
◆ very short check example	EmptyStatementCheck.java	line 46	public class EmptyStatementCheck ext	anja	1/27/11 4:12 PM
◆ example of check	OneStatementPerLineCheck.j	line 30	public final class OneStatementPerLine	anja	1/27/11 4:18 PM

Figure 3: Eclipse with the Pollicino plug-in

that creating bookmarks is cumbersome. To alleviate this, the user can create bookmarks with a simple keyboard shortcut, easing the mechanical process of creating bookmarks.

Showing bookmark information. The POLLICINO view can be displayed with one click (Point 2), showing the bookmarks that are currently in the user's workspace and the information associated with them (Point 3). The view shows the user's comment, the resource and the line where the bookmark is located, the line content, the author and the date.

Grouping and sorting bookmarks. There are different ways to organize bookmarks on the view. They can be grouped by "active/archived", or the user can create customized groups. The user can also sort bookmarks according to any column of the view. The information in all the columns can be searched for specific information using the search bar (Point 4).

Navigating through bookmarks. Double-clicking on a bookmark entry in the view takes the user to its location in the editor. The view also provides a *Navigation* mode (Point 5), which shows the location of a bookmark with a single click (or navigating with up and down arrows).

Sharing bookmarks. It is possible to import/export all or a selection of the bookmarks from/to an XML file that follows the schema presented in Section IV-A (Point 6).

Customizing bookmarking. When creating a bookmark, by default the user has to enter a comment and her name. The user can transform the creation of bookmarks as a one-step process by customizing the information she needs to enter on the preferences page (*e.g.,* save her name and add a blank comment).

POLLICINO is available for download at http://www.st.ewi.tudelft.nl/~guzzi/pollicino/.

V. PRE-EXPERIMENTAL STUDY DESIGN

We conducted a user study with the goal of investigating the potential of our approach. To achieve our goal, we assess the tool's *effectiveness, adequacy and usability* using an exploratory pretest-posttest pre-experimental design [3]. Pre-experimental indicates that the experiment does not meet the scientific standards of experimental design [1], yet it allows to report on facts of real user-behavior, even those observed in under-controlled, limited-sample experiences.

The study is composed of several experimental runs. Each run consists of the initial pretest questionnaire, a demo/tutorial of POLLICINO followed by three assignment tasks (to perform within Eclipse with our plug-in installed), a final posttest questionnaire and a concluding debriefing talk. The complete handout used in the experiment, including the tasks, the pretest and posttest questionnaires, can be found in our on-line appendix [8].

A. Research Questions

The main goal of the experiment is to investigate whether POLLICINO can help software developers during program comprehension activities (*effectiveness*). Our main research question is:

RQ1 Can collective code bookmarks help developers to perform program comprehension activities?

We identify the following three sub-research question:

RQ1.1 Can Pollicino be used to (micro-)document a developer's own findings?
RQ1.2 Can Micro-documentation via bookmarks be useful to team members to get starting points (*i.e.,* entry points for program comprehension)?
RQ1.3 Can Pollicino be useful during development tasks?

A secondary goal of the experiment is to gather initial feedback on our tool. In particular we are interested to explore whether the tool's outcome matches the user's expectations (*adequacy*) and whether the tool is easy to use (*usability*). We thus identify the two additional research questions: (**RQ2**) Does Pollicino match the expectation for a code bookmarking tool? and (**RQ3**) Is Pollicino sufficiently usable by developers? Finally, since Pollicino is in an initial stage of development, we also want to gather feedback on how to improve it.

B. Pretest-Posttest Design

In a pretest-posttest study, participants are subject to a first test (questionnaire), before performing the experiment, and to a second test, after performing the experiment. For most questions in our questionnaires, we make use of closed-ended matrix questions: participants are asked to rate a number of statements on a five-point Likert scale.[3]

Pretest Design. The pretest questionnaire is split into 5 parts. The first two parts are dedicated to understand the personal background (part 1) and the software development experience (part 2) of our participants.

