Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Refining Code Ownership With Synchronous Changes

Lile Hattori · Michele Lanza · Romain Robbes

Received: date / Accepted: date

Abstract When mining software repositories, two distinct sources of information are usually explored: the history log and snapshots of the system. Results of analyses derived from these two sources are biased by the frequency with which developers commit their changes. We argue that the usage of mainstream SCM (software configuration management) systems influences the way that developers work. For example, since it is tedious to resolve conflicts due to parallel commits, developers tend to minimize conflicts by not contemporarily modifying the same file. This however defeats one of the purposes of such systems.

We mine repositories created by our tool *Syde*, which records changes in a central repository whenever a file is compiled locally in the IDE (integrated development environment) by any developer in a multi-developer project. This new source of information can augment the accuracy of analyses and breaks new ground in terms of how such information can assist developers.

We illustrate how the information we mine provides a refined notion of code ownership with respect to the one inferred by SCM system data. We demonstrate our approach on three case studies, including an industrial one. Ownership models suffer from the assumption that developers have a perfect memory. To account for their imperfect memory, we integrate into our ownership measurement a model of memory retention, to simulate the effect of memory loss over time. We evaluate the characteristics of this model for several strengths of memory.

Keywords Code ownership · Mining software repositories · Fine-grained changes · Software visualization

We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "GSync" (SNF Project No. 129496).

Lile Hattori · Michele Lanza
REVEAL @ Faculty of Informatics, University of Lugano
Via G. Buffi 13 - 6904 Lugano, Switzerland
Tel.: +41 58 666 42 93, Fax: +41 58 666 45 63
E-mail: {lile.hattori,michele.lanza}@usi.ch

Romain Robbes
PLEIAD Lab, Computer Science Department (DCC), University of Chile
Blanco Encalada 2120, of. 308, Santiago, Chile
Tel.: +56 2 978 4974, Fax: +56 2 689 5531
E-mail: rrobbes@dcc.uchile.cl

1 Introduction

To manage the life-cycle of software systems, developers use a number of tools, such as software configuration management (SCM) systems, bug trackers, discussion boards, *etc.* These tools store a large amount of information that is exploited by researchers to understand different aspects of software evolution. SCM repositories, in particular, are a rich source of information because they contain both the history of the source code and metadata describing who was responsible for which change.

A significant number of studies have mined SCM repositories to reveal the nature of software changes [**?**,**?**,**?**], and to understand the correlation between changes and developer roles [**?**,**?**]. These studies are based on largely adopted SCM systems, such as CVS and Subversion (SVN). However, any inference derived from such systems is subject to the granularity of information encountered in their repositories.

In their report on the impact of SCM systems, Estublier *et al.* stated that one of the next steps for SCM systems was to break the assumption of language independence [**?**]. Contradicting this statement, largely adopted SCM systems are still file-based and do not model the particularities of a programming language. Hence the changes to software entities must be reconstructed from the text-level changes stored in the SCM. Combined with the *check-out/checkin* commands, where a developer checks out the code before an implementation session, and checks in the changed files after an indefinite period of time, SCM systems lose precious information about source code changes that cannot be recovered even with elaborate mining and reverse engineering techniques [**?**].

Since checking in source code is an intermittent action and development is a continuous activity, knowledge derived from the history log may deviate from what actually happened. For example, a technique that spots specialists for parts of a system based on check-in frequency does not take into account the actual effort spent by developers in terms of time and written code. Also, the frequency with which developers check in their code is biased by the lack of language-oriented support for merging parallel changes. Since a developer does not know whether someone else is changing the same file, studies have shown that they tend to rush to check in their code [**?**], and even check in partial changes [**?**] to avoid dealing with merge conflicts.

Modern decentralized software control management systems, such as Git[1], offer additional support for parallel development. In Git the check-out/check-in model is replaced by the clone/pull/push model, with which every developer maintains his own repository by cloning someone else's repository. Different from file-based SCM systems, which track changed files by their names, Git is content-based, which means it tracks changed files by the contents of their changes. One advantage of this approach is that Git can track the rename of a file, contrary to CVS or SVN, for example. However, some of the consequences of Git's decentralized model are that commits are not automatically visible to other developers; instead of one history log, there are as many as the number of repositories created; the logs of privately owned repositories are not accessible to everyone; and all the changes created in everyone's private repositories have to be occasionally merged, and conflicts resolved [**?**].

The nature of information found in software repositories determines what we can infer from it [**?**]. File- and content-based SCM systems store snapshots that represent the system's state at points in time, rather than a continuous evolution of the changes made to the system to bring it from one state to the other. We believe that studies derived from file- and content-

[1] See `http://git-scm.com/`

based SCM systems are threatened by the loss of information that comes with the underlying models.

We propose the use of a new software repository that is created by our Syde tool to overcome the limitations of current SCM data. Syde is a collaborative tool that extends Spyware's [**?**] change-centric approach to augment the awareness of a team of developers by propagating changes *as they happen* [**?**]. Syde offers to developers a collection of Eclipse plug-ins to keep them aware of which source code artifacts are being changed [**?**], and which are the source code changes that can impact on someone's current work [**?**]. It runs concurrently with the project's SCM system and does not obstruct or modify its usage. Syde's repository stores every change performed by every developer at the exact time it happens. We define a change as every successfully compiled file that has undergone at least one character edit since the last compilation (See Section 3). Hence, the once approximate data about *who* changes *what* and *when* is now accurate and complete; leading to what we call *synchronous changes*.

In this paper we describe how we used Syde's change history together with the history logs of these projects to understand the dynamics of the developers, and to create a refined notion of ownership of the code. Further, we account for the imperfect memory of developers by integrating in our model of ownership the notion of forgetting: A developer who changed a file early on may have less actual knowledge of it than another developer who changed it more recently, even if it was changed less. To conduct this study, we used Syde to record several development periods in three systems, including a commercial one.

We envision using the technique and findings of this study as a foundation to integrate an expertise recommender to the set of plug-ins that Syde offers. The recommender will assist developers to search for help when striving to understand a piece of code. In such cases, the recommender will show a list of developers who are knowledgeable about the code artifact, ranking them based on their current knowledge. Thus, we aim at bringing a traditionally *post mortem* analysis to forward engineering to help developers maintain team awareness.

Structure of the paper. In Section 2 we review related work. In Section 3 we detail our change recording and broadcasting approach and its supporting implementation in the form of Syde. In Section **??** we describe the notion of ownership of file by developers, and the effect of time in their ability to recall information. In Section **??**, we describe a visualization we use as support for interpreting the ownership data, the Ownership Map. In Section **??** we then use Syde to analyze the history of three software systems, before discussing threats to the validity of our study in Section **??**. Finally, we conclude in Section **??**.

2 Related Work

With Syde, we essentially propose a chance-centric approach to promote team collaboration. Thus it is related to:

1. tools that support collaboration, and
2. operation-based SCM solutions.

In this work we are primarily interested in the impact of the data captured by Syde on code ownership; we review the literature on that domain as well.

2.1 Tool Support for Collaboration

The continuous adoption of language independent SCM systems in the context of team-based development influenced the creation of solutions to overcome the workspace isolation enforced by them [**?**]. Tool support for collaboration ranges from full-fledged platforms, such as Jazz.net [2] and CollabVS [**?**], to specific workspace awareness solutions [**?**,**?**,**?**].

Jazz.net is designed to be the central tool for planning, managing and performing development activities. It enriches Eclipse and Visual Studio to create a new environment to support intra and inter-team collaboration, automation, and traceability of code, tasks and issues. Microsoft's CollabVS extends Visual Studio by adding communication channels, such as text and audio-video chat, browsing of remote unchecked versions of files, and notification of developer presence in code elements inside a file [**?**].