In part 3 we ask participants to rate a number of statements regarding their habits when understanding code. A participant's habits might influence her expectations with respect to a tool that eases program comprehension. We thus investigate common practices, *e.g.,* reading the comments inside the code, reading the available documentation, and making changes to the code and run it to see what happens. Similarly, in part 4, we try to assess the participant's familiarity and attitude toward code bookmarking. Participants are asked to rate dedicated questions, according to whether or not they were aware of the bookmarking feature of Eclipse. Additionally, an open ended question asks them to provide alternatives that could replace bookmarks in their function of *code location markers*.

In part 5, participants are given an abstract description of Pollicino (see [8] for the full text), and asked to rate a number of statements about their expectations of a tool with such functionalities (see Table I). These statements are then asked again in the posttest, after introducing the independent variable (*i.e.,* the use of the tool) and serve to assess adequacy.

Table I: Statements used in the pretest to measure adequacy

a)	Such a tool would prevent me from getting lost in the code
b)	I don't see the added value of such tool
c)	Bookmarking would help me when I'm trying to understand a functionality
d)	As soon as I understand the code, bookmarks become useless to me
e)	Such a tool would help me manage points of interest in the code
f)	The tool will not be able to help me in real problems
g)	My bookmarks will help others understand what I did

Posttest Design. The posttest questionnaire is given to the participants after they performed the assigned tasks to, *e.g.,* verify whether Pollicino is seen as a good implementation of a bookmarking tool.

We split the posttest into 5 parts. In part 1, we ask the participants to assess a few statements about their general experience in performing the experiment, *e.g.,* whether they found the three tasks doable and whether they felt time pressure.

In part 2 of the posttest we ask participants about their experience with Pollicino while performing the tasks. In particular, we collect information on the tasks and purposes the tool has been used for. We use the results from this part to address **RQ1**.

Part 3 of the posttest is dedicated to assess the participants' perception of Pollicino. Participants are asked to rate similar statements as in question 5 of the pretest. The difference is that, at this time, we ask directly about the tool and not about code bookmarking in general. By comparing answers to this question and to its counterpart in the pretest, we can verify how Pollicino meets the participants' expectation and address **RQ2**.

In part 4 participants are asked whether they used particular features of the tool (*e.g.,* key binding to add a marker) and, if yes, to indicate how useful they were on a scale from 1 = "very useless" to 5 = "very useful". The questions in part 5 are used to measure the usability of the tool. These measures are used to address **RQ3**. In addition, these statements allow us to verify whether usability issues might have influenced the overall experience with the tool during the assignment.

After the posttest questionnaire, we held a debriefing talk to collect additional information, both on the tool and the experiment. During this individual talk with participants, we could better understand about their experience with the tool and ask them about their opinion on possible improvements.

C. Checkstyle

The system we chose as object of our experiment is *Checkstyle*.[4] We used version 5.3, which consists of 341 classes distributed across 22 packages, for a total of 46

[3] 1 = "strongly disagree" , 2 = "disagree" , 3 = "neither disagree nor agree", 4 = "agree", 5 = "strongly agree".

[4] See http://checkstyle.sourceforge.net/

KLOCs.[5] Our choice was motivated by the following factors: Checkstyle's size allows for performing an experimental session, yet being representative of real life programs. It is written in Java, with which many potential participants are sufficiently familiar. It has been used in previous experiments [15], [17], [6], from which we could reuse one of the tasks.

D. Tasks

We designed the tasks for our experiment according to the sub-research question related to **RQ1**, from which we derive the following three different scenarios: (1) investigate and understand a part of a system, (2) (micro-)document a system's functionality, and (3) add functionality to the system. The scenarios require a program comprehension process and are inspired by Pacione's taxonomy [12].

Task T1 is used to address **RQ1.2**. It requires the participant to gain general knowledge about the execution stages of Checkstyle, and is reused from Cornelissen's controlled experiment [6]. To simulate a collaborative environment, a number of code bookmarks were already placed in the project. T2 is used to address **RQ1.1**. It encourages the participant to understand and simultaneously document (by adding bookmarks) the class hierarchy related to the functionality that checks the adherence to each code convention. The last task T3 is focused on implementing a new check, to which the knowledge obtained and the bookmarks added in the previous two tasks is potentially useful. We use this task to address **RQ1.3**.

The three tasks were tailored to be feasible in the allotted time (20 minutes for each task) considering the minimal experience level required from the participants. For more information on each task we refer the reader to [8].

E. Pilot Studies

We conducted four pilot studies to test the experiment's feasibility and duration. With the first two runs we found out that two of the tasks needed to be more focused and have more guidance to be doable in the allotted time. We furthermore identified some defects in POLLICINO, which were fixed before the actual experiment. After the third test run, we adjusted a few statements from the questionnaires. The fourth pilot then ran without any particular problem.

VI. RESULTS

We report on the results obtained from our experiment. We first describe our participants and their attitude toward program comprehension and code bookmarks (as measured from the pretest). Then, we report on their performance during the assignment (as measured from the posttest and from analyzing the code bookmarks placed during the experiment). Finally, we present the results from the posttest, along with the feedback from the participants, to answer our research questions.

A. Participant Characteristics

12 volunteers participated in our experiment. 2 participants were from the University of Antwerp, 2 from the University of Lugano and 7 from the Delft University of Technology. 4 participants were PhD students, 7 participants were MSc students (3 of which are working as part-time developer). All participants were male, aged between 24 and 30. One participant reported that the tasks were not feasible. Since task feasibility was a requirement for our experiment, we excluded his results from our analysis, giving a total of 11 subjects in our study.

Figure 4 reports the participants' experience regarding software development on a 5-point scale.[6] The box plot depicts the following data: minimum value, lower quartile, median, upper quartile, and the maximum value. As can be observed, participants reported to have good knowledge of Java, IDEs, and Eclipse, while the experience in working on industry-sized systems is lower. All participants reported to have no or low experience with Checkstyle, indicating that their answers were were not influenced by previous knowledge about Checkstyle.

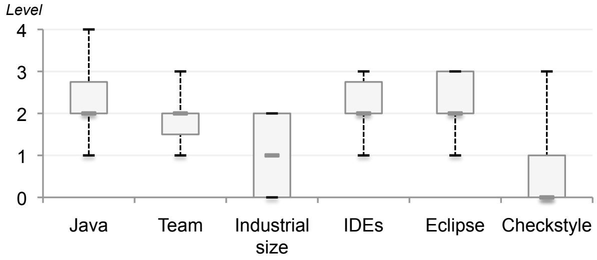


Figure 4: Participants' experience

B. Comprehension Attitude

In the pretest, we ask the participants to rate a number of statements aimed at measuring their attitude toward both program comprehension and bookmarking. We next go into detail of these two aspects.

Attitude toward program comprehension. The most popular practices (indicated by 10 out of 11 participants) when working on a task on unfamiliar code of a system are to investigating the code before starting with the task, and to reading the code comments (if available). Most of the participants recognized that they often navigate through several classes and find themselves lost amongst many open tabs. 7 participants stated that they read the available documentation, and make changes and/or add print statements to the code and then run it to see what happens. Some less popular practices indicated are: to start looking at available

[5] Measured using http://eclipse-metrics.sourceforge.net/

[6] 0 = "does not know the subject", 1 = "familiar with the subject, but still have some difficulties with it", 2 = "comfortable in the subject and currently using it daily", 3 = "highly proficient in this subject", 4 = "serves as reference for colleagues, and feels confident in helping them".

Table II: Examples of bookmarks placed during the experiment

Comment	File	Location
Use this method to specify which tokens to respond to	Check.java	getDefaultTokens()
Implement which token the check is interested in	Check.java	getDefaultTokens()
If you're interested, this is where your Check is actually called	TreeWalker.java	notifyVisit(DetailAST aAST)
Data structure, it's parsing a tree	DetailAST.java	class declaration
This class must be extended for Format checking purposes	AbstractFormatCheck.java	class declaration
Use this class for logging check output	AbstractViolationReporter.java	class declaration
Checks respond on tokens. These are the types you can respond to. Compare this to the Listener system: you subscribe to all tokens you want to respond on	TokenTypes.java	class declaration

test suites, to use specific tool or markers (*e.g.,* TODO), or to look at UML diagrams.

Overall, we can say that our participants' effort in program comprehension relies mostly on investigating the code, reading comments and documentation (when applicable), and making changes (or insert print statements) before running the code.

Attitude toward (code) bookmarking. One participant who knew about code bookmarking within Eclipse indicated that the feature is relatively easy to use. According to him, creating a bookmark is not particularly cumbersome and code bookmarks are useful. Another participant did not think that bookmarks are easy to use. He indicated that creating code bookmarks is cumbersome and he is not sure whether they are useful.