There are a number of valuable efforts to solve some specific problems raised by workspace isolation generated by SCM systems. More specifically, they recover and broadcast information about changes that occur between a check out and a check in, which tends to become more critical as the gap grows larger.

Lighthouse is an Eclipse plug-in that aim at avoiding conflicts by propagating change events from Eclipse and SCM among workspaces, and showing them on a view of the emerging design representation of the system [**?**]. Lighthouse requires a side-by-side presentation of the design representation and the code, which is only feasible if developers work with two screens.

Palantír is an Eclipse plug-in that addresses direct and indirect merge conflicts [**?**]. Direct conflicts are caused by concurrent changes to the same artifact. Indirect ones are caused by changes in one artifact that affect concurrent changes in another artifact. Palantír informs the involved developers about the existence of conflicts, and their severity (*e.g.*, it is high if one of the conflicting versions has already been checked in).

Schneider *et al.* use a shadow CVS repository to record changes every time that someone edits a file. The shadow repository is then mined and information about who is working with what is visually presented to developers to augment group awareness [**?**].

FASTDash offers to developers real-time information about changes: which team members have source files checked out, which files are being viewed, and which classes and methods are currently under change [**?**].

The demand for workspace awareness is becoming urgent as intensive and globally distributed team collaboration becomes the state of the practice. Although the solutions discussed above increase workspace awareness by working around some of the limitations imposed by SCM systems, the root of the problem lies in the currently used SCM models, which offer insufficient support for collaboration.

2.2 Operation-based SCM

The key characteristic of file-based SCM systems is that they are able to version any type of document, since documents are represented as files in a computer. In the context of software development, this rather strong feature comes with a tradeoff: they are unable to model, and hence, properly version source code changes. Source code is treated as plain text, which forces developers to deal with textual merging of source code, with consequences that range from compilation errors to bugs generated from runtime errors.

[2] See `http://jazz.net`

On the other end of the spectrum there are language-dependent operation-based SCM systems [**?**,**?**,**?**], which have support for the language model, and version the system as a sequence of change operations. Some advantages of this approach are that operations can be replayed or rewound to bring the system from one state to another, and merge conflicts can be resolved with operation-based merge algorithms [**?**]. However, despite a few noteworthy efforts to provide operation-based SCM solutions, there is still a list of issues to be addressed until they become fully functional.

For example, MolhadoRef, proposed by Dig *et al.* [**?**], is not a pure operation-based SCM, but a mixture of state-based and operation-based, i.e., it does not record every change made by every developer. Instead, it calculates the deltas before changes are checked in; only refactoring operations are fully recorded. Consequently, there is still loss of information, and not all system states can be recovered from the MolhadoRef repository.

In previous work, we introduced Spyware, a change-centric solution that records every change made by one developer. Spyware is able to recover any state of the system [**?**]. Its main restriction is that it is a one-developer solution, *i.e.,* it does not support a multi-developer context. Our goal with Syde is to port Spyware's approach to a multi-developer context without losing information. Like Spyware, we do not intend to replace file-based SCM systems, but to complement them by storing additional information.

2.3 Ownership of Files and Expertise of Developers

There have been several works on determining which developer is the most expert in a given area of a large software system. The rationale behind these approaches is that since no one can be knowledgeable over the entire system, one can instead identify who are the people to contact to get more information about a given part of the system – the experts. People have used several data sources to determine the expertise of developers.

Several approaches use SCM data to compute expertise and ownership. They assume that people gain expertise on a part of the system when they change its implementation. McDonald and Ackerman used SCM author information to determine the expertise of developers. They also included technical support data that was available in their particular case study [**?**]. Mockus and Herbsleb in contrast, used only change data from the SCM system as the data source of their Expertise Browser [**?**]. Gîrba *et al.* focused more particularly on *ownership*, where the owner of a file is the developer with the most expertise on it. They also used SCM data to compute ownership [**?**].

Other approaches use different data sources. Anvik and Murphy used bug archive data to determine implementation expertise, and found that it can serve as a replacement for SCM data in the cases where the latter is not accurate [**?**]. Matter *et al.* determined the expertise of developers based on the vocabulary they use. They used that expertise information to assign bugs to developers [**?**]. Finally, Ma *et al.* introduced the usage expertise, where the expertise of people using a given piece of code is taken into account, as opposed to the expertise of the people who implemented it [**?**].

3 Syde

Syde is a client-server application that manages and stores object-oriented software systems implemented in Java. The client is a collection of Eclipse plug-in that both inspect the developer's workspace and enrich Eclipse with visualizations that provide awareness information

to developers. Figure **??** shows some of the visualizations provided by Syde, which are currently grouped into three plug-ins: Inspector Plug-in – responsible for tracking changes –, Scamp Plug-in – delivers change awareness information –, and Conflicts Plug-in – notifies developers of potential conflicts.

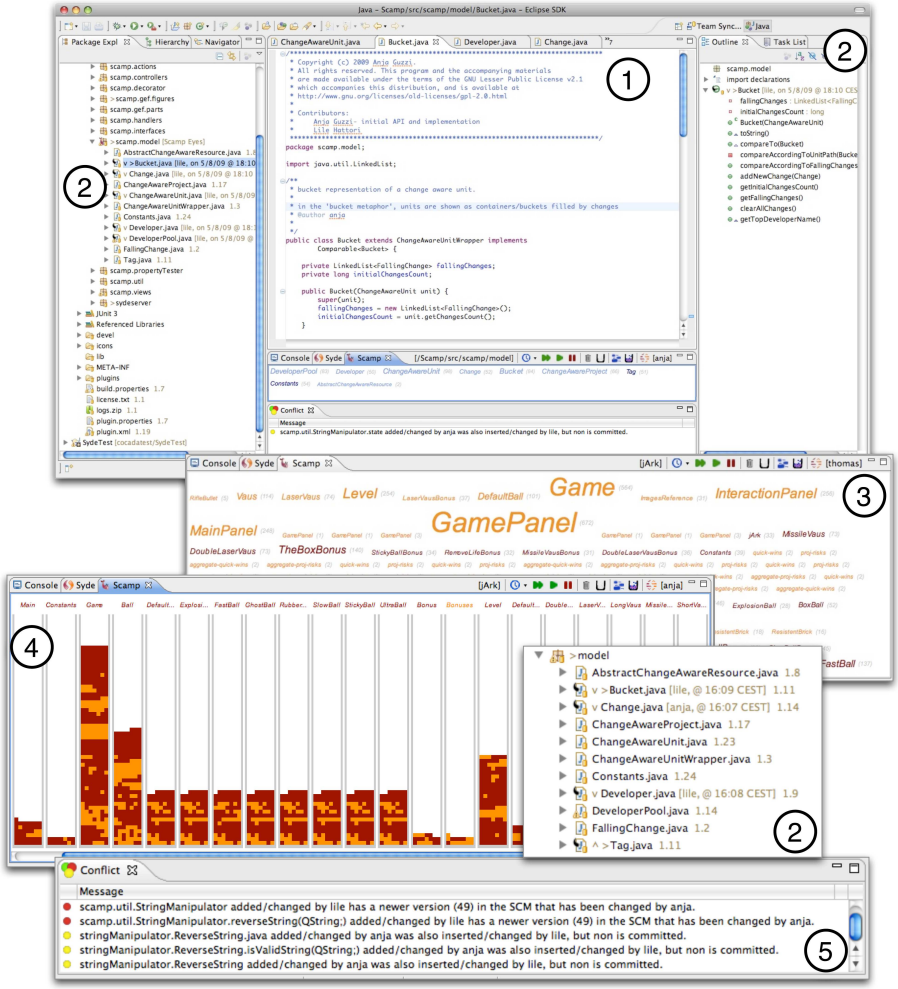

Fig. 1 Syde screenshots. 1: The Inspector Plug-in. 2: Scamp Plugin - Decorations View. 3: Scamp Plugin - WordCloud View. 4: Scamp Plugin - Buckets View. 5: The Conflict Plug-in - Conflicts View. 6. The Conflict Plug-in - Annotation on Java Editor