7 out of the 9 participants who did not know about this feature, indicated that they agree with the statement *"Code bookmarks seem to be useful"*. The remaining 2 indicated they "neither disagree nor agree" with the statement.

We asked participants to shortly report on which tools or methods they use to function as "code location markers" during their program comprehension routines. One participant reported that he makes use of visually outstanding comments (such as *"//*************//"*), so that the location is easily spotted when looking at the code. Alternatively, he introduces lots of consecutive new lines, to create a easily noticeable blank space. Another participant explicitly introduces compilation errors and also uses a custom tool. One participant wrote that he uses Mylyn to link code to tasks.

Overall, most participants were not aware of code bookmarking features, yet acknowledged the usefulness of code location marking facilities.

C. Task Performance

During the assignment, the participants were invited (but not required) to use the bookmarks already placed in the project, and to add their own bookmarks. Table II illustrates some typical bookmarks they added while performing the tasks. Observe that several bookmarks were placed at the declaration of a class with a short description of its purpose. Other locations where bookmarks were commonly placed are at the method declaration, with a short description of its functionality; or at some statement inside a method, where there is a method call of interest.

Table III: Number of participants who placed a certain number of bookmarks for T2

# Participants	2	2	1	1	3	2
# Bookmarks	8	6	5	4	3	0

For T1, we provided a set of bookmarks that was split into 5 groups, which corresponded to the main stages of a check in Checkstyle. Most participants seem to have benefitted from the grouping, since 10 out of 11 correctly identified the stages, which varied from 4 to 6 in their answers.

T2 required understanding the Check hierarchy, placing bookmarks during this process. The number of bookmarks placed by the participants is shown in Table III. A common practice was to look for the abstract implementation of a `Check` that could be extended, or an example of its implementation, and mark its location.

For T3, we asked the participants to implement a check to count the number of methods in a class. 2 participants had correct implementation, 3 copied the content of a similar class (making the implementation also correct), 2 had incomplete implementation on the right track, and 4 had incorrect implementation. We did not identify a correlation between the number and quality of bookmarks added in T2 and the quality of the code implemented for T3.

Overall, participants reported that the tasks were feasible, interesting, and realistic, that the warm up task (a short "hands-on", which preceded the tasks) was useful, and that the experiment was fun to do. 4 participants indicated that they would have needed more guidance to complete the tasks. 8 participants felt time pressure.

D. Experience with collective code bookmarks

In this section, we discuss and answer **RQ1**, which is associated with the effectiveness of POLLICINO in helping developers to perform program comprehension activities. We first address the three sub-research questions related to it.

RQ1.1: *Can* POLLICINO *be used to (micro-)document a developer's own findings?* Participants reported that POLLIcino was useful during T2 (6 participants reported to have used the bookmarks already contained in the code), where the scenario was to (micro-)document some of the system's functionality.

In addition, participant feedback on Pollicino's bookmarks was generally positive. They found the navigation of bookmarks useful, as they could *"tag important stuff and then (..) quickly navigate to it later"* (P6). P8 hypothesizes bookmarks could be used as a mind map for a developer to document his findings. P2 sees *"a very natural relationship between* Pollicino*, code, and diagrams"*, and suggested the possibility to link Pollicino's bookmarks with design documents (*e.g.,* UML diagrams). P7, who has work experience, said his practice is to annotate code via comments and then search for them. After trying our tool, he thinks *"a bookmark would be very handy for that"*.

We conclude that Pollicino can be used to (micro)-document a developer's own findings.

RQ1.2: *Can micro-documentation via bookmarks be useful to team members to get starting points?* Participants reported that Pollicino was useful during the first task T1, where the scenario was to investigate and understand a part of a system. 10 had correct answers to the task.

Additionally, during the debriefing talks, participants recognized the value of using Pollicino within a team, *e.g.,* by having step-by-step instruction to help newcomers steer their way in a project (P9). They also stressed the importance of sharing and synchronization of bookmarks, recognized the need of supporting synchronization of bookmarks, proposing to have bookmarks automatically integrated with a version control system. Not all participants are convinced that existing bookmarks helped them to understand the system. They emphasized that a bookmarks' description should be as meaningful to other people as to its author.

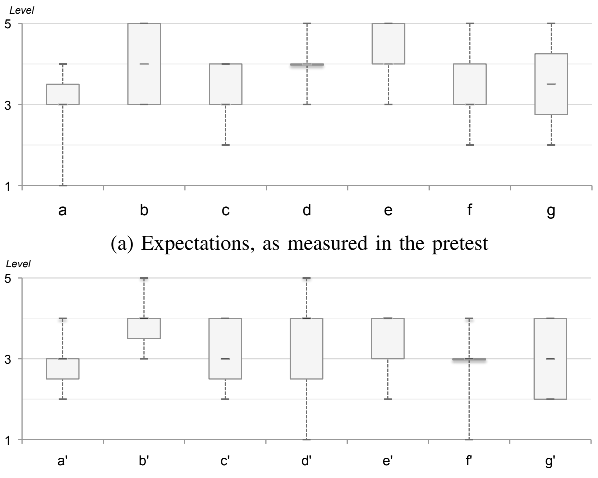
We conclude that Pollicino has a potential to be useful for team members to get starting points.

RQ1.3: *Can* Pollicino *be useful during development tasks?* Only 2 participants reported Pollicino as useful during T3, which asked to implement a functionality in the system. Also, there seem to be no correlation between the number of bookmarks placed and the quality of the code implemented for T3.

A few participants mentioned during the debriefing talks that Pollicino bookmarks could be used during development tasks, *e.g.,* as a replacement for "temporary" TODOs (P3). Participants were not convinced about the potential of Pollicino during development.

We conclude that regarding active development, Pollicino is not as useful as for documenting or understanding code.

Summary. Overall, participants found our tool useful when their tasks were precisely to understand or document the code (9 and 8 matches, respectively), while they did not find it useful during implementation (only 2 found it useful). This observation reflects on the use of bookmarks, which were mostly added during the micro-documenting task, as indicated by 10 participants. Therefore, we can answer **RQ1** positively by stating that collective code bookmarks can help a developer during the documentation of her own findings, and this information can be useful to her team members.



(a) Expectations, as measured in the pretest

(b) Perception, as measured in the posttest

Figure 5: Expectations and perception of a tool like Pollicino

E. Expectations vs Perception of Pollicino

To answer **RQ2**, associated with the *adequacy* of our tool, we analyze of expectations vs. perception of Pollicino. In the pretest, the participants rated seven statements about a hypothetical tool with Pollicino's functionality (see Table I). In the posttest they rated the same statements, but this time about Pollicino itself. By comparing the answers to this question and to its counterpart in the pretest, we can verify how the perception of our tool meets the participants' expectation.

Our sample is composed of (the ratings of) 11 participants. The box plots in Figure 5 depict the following data from our sample: minimum value, lower quartile, median, upper quartile, and the maximum value. To ease the visualization and analysis of the data, we mirrored the ratings of statements b, d and f (*cf.* Table I) that were formulated with a negation (*e.g.,* "4 = agree" becomes "2 = disagree"). We did the same for the ratings of the corresponding statements in the posttest.

Figure 5a summarizes the ratings obtained for the expectations of a bookmarking tool. Thus, in advance many participants are not sure about whether the tool can prevent them from getting lost in the code (a). Most participants are positive about the added value of the tool (b), its value to others (d), and about helping someone to manage points of interest in the code (e). Participants were less, but still positive about the tool helping them while trying to understand a functionality (c), especially in real problems (f), and also helping others to understand one's findings (g). In general, the expectations of a bookmarking tool with collective benefits were very positive.

Figure 5b summarizes the rating levels we obtained for the perception of POLLICINO. We can see that after using POLLICINO, the perception is positive for most participants and in most aspects. This is reflected by the fact that most participants see the added value of POLLICINO (b'), think that the tool did not become useless when they understood the code (d'), and believe that the tool helped them to manage points of interests in the code (e'). Participants are still not sure whether POLLICINO was able to help them with real problems (f'), and prevented them from getting lost in the code (a'). They have different opinions on whether their bookmarks will help others understand what they did (g').