The Inspector plug-in listens to “build” operations, which are often linked to “save” operations in Eclipse. Every time a developer compiles a changed file, the Inspector’s listeners are triggered and send a new version of the file to the server. If the file does not successfully compile, a notice of unsuccessful compilation together with the changed file is sent to the server. The server saves the received information and broadcasts information about the

(successfully compiled) changes to all active client instances. Finally, each client instance of the plug-in displays the broadcasted change information on the views inside Eclipse's workbench. We developed Syde with a number of goals in mind:

– *Complement SCM systems.* As stated before, our goal is to complement file-based SCM systems. A software project comprises not only source code, but also requirements, specification, etc. For now, Syde focuses exclusively on the source code of a project.
– *Be non-intrusive and lightweight.* Syde displays the activity of other developers in views that the developer can simply close or minimize. Thus, Syde provides extra information without disrupting or distracting a developer from work. As opposed to holistic and complex approaches such as Jazz.net or CollabVS our goal is to provide effective collaboration support with minimal, lightweight, and complementary changes to the settings of the developers.
– *Enhance awareness.* Similar to other solutions to augment workspace awareness, Syde informs developers about recent changes in the source code that may not have been checked into the SCM repository yet. A developer can request these changes even before they become available through the SCM. We conducted a qualitative evaluation of our tool set to provide awareness [**?**], where developers indicated that they appreciated the benefits of knowing what others were doing in real-time.
– *Enrich SCM history.* Similar to the history logs of CVS or Subversion, Syde provides the history of changes with the following information: changed file, author, and timestamp. The fundamental difference is that Syde provides a historic entry for every change performed on Eclipse, even if they were not checked in lately.

3.1 Design & Implementation

Syde's overall design and information flow is illustrated in Figure **??**.

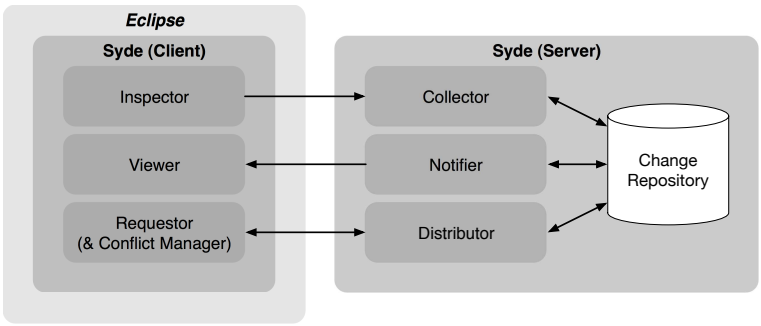

Fig. 2 Syde Architecture.

Syde features the following components:

– *The Inspector and the Collector.* Syde's inspector implements listeners to capture from Eclipse's workbench the changes performed by a developer. The inspector collects the actual changes and the metadata. The metadata contains the author's name, a timestamp, and status of the change. Syde's collector receives information from the inspector and stores it in a centrally accessible repository.

– *The Notifier and the Viewer.* Syde's notifier maintains a list of client instances that need to be notified of any change, and is responsible for broadcasting the metadata to all members of the team. Syde's client features display information about the changing system in views and visual cues within Eclipse itself, thus providing awareness of changes to all developers.
– *The Distributor and the Requestor.* Once a developer is aware that certain parts of the system have changed, he can preempt the underlying classical SCM system and request from the Syde server an update of specific parts of the code, which are then sent by Syde's distributor, and updated in the client's source base.

We implemented Syde in Java for the Eclipse IDE, with the goal of complementing the workspace awareness offered by SCM systems. However, its implementation does not depend on a specific SCM system, and can also be used without one. To explain Syde's implementation, we follow the information flow illustrated in Figure **??**.

The *Inspector* is located in Syde's plug-in and is responsible for monitoring and sending source code changes to the server. To inspect source code changes, it relies on the following strategy. If the project under inspection uses the standard *Java Builder* for compilation, the *Inspector* implements the `IResourceChangeListener` interface to listen to `POST_BUILD` events. Before it sends the changed file to the server, it checks for compilation errors inside the file, and annotates the metadata with this information. Even though the changed file can group more than one source code change, we argue that this is a reasonable approach because a developer tends to save (and automatically compile) changes frequently enough for differencing algorithms, such as the one proposed by Fluri *et al.* [**?**], to be able to precisely find all changes from two subsequent versions. If the project does not use the *Java Builder*, the *Inspector* listens to `POST_CHANGE` events, and is therefore unable to check for compilation errors in the file.

On the server side, the *Collector* (1) receives the file, (2) versions and saves it, (3) saves the metadata, and (4) activates the *Notifier*.

The *Notifier* manages which developers are connected to Syde for a given project by keeping a set of projects and, for each project, a set of developers. Immediately after a new version of a file is available on the server, it broadcasts an alert to all developers. To show the alerts on Eclipse, the *Viewer* is currently composed of two plug-ins: Scamp and Conflicts. Finally, the *Requestor* adds the action "Get last version" to one of Syde's views, which requests from the *Distributor* the latest version of the selected file in the view.

Data. The history log of mainstream SCM systems usually describes which files have been checked in, when and by whom. For example, CVS history logs show: file name, revision, author, timestamp, author's comment, and number of lines of code added and removed. Subversion (SVN) gives the same information, except for the number of lines of code added and removed. On the other hand, SVN automatically groups file revisions committed together, whereas this must be reconstructed for CVS data.

Syde's history log offers the same kind of information[3], but for *every change* performed by a developer. It shows file name, revision, author, timestamp, and whether the file has compilation errors or not.

[3] In the meantime, Syde records more and more fine-grained information, whose description is outside the scope of this article.

4 Code Ownership

Code ownership quantifies the amount of knowledge each developer has and indicates which developer owns which artifact of a software system by measuring who has accumulated more knowledge of each artifact. The notion of code ownership is important in large projects, where developers do not know each artifact of the system. Code ownership can be used to answer questions such as "who should fix this bug" [**?**] or "who should I ask about artifact XY" [**?**].

We present the measurements that we use to compute code ownership. The general assumption is that whoever performs the greater number of changes on a file, is the one most knowledgeable in it. We use distinct measurements of code ownership based on three different repository sources: CVS, SVN, and Syde. For CVS and SVN, we adopt the measurement previously introduced by Gîrba *et al.* [**?**], whereas for Syde, we present a lightweight approach to compute code ownership by mainly using historical information contained in the Syde change logs.

4.1 Measuring Code Ownership with CVS/SVN Logs

The ownership definition based on CVS and SVN logs exclusively uses information contained in their respective logs: file name, revision, author, and number of lines added and deleted. Since SVN logs do not contain the number of lines added and deleted for each revision of a file, we implemented a parser to extract this information by comparing every subsequent revision of a file in the repository.

According to Gîrba *et al.* [**?**], a developer owns a line of code in a file if he was the one who committed that line. The overall owner of a file is the one who owns the largest percentage of it. To compute the ownership, we need to first estimate the size of a file. We only know the number of lines added and deleted, but do not know the initial size of the file, because we only use information containing in the log history[4]. Given a file f, let f_n be a revision of f, α_{f_n} be the author of that revision, a_{f_n} be the number of lines added, and r_{f_n} the number of lines removed. The size of a file revision s_{f_n} is given by:

$$
\begin{aligned}
s'_{f_0} &= 0 \\
s'_{f_n} &= s'_{f_{n-1}} + a_{f_{n-1}} - r_{f_n} \\
s_{f_0} &= |min\{s'_x\}| \\
s_{f_n} &= s_{f_{n-1}} + a_{f_n} - r_{f_n}
\end{aligned}
$$

To exemplify the size estimation of a file, suppose the following sequence of changes:

f_1: 8 lines added, 3 lines deleted;
f_2: 7 lines deleted.

We apply the given formulae to estimate the sizes of the files:

$$
\begin{aligned}
s'_{f_0} &= 0 \\
s'_{f_1} &= 0 + 8 - 3 = 5
\end{aligned}
$$

[4] In our case study, we have an industrial system that uses CVS, and two academic systems that use SVN. For the first, we only had access to the logs, but not to the source code. Thus, we decided to consistently apply the same approach to all three projects, by estimating the size of a file.

$s'_{f_2} = 5 + 0 - 7 = -2$

Since there can not be more number of lines deleted than added in total, we need to adjust the above values.

$s_{f_0} = |min\{-2, 5\}| = 2$
$s_{f_1} = 7$
$s_{f_2} = 0$

Given the size s_{f_n} of a file revision, the percentage $own^{\alpha}_{f_n}$ of lines in a revision owned by a developer α is given by:

$$own^{\alpha}_{f_0} = \begin{cases} 1 & \text{if } \alpha = \alpha_{f_0} \\ 0 & \text{else} \end{cases}$$

$$own^{\alpha}_{f_n} = own^{\alpha}_{f_{n-1}} \frac{s_{f_n} - a_{f_n}}{s_{f_n}} + \begin{cases} \frac{a_{f_n}}{s_{f_n}} & \text{if } \alpha = \alpha_{f_0} \\ 0 & \text{else} \end{cases}$$

From the percentage $own^{\alpha}_{f_n}$ of lines owned by each developer, the owner of a file revision is the one who owns the greatest percentage.

This measurement technique relies on the assumption that the number of lines of a file owned by a developer reflects the amount of effort spent by him to write these lines. However, the development of a code artifact is not a linear action that can be summed up as the amount of lines added to a file. Developers do and undo changes, try a couple of alternatives, refactor the code –which may reduce the size of the file–, *etc.* The knowledge they retain from an artifact depends more on how much effort they put to implement it, than on the final result. Thus, this technique can be effective when developers check in their files frequently. However, if within a team there are developers who frequently check in their changes and others who work for long periods before checking in, this technique is prone to discrepancies.

4.2 Measuring Code Ownership with Syde Logs

We use the history logs provided by Syde to measure code ownership, therefore basing the definition of ownership on every small change that is being performed on a system. *Every small change* means every change performed between two save – and consequently compilation – actions. One can argue that the definition of ownership using Syde logs, analogous to what happens with CVS and SVN logs, is also biased by the frequency with which developers save their code. However, the way Eclipse works drives developers to maintain their code with no compilation errors and hence to save and build files often. Eclipse constantly shows where compilation errors are, discouraging developers to run or test code with errors, and in general incites them to fix them as soon as possible.

Given Syde logs, let f be a file, f_n a version of this file, α_{f_n} the author of that version, the number $own^{\alpha}_{f_n}$ of changes owned by a developer α is given by:

$$own^{\alpha}_{f_0} = \begin{cases} 1 & \text{if } \alpha = \alpha_{f_0} \\ 0 & \text{else} \end{cases}$$

$$own^{\alpha}_{f_n} = own^{\alpha}_{f_{n-1}} + \begin{cases} 1 & \text{if } \alpha = \alpha_{f_n} \\ 0 & \text{else} \end{cases}$$

The owner own_{f_n} of a file at a certain version is the one who has accumulated the largest number of changes from the creation of the file until the date of the considered version:

$own_{f_n} = max\{own_{f_n}^{\alpha_1}, own_{f_n}^{\alpha_2}, ..., own_{f_n}^{\alpha_m}\}$, where m is the total number of developers.

To exemplify how to compute the ownership of a file with Syde log, we show the change history of a hypothetic file *Foo* in Figure **??**.

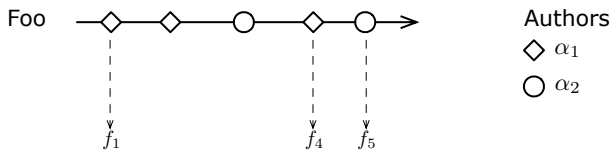

Fig. 3 Change history of file Foo. This file is currently in version 5 and two developers have changed it.

In this example, we want to find the owner of *Foo* by the time version f_5 is created. The number of changes owned by each author at this point is:

$own^{\alpha_1} = 3$
$own^{\alpha_2} = 2$

The greater number own_{f_5} of changes owned of file f at revision f_5 is:

$own_{f_5} = max\{own^{\alpha_1}, own^{\alpha_2}\} = own^{\alpha_1}$

Hence, the owner of file f at revision f_5 is developer α_1.

This measurement assumes that developers accumulate knowledge about a class, or file, but never forget it, even though months or years have passed. This is a rather naïve assumption, since the content of files might change over time. Suppose a developer creates a file and with 30 Syde changes he introduces 10 lines of code. Later, a second developer edits the same file and, with 15 Syde changes, he completely changes the 10 lines of code. The current ownership measurement based on Syde changes still considers the first developer as most knowledgeable, whereas the measurement based on CVS/SVN commits considers the second as most knowledgeable because he currently owns the greater percentage of lines (assuming that the total number of lines remains constant, the second developer is the owner of 100% of the lines).

In addition, there is a natural process of forgetting the content and functionality of a class over time, even though they do not change. To address these issues, we add the notion of forgetting on the code ownership measurement based on Syde log. Although the measurement based on CVS/SVN logs also ignores the natural process of forgetting, we focus exclusively on investigating the forgetting effects on the ownership measurement based on Syde logs.

The Forgetting Effect on Code Ownership. Forgetting is a natural process in which old memories are unable to be recalled from a human's memory. It has been extensively studied

by psychologists, since the pioneering work of Ebbinghaus [**?**]. The curve that commonly describes forgetting is expressed as $R = e^{-T/s}$, where R is the memory retention, s is the relative strength of memory, and t is time. Although there has been a continuous discussion on whether forgetting is best described by a power or exponential function [**?**,**?**,**?**], the differences represented by each curve can be considered minimal for the context of ownership measurement. Thus, we consider the exponential formula introduced above.

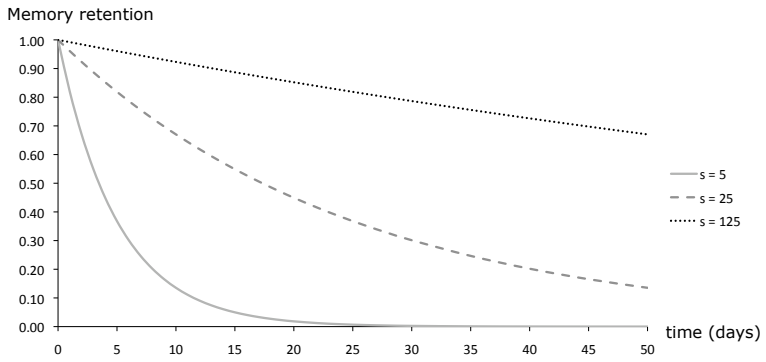

Fig. 4 Forgetting function $R = e^{-T/s}$, where R is the memory retention, s is the relative strength of memory, and t is time. The higher is the value of S, the more likely the person will remember an event for a longer period.

Figure **??** shows the plot of the forgetting function for three different values of s (5, 25, and 125). In the context of code ownership, our time unit is the day. The s parameter reflects how long a person might remember the contents of a certain file. The smaller s is, the weaker the memory. For instance, for $s = 5$, there is around 30% of probability that a person remembers the contents of a file edited 6 days ago (see Table **??**, page **??**, for other figures). This situation could be true in a scenario where this person is developing multiple systems at the same time, and only makes small changes, which are not sufficient for him to absorb concrete knowledge of each class. Therefore, the parameter s is influenced by a number of factors that are specific to each project, *e.g.*, the complexity of the project or feature, the level of experience of each individual, the accumulated experience of each individual in the context of the project, the total number of developers, *etc.* Hence, it is not our goal to determine an ideal s value for all projects, but to explain how it can be adjusted according to each scenario.

We include the notion of forgetting in the code ownership measurement by adding the time factor in the calculation of the owner $own^{\alpha}_{f_n}$ of a revision. In the previous formula, the value 1 was given to a developer for each change he made. With the forgetting measurement, we use the formula $R = e^{-T/s}$ to weight this value, which now has a range of $(0, 1]$.

To formalize the new measurement, for each version f_n of a file f, we consider the time t_{f_n} of the creation of version f_n. Given a point in time t, the argument T of the forgetting function $R = e^{-T/s}$ is the amount of time elapsed between the creation of version f_n and the current time t:

$$\Delta t_{f_n} = t - t_{f_n}$$

Hence, given a point in time t, the new ownership measurement is computed as follows:

$$\Delta t_{f_0} = t - t_{f_0}$$
$$own^{\alpha}_{f_0} = \begin{cases} e^{\frac{-\Delta t_{f_0}}{s}} & \text{if } \alpha = \alpha_{f_0} \\ 0 & \text{else} \end{cases}$$

$$\Delta t_{f_1} = t - t_{f_1}$$
$$own^{\alpha}_{f_1} = own^{\alpha}_{f_0} + \begin{cases} e^{\frac{-\Delta t_{f_1}}{s}} & \text{if } \alpha = \alpha_{f_1} \\ 0 & \text{else} \end{cases}$$

$$\Delta t_{f_n} = t - t_{f_n} = 0$$
$$own^{\alpha}_{f_n} = own^{\alpha}_{f_{n-1}} + \begin{cases} e^{\frac{-\Delta t_{f_n}}{s}} & \text{if } \alpha = \alpha_{f_n} \\ 0 & \text{else} \end{cases}$$

The owner own_{f_n} of a file at a certain version is the one who has accumulated the greatest value of weighted knowledge from the creation of the file until the time t_{f_n} of the considered version:

$own_{f_n} = max\{own^{\alpha_1}_{f_n}, own^{\alpha_2}_{f_n}, ..., own^{\alpha_m}_{f_n}\}$, where m is the total number of developers.

Recalling the example from Figure **??**, we added the notion of time in Figure **??** to recompute the ownership considering the forgetting effect. Analogous to previous example, we want to find the owner of *Foo* at the time t_{f_5} of the creation of version f_5.

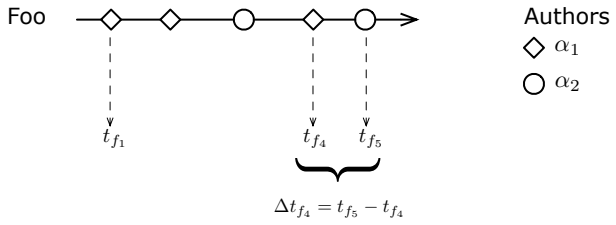

Fig. 5 Change history of file Foo with additional notion of time.

Hence, the value of weighted knowledge accumulated by each developer, given the strength of memory $s = 2$, is:

Let $t_{f_1} = 1, t_{f_2} = 2, t_{f_3} = 3, t_{f_4} = 4, t_{f_5} = 5$,

$$own^{\alpha_1}_{f_5} = own^{\alpha_1}_{f_4} + 0 = e^{\frac{-\Delta t_{f_1}}{s}} + e^{\frac{-\Delta t_{f_2}}{s}} + 0 + e^{\frac{-\Delta t_{f_4}}{s}} + 0 = e^{\frac{-4}{2}} + e^{\frac{-3}{2}} + 0 + e^{\frac{-1}{2}} + 0 \simeq 0.96$$

$$own^{\alpha_2}_{f_5} = own^{\alpha_2}_{f_4} + 1 = 0 + 0 + e^{\frac{-\Delta t_{f_3}}{s}} + 0 + e^{0} = 0 + 0 + e^{\frac{-2}{2}} + 0 + 1 \simeq 1.37$$

The greater value own_{f_5} of knowledge accumulation for file f at revision f_5 is: $own_{f_5} = max\{own^{\alpha_1}, own^{\alpha_2}\} = own^{\alpha_2}$

Even though developer α_1 has the greater number of changes, if we consider the forgetting effect on knowledge retention and the recency of changes, the owner of file f at revision f_5 is developer α_2.

With this new measurement of code ownership, it is possible to adjust the strength of memory s to prioritize recent changes over old ones. The range of s most suitable for a project depends on a number of factors intrinsically related to its characteristics. In Section **??** we apply and discuss the influence of the forgetting effect on ownership maps of three projects.

5 Ownership Maps

To visually assess the differences in ownership between versioning system changes and Syde changes, we display the data using ownership maps.

The Ownership Map visualization was first introduced by Gîrba *et al.* [**?**] with the aim of characterizing behavioral patterns of developers throughout the life-cycle of a software system. Gîrba's Ownership Map was inspired by the visualization proposed by Rysselberghe and Demeyer [**?**], where the horizontal axis represented each file of the system, and the vertical axis represented the time of a change.

In the original ownership map, each line represents a file in the system and time is represented on the horizontal axis. Every change to a file is shown as a colored disc. The color of the disc represents the developer who made that change. Finally, the line of the file is colored according to its current owner. A file can have multiple owners throughout its history.

We use three distinct ownership maps: *CVS/SVN Ownership Map*, which presents the ownership of files according to CVS/SVN commits; *Syde Ownership Map*, which shows the ownership of each file according to Syde changes; and *Delta Map*, which illustrates the differences in ownership classification of the previous two maps. In the following, we detail each map, and explain how we order the files in the maps.

5.1 CVS/SVN Ownership Map

In the CVS/SVN ownership map, each rectangle represents a commit [5]. Each developer is represented by a unique color, which is used to indicate who is the author of a commit – coloring the rectangle –, and who is the owner of the file at a certain period of time – coloring the corresponding part of the line.

Figure **??** illustrates an example of this map. Recall that the measurement used to compute ownership for CVS/SVN is based on the number of lines added and deleted from each version, and takes into account the entire history of the file evenly.

[5] We use rectangles instead of circles for SCM logs so that one can differentiate between SCM changes and Syde changes when both are overlapped on the same map.

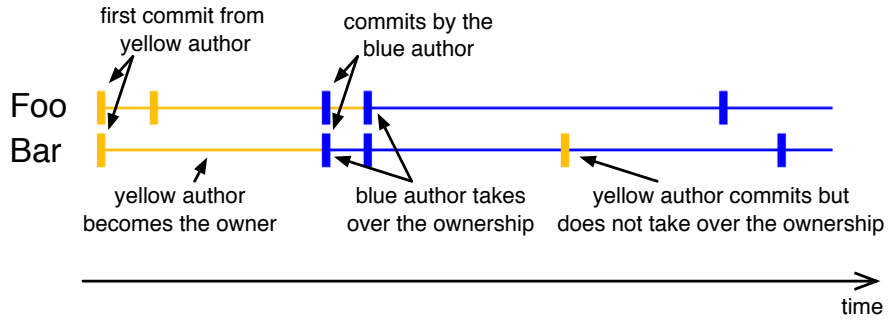

Fig. 6 Example of CVS/SVN ownership map.

5.2 Syde Ownership Map

In the Syde ownership map, a colored disc represents a Syde-level change, where the color indicates the author of the change. The line of the file is painted with the color of its owner at a certain period of time.

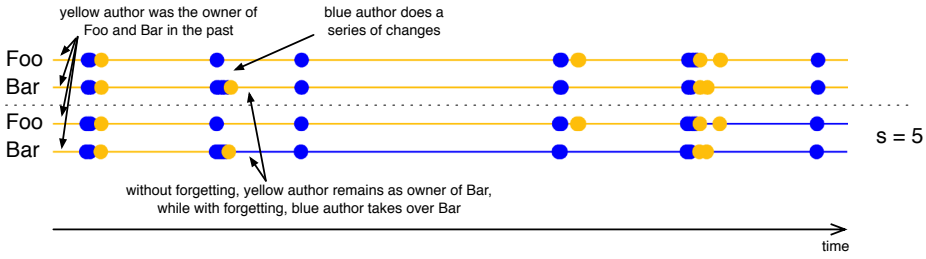

Fig. 7 Example of Syde ownership map.

Figure **??** shows two Syde ownership maps. The first uses the original Syde ownership measurement, while the second integrates the forgetting notion, using a value of $S = 5$ to model the strength of memory. It is easy to spot the differences in ownership between the two maps. Consider file Bar, which experiences a series of changes from the blue author early on. In the first case, the yellow author remains the owner of Bar, but in the second, the blue author takes over. We discuss in Section **??** the differences in ownership according to how fast a developer might forget the contents of a file. When forgetting is considered, the ownership of a file might change at any moment, not just after a file has changed.

5.3 Delta Ownership Map

The Delta ownership map is a combination of the previous two, with rectangles representing CVS/SVN commits, and discs representing Syde changes. Discs and rectangles receive the color of the author of the change or commit. The lines are either grey –when there is no

difference in ownership classification between Syde and CVS/SVN measurements– or red –when there is a difference between the measurements.

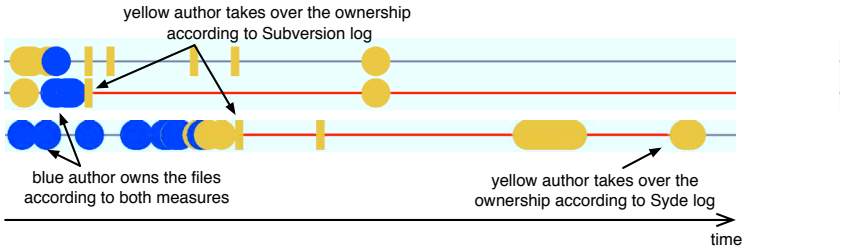

Fig. 8 Example of delta ownership map.

Figure **??** shows an example of the delta map. In this example, the yellow author is the sole owner of the first file, even though the blue author has one Syde change. For the second and third files, the blue author is the owner according to Syde changes. Yellow takes over the files according to CVS/SVN measurement after committing changes on them. After a sequence of Syde changes, Yellow takes over the third file, ending the differences in classification in it. Although not seen in this example, the difference in ownership classification is oftentimes caused by the instant propagation of changes in Syde against the latency in propagation caused by CVS/SVN commits.

5.4 Ordering of the Files

The order of the files in the ownership map has a large effect on its legibility. We order the files by their similarity in their change history, instead of using the standard alphabetical order. Using this ordering, the relationships between the files stand out much more, as files changing at similar times – inherently related to one another – are grouped.

To measure the similarity of the change histories of two files, we use a variant of the Levenshtein distance [**?**], which measures the similarity between two sequences by counting the number of edit operations necessary to transform one into the other. There are three kinds of operations: Insertion, Deletion and Modification of a sequence of elements. These operations are defined on abstract sequences of items and are not related to SCM operations.

In our case, we use the Syde change histories of the files as sequences. The CVS/SVN maps use the Syde clustering as well, so that the ordering is the same and comparison between maps is easier. We split the change history in a series of time intervals, and count the number of changes that affected the file during that interval. This yields a sequence of change intensities ordered by time. We tried intervals of 1, 4, 12 and 24 hours, inspected the clusters produced in each case, and concluded that a 12 hours clustering produced the best results: clustering the greatest number of files that were modified together (*e.g.*, files that were modified together were put side by side more often with 12 hours intervals than with other values).

Each edit operation is associated with a cost. To account for the different nature of our data, our definitions of the costs vary from the common definition. The Levenshtein distance is often used to measure the distance between words – where one can assume that

the characters are independent –, whereas we are comparing patterns of changes with an intensity of changes in a given interval. Intuitively, the distance between two characters is constant, but this assumption does not carry over to changes. Measuring the difference of intensity of changes allows us to use a more precise distance metric.

We retain the standard costs of 1 for insertion and deletion operation of items in the sequence. These operations are primarily used to switch the order of two time intervals when items with the same values have nearly identical indices in the two sequences (*e.g.*, a burst of changes on file $f1$ occurred just before a similar burst of changes on file $f2$).

We use a different definition of the cost for the modification operation. The modification operation is used to alter the value of an item in the sequence so that it corresponds to the value of the item at the same index in the second sequence (*e.g.*, a burst of changes on file $f1$ at time $t1$ is slightly smaller than the burst of changes of $f2$ at the same time). We define the cost of a modification operation between two change amounts a and b as:

$$\mathit{ModificationCost}_{a,b} = \begin{cases} 1, & \text{if } a = 0 \text{ and } b > 0 \\ 1, & \text{if } b = 0 \text{ and } a > 0 \\ \frac{|b-a|}{10}, & \text{otherwise} \end{cases} \quad (1)$$

The rationale behind our choice is that moderate variations in the intensity of changes during an interval should result in a lower edit cost than larger ones. On the other hand, a transition from no changes to any number of changes is always costly.

After computing the Levenshtein distance, we use a hierarchical clustering algorithm to order the files with respect to their similarity according to the distance we defined. All the maps we show in the remainder of the article are ordered using this scheme.

6 Case Studies

In previous work, we performed an initial analysis of the history of projects developed by a single developer [**?**]. In the context of this article we use the data provided by Syde to tackle the following research question: *How can Syde's history log help to characterize code ownership?* To discuss this question we analyze three projects: Speed, jArk, and Pacman.

6.1 Presentation of the Projects

Speed[6] is a commercial project that was under development at the software factory of CPM-Braxis [7]. This software factory was chosen because of its professional characteristics: it has a well defined production process certified by CMMI-DEV 5 and ISO 9001:2000 standards; its projects adopt metrics, software reuse, and new technologies for delivering high quality products.

Pacman and jArk are group projects developed by students in the context of the "Programming Fundamentals 2" course given at the University of Lugano. Contrary to Speed, where we collected data for a brief period, the students used Syde for the entire duration of their projects. Hence, we have the full history of development of these projects.

Table **??** presents the data we gathered for the three projects we monitored in this study.

[6] We use pseudonyms to conform with the non-disclosure agreement.

[7] See `http://www.cpmbraxis.com`

Table 1 Projects studied, the period analyzed for each project, the number of java files that were committed to the SCM repository, the number of files with changes captured by Syde, the number of SCM commits, and the number of Syde changes.

Project	Period	Developers	Files in SCM	Files in Syde	Commits	Syde Changes
Speed	15 days	4	14	56	26	2,429
jArk	5 weeks	2	146	172	266	23,786
Pacman	5 weeks	2	140	223	149	14,460

The number of files with Syde changes is higher than the number of files committed to CVS or SVN. We examined the source code and observed that the main cause of this disparity is when developers create a class, work on it for a while, but change its location before checking it in the repository.

An immediate observation we can make is that the changes recorded by Syde are much more fine-grained than the ones recorded by CVS or SVN: There are two orders of magnitude more changes than there are CVS or SVN commits.

Table **??** shows the summary statistics of the number of Syde changes and CVS/SVN commits per day.

Table 2 Summary statistics for Syde changes and CVS/SVN commits per day.

Project		Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum
Speed	CVS	2	4.00	6.00	6.00	8.00	10
	Syde	1	48.75	143.50	242.90	367.50	668
jArk	SVN	1	3.00	6.00	7.82	12.75	24
	Syde	1	121.00	454.00	792.90	863.80	9,242
Pacman	SVN	1	1.00	4.00	5.14	8.00	17
	Syde	5	113.00	300.09	437.90	611.00	2,981

To compute the summary, we removed the days with no changes and no commits (*i.e.*, if there was at least one commit or one change, the day is included for both measurements). For at least 75% of the days, the number of Syde changes is two orders of magnitude higher than SVN commits for jArk and Pacman (2nd, 3rd, and 4th quartiles). For Speed, for at least 50% of the days the same assumption holds (3rd and 4th quartiles).

Figure **??** shows an overall view of the Syde ownership maps of the three projects. Since the full maps are large, we use magnified, readable parts of them for the analyses that follow.

6.2 Characterizing Code Ownerships with Syde

Table **??** shows the comparison between the total number of Java files contained in Syde's log and the number of these files with differences in ownership classification (deltas) between Syde and CVS/SVN. According to Table **??**, the number of Syde changes is two orders of magnitude higher than the number of commits. Therefore, we expect Syde's ownership measurement to give finer-grained information about who is the current expert on a file, than the measurement based on CVS/SVN. Speed has a low number of files committed – only 14 files –, which influences the low value of files with deltas. Pacman and jArk, however, have a high number of files with differences in classification. The significant number of commits and the longer period of Syde usage justify the high number of files with deltas.

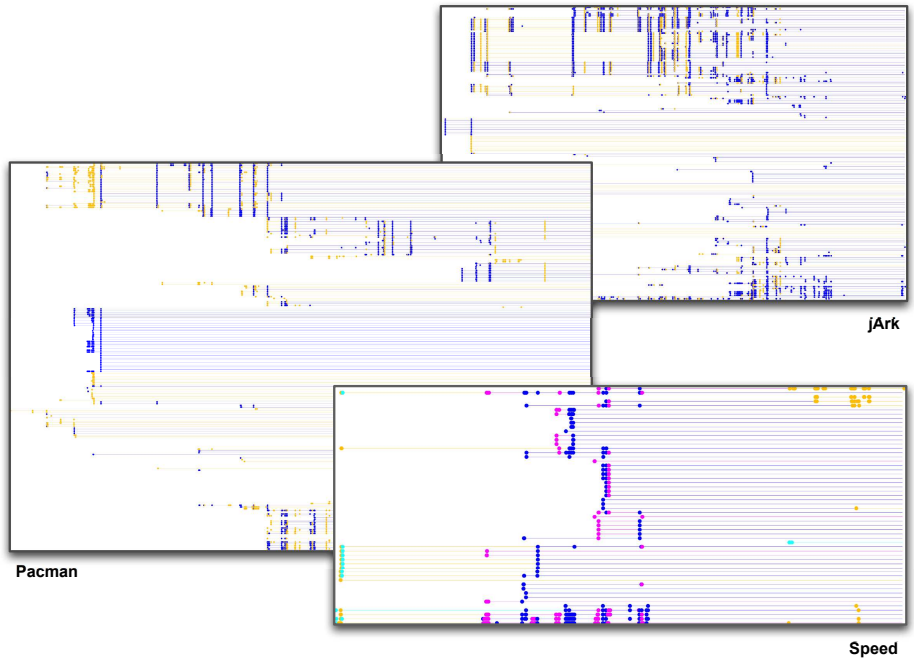

Fig. 9 Overview of Syde ownership maps of projects Speed, jArk, and Pacman. This picture aims at providing an overview of the ownership maps of the three projects we studied. It is not meant to be inspected in details by the reader.

Table 3 Comparison between total number of files and number of files with differences between Syde and CVS/SVN ownership classifications.

	Speed	jArk	Pacman
Total files	56	172	223
Files with deltas	8	130	122

Speed. Figure **??** shows Syde, CVS, and delta ownership maps for a set of files of the Speed project. The Syde map suggests that developers dark blue (Bob) and pink (John) are the owners of the majority of the files, whereas the CVS map shows that John has not committed any file. Figure **??** reinforces this observation. While John is responsible for 40% of Syde changes, he did not commit any change to CVS. The contrary happens with Alice, who is responsible for less than 20% of Syde changes, but appears as the major contributor according to CVS.

We investigated why John did not commit any change to CVS, and this behavior was influenced by two factors. The first one was the late adoption of CVS, which was done four days after the project had started –towards the end of the first week. The second factor was that John was actively involved in three other projects within the company, and had higher priority on one of the other projects in the second week of the experiment. Therefore, he was only able to commit his changes after the period of the study.

It is evident from the differences between the Syde and CVS maps that the Speed developers do not commit their code frequently, nor do they present a common behavior. Figure **??**

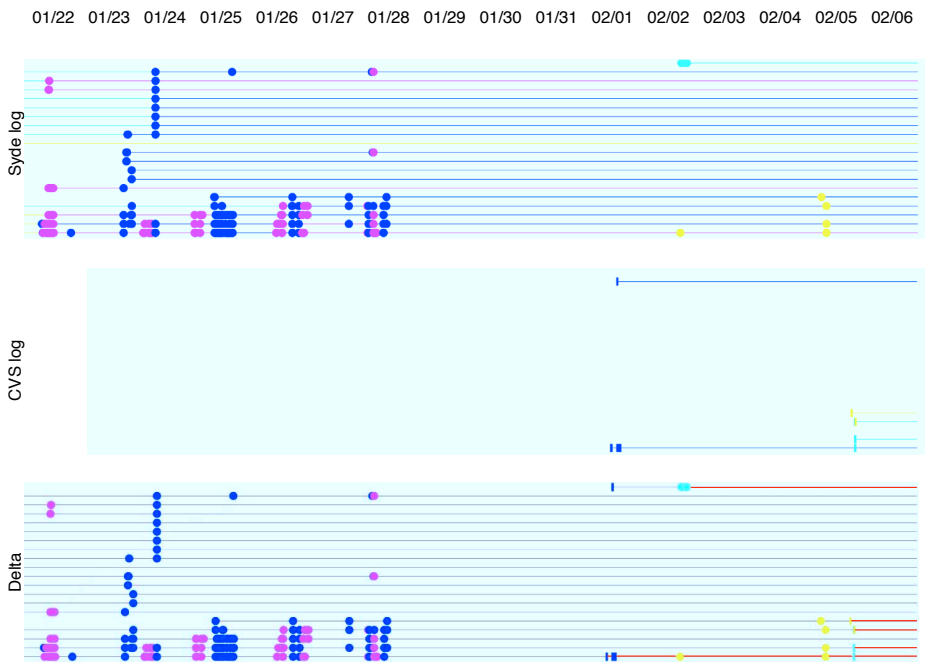

Fig. 10 Characterization of ownership of a set of files of the Speed project.

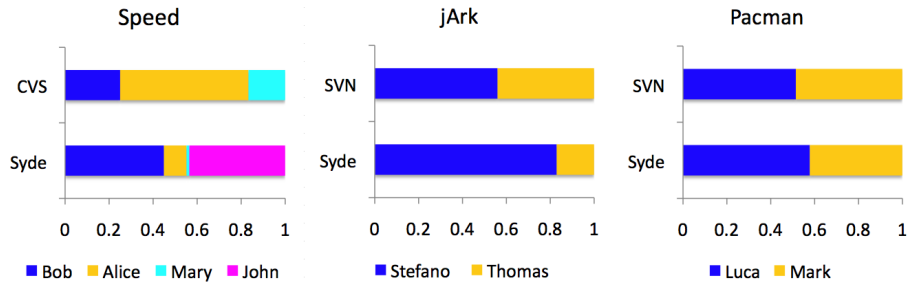

Fig. 11 Distribution of changes per developer for Syde, and commits per developer for CVS/SVN for the three projects. We include all files that had at least one Syde change, or at least one commit for CVS/SVN.

leads us to the same conclusion: In the context of Speed, the definition of code ownership according to Syde is more suitable than the one according to CVS. Based on this, we suggest that the larger the difference between the effort of a developer (measured as number of small changes) and the frequency of his commits, the more suitable our approach is in relation to the one of Gîrba *et al.* However, since the developers of this project reported that they were developing multiple projects at the same time, and that some of them occasionally forgot to use Syde, further investigation was needed to support our suggestion. We hence performed the same study on the two other projects at our disposal.

Pacman and jArk. Looking at the distribution of changes/commits per developers of Pacman and jArk, we observe that there is consistency between Syde changes and SVN commits. The authors who performed the most changes are also the ones who committed more, even though the percentage of Syde changes are higher than the percentage of commits (*e.g.*, in jArk, Stefano performed about 80% of Syde changes, and about 55% of the commits). We know that the students were working together most of the time, or remotely, but frequently communicating through instant messaging. Contrary to what happened with Speed, they were focusing on only one project, and equally splitting the workload. We believe that these characteristics influenced the consistent relation between Syde changes and SVN commits.

According to the characteristics of distribution of changes/commits per developers of Pacman and jArk, the differences in ownership classification based on Syde and SVN are influenced by the frequency with which the developers commit. In other words, we expect to see differences in classification during the period between a set of Syde changes and a commit –when the takeover happens. To investigate where deltas appear, we take a close look at the maps of one day of work in Pacman.

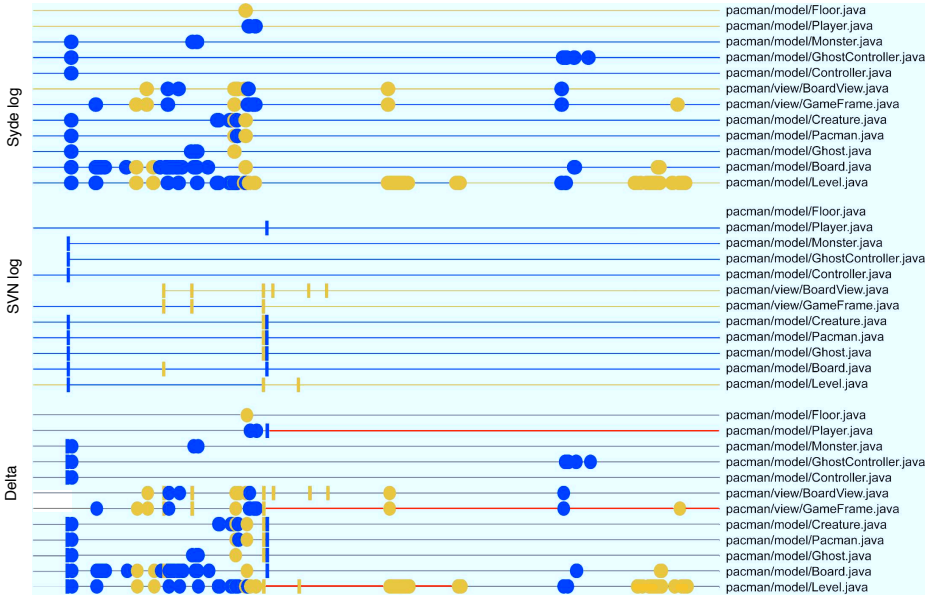

Fig. 12 Characterization of ownership of a set of files of the Pacman project.

Figure **??** shows the ownership maps of one day of work for Pacman. By carefully analyzing the three maps, we notice that in most of the cases a developer works on a couple of files for some time, and commits his changes later. This is the case of files `BoardView` and `GameFrame`; `Creature`, `Pacman`, and `Ghost`; and `Board` and `Level`.

The Syde and SVN maps show that both developers are often working on the same files in parallel (*e.g.*, `BoardView`, `GameFrame`, `Creature`, `Pacman`, `Gost`, `Board`, and `Level`). In such cases, when differences between Syde and SVN ownership appear, they are related to both the frequency of commits and the nature of the measurements. While the measurement based on Syde logs relies on the actual amount of work and time that one

spent, the measurement based on CVS/SVN commits relies on the number of lines changed, which does not reflect directly one's effort.

For example, imagine that a developer created and implemented a method, tested it, fixed some tricky defects that took him a considerable time, and finally refactored it. Syde records every edit step, reflecting how much effort this developer put on implementing this method, while CVS/SVN only informs the number of lines added on the file corresponding to this method. While the later can be a good effort indicator, it is common sense that the more effort someone puts on a task, the more likely it is that he will remember it.

Based on this, we reaffirm that the larger the difference between the effort of a developer (measured as number of small changes) and the frequency of the commits, the more suitable our approach is in relation to the one of Gîrba *et al.*

6.3 Evaluating Syde Ownership with Forgetting

In Section **??**, we introduced the concept of forgetting, and incorporated it in Syde's measurement of ownership. In this section we evaluate the forgetting effect by comparing the results of the measurement with different values of strength of memory. The function we adopted to describe forgetting is $R = e^{\frac{-T}{s}}$, where R is the memory retention, T is time, and s is the strength of memory. Ideally, the value of s is empirically determined by comparing it to the opinions of the developers themselves. However, we did not have that information at our disposal. We hence rely on heuristics and compared the behavior of ownership for several values of s. The values of s that we selected are 5, 25, and 125; we chose powers of 5 since the forgetting function is exponential, and with these 3 values we cover a reasonable variability of memory retention. According to Table **??**, a value of $s = 5$ reaches a low memory retention after 10 days (0.14), while $s = 25$ reaches the same value after nearly two months; a value of $s = 125$ yields a memory retention that is still strong after that time.

Table 4 Percentage of memory retention after a number of days for the values of strength of memory chosen for this study: 5, 25, 125.

	days						
Memory strength	**1**	**5**	**10**	**20**	**30**	**40**	**50**
s = 5	0.82	0.37	0.14	0.02	0.00	0.00	0.00
s = 25	0.96	0.82	0.67	0.45	0.30	0.20	0.14
s = 125	0.99	0.96	0.92	0.85	0.79	0.73	0.67

To determine the best memory setting for each project, we use two heuristics: minimizing the ratio of *short-term* switches among all the switches, and minimizing the overall average number of switches per files. A switch happens when a developer takes over the ownership of a file, but only when the file had a previous owner. A short-term switch is a switch whose overall duration is four hours or less. Finding the value of s that minimizes these two values leads to a better ownership description, as it minimizes what can be seen as spurious changes of ownership. Table **??** reports both these values for each project and each memory strength, including the default ownership measurement having a full memory.

To have a finer-grained view of the ownership switches, we also show the distribution of short and long-term ownership switches among files and memory settings in Figure **??**. This figure shows a series of histograms of ownership switches per file. The x axis represents the

Table 5 Percentage of short-term ownership switches among ownership switches, and average number of switches per file, per project and memory strength.

	short switches			switches per file		
Memory strength	**Speed**	**jArk