Overall, the perception of the participants remained positive, although the comparison of the box plots in Figure 5 suggests that POLLICINO does not match the high expectations of the participants. To test whether this difference is significant, we performed a Wilcoxon signed-rank test with the ratings from the paired statements. Wilcoxon is a non-parametric test to assess the null hypothesis that the medians of two samples do not differ. The results show that for each pair the Wilcoxon test is non-significant (p-value>0.05), hence from this data we can not conclude a difference between the expectations and perception of POLLICINO.

Therefore, we answer **RQ2** by arguing that, even though the results of the comparison suggest that POLLICINO did not match the expectations of a collective code bookmarking tool, the Wilcoxon test did not show a statistical significance on the difference between expectations and perception. During the debriefing talks, some of the participants argued that they were expecting the tool to teach them how Checkstyle works (*e.g.,"they are a starting point, but they don't teach you the system"* (P8)). Instead, the goal of POLLICINO is to guide its users to points of interest while understanding a concept of the system. Hence, there was some mismatch between what the tool could offer and what the participants were expecting from it.

F. Tool Feedback

Pollicino Usability. Of the 11 participants, 9 found POLLICINO easy to use, while the other 2 rated the statement *"I found Pollicino's Eclipse extension easy to use"* with "neither disagree nor agree". Participants are generally positive that they would be able to get used to POLLICINO during their everyday work activities. Participants reported they did not get error messages while using the tool and that defects they possibly experienced did not severely hinder its usefulness. Furthermore, no participant reported that the information from the tool distracted him from the tasks.

The feedback on the tool's usability was positive: participants liked the simple, but complete and easy to use, interface of POLLICINO. P2 said that *"everything I needed was there"*, while P5 observe that *"It's a basic Eclipse feature, so it's easy to use and understandable"*.

Based on this, we can answer **RQ3**, regarding POLLICINO's *usability*, positively.

Pollicino features. Since our tool is in an initial stage of development, we also want to gather feedback on how to improve it. We focus here on the tool's features. Their perceived usefulness was measured in part 4 of the posttest, while additional feedback and suggestions were collected during the debriefing talks.

All eleven participants indicated that the possibility of grouping bookmarks and the POLLICINO view are "useful" and "very useful". The same rating level was given to both means to add a bookmark in the IDE (*i.e.,* the popup menu and the keybinding), for which most participants only used the popup menu, two participants used only the keybinding and one reported to have used both (finding both of them useful). Moreover, participants find the navigation mode in the view, used by 9 of them, and the sharing (import/export) of bookmarks also useful. 8 participants used the option to archive and activate bookmarks, and they have different opinions about whether it is useful or not, with a tendency to find the feature not so useful.

In the debriefing talks, participants suggested ways to improve POLLICINO. In summary: have a custom order for the bookmarks within groups; be able to hide archived bookmarks; have the possibility to add a new empty group; have the possibility to create a hierarchy for bookmarks (per user, per package, per class, *etc.*); be able to add multiple bookmarks to one location in the code; have different type of bookmarks (*e.g.,* text comment, example, todo) to further categorize bookmarks, and maybe have different colors for each type.

VII. THREATS TO VALIDITY

A. Internal Validity

Participants. To ensure the minimal knowledge required to perform the assignment, we asked them to rate their expertise on a number of topics related to the experiment. In addition, feasibility of the tasks was a requirement, and the one participant who felt the assignment was unfeasible has been excluded from the analysis.

Questionnaires and Tasks. The 5-point scale questions may have influenced the participants to follow a pattern on assigning points to the statements. We interleaved affirmative and negatory statements to mitigate this effect. When we provided a description of a collective code bookmarking tool, we may have influenced the participants to think our tool would teach them about the object system, which may have had a negative influence on the comparison between expectation and perception of our tool. Each task was associated to one program comprehension activity for which POLLICINO may be helpful. For one of the tasks, the authors added bookmarks that might have made it too easy to answer.

Experimental runs. There were several runs, and differences between them, such as different training of POLLICINO, may have influenced the results. To alleviate this effect, we ran 4 pilots to fine tune the experiment, and followed a defined script when giving the tutorial of the tool.

B. External Validity

Participants. The fact that the participants were from academia may have limited our ability to generalize the results to the industrial environment. To alleviate this effect, we made sure participants had a minimum knowledge of the related topics, and felt the tasks were feasible. Also, a number of participants have experience as practitioner.

Tasks. Our choice of tasks may not reflect real questions related to program comprehension. To mitigate this threat, our tasks were inspired by Pacione's taxonomy [12], and Task T1 was reused from a previous controlled experiment on program comprehension [6].

Object System. Even though Checkstyle is a largely used open-source system, it may not be representative of complex commercial systems. Thus, the use of a different object system may have yielded different or more reliable results.

VIII. CONCLUSION

We have reported on the results of a survey with 209 respondents and on interviews with 4 practitioners. After eliciting the requirements for a collective code bookmark approach, we presented POLLICINO. We reported on a pretest-posttest pre-experimental user study to assess the effectiveness, adequacy, and usability of POLLICINO. 11 subjects participated in our user study, which consisted of: performing three program comprehension tasks (using the Eclipse IDE with POLLICINO installed), answering two questionnaires (one before and one after the tasks), and having an individual semi-structured debriefing interview.

The results illustrate that POLLICINO can be effectively used to (micro-)document a developer's findings, and that those can be used by others in her team. However, the tool was not effectively used for program comprehension during active development. This could be partially due to the need of adjusting one's work habits, when a new tool is introduced. We also assessed and concluded that POLLICINO is usable. Directions for future work are to improve our tool (*e.g.,* to better support synchronization of bookmarks), and to perform a longitudinal study to properly assess the value of POLLICINO and of collective code bookmarks.

ACKNOWLEDGMENTS

We thank the participants in our survey, interviews and user study. We also thank Andy Zaidman for his valuable input. Hattori is supported by the Swiss Science Foundation through the project "GSync" (SNF Project No. 129496).

REFERENCES

[1] E. Babbie. *The practice of social research.* Wadsworth Belmont, 11th edition, 2007.

[2] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz. Enriching reverse engineering with annotations. In *Proceedings of MoDELS 2008 (the 11th Intl. Conf. on Model Driven Engineering Languages and Systems)*, pages 660–674. Springer-Verlag, 2008.

[3] D. Campbell, J. Stanley, and N. Gage. *Experimental and quasi-experimental designs for research.* Rand McNally, 1963.

[4] L.-T. Cheng, M. Desmond, and M.-A. Storey. Presentations by programmers for programmers. In *Proceedings of the ICSE 2007 (29th Intl. Conf. on Softw. Eng.)*, pages 788–792. IEEE Computer Society, 2007.

[5] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[6] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 2011.

[7] B. Dagenais and H. Ossher. Mismar: A new approach to developer documentation. In *Proceedings of ICSE 2007 (29th Intl. Conf. on Softw. Eng. - Companion*, pages 47–48. IEEE Press, 2007.

[8] A. Guzzi, L. Hattori, M. Lanza, M. Pinzger, and A. van Deursen. Collective code bookmarks for program comprehension – online appendix. http://www.st.ewi.tudelft.nl/~guzzi/pollicino/user-study-1/.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM Intl. Conf. on Softw. Eng.)*, pages 492–501. ACM, 2006.

[10] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23:76–83, 2006.

[11] C. Oezbek and L. Prechelt. Jtourbus: Simplifying program understanding by documentation that provides tours through the source code. In *ICSM 2007 (Intl. Conf. on Softw. Maintenance)*, pages 64 –73. IEEE Press, 2007.

[12] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proceedings of WCRE 2004 (11th Working Conf. on Reverse Engineering)*, pages 70–79. IEEE CS Press, 2004.

[13] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings of ICSE 2008 (30th Intl. Conf. on Softw. Eng.)*, pages 251–260. ACM Press, 2008.

[14] M.-A. Storey, J. Ryall, J. Singer, D. Myers, L.-T. Cheng, and M. Muller. How software developers use tagging to support reminding and refinding. *IEEE Trans. Softw. Eng.*, 35:470–483, 2009.

[15] B. van Rompaey and S. Demeyer. Estimation of test code changes using historical release data. In *Proceedings of WCRE 2008 (15th Working Conf. on Reverse Engineering)*, pages 269–278. IEEE Computer Society, 2008.

[16] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 1995.

[17] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *Intnl. Conf. on Softw. Testing, Verification, and Validation)*, pages 220–229. IEEE, 2008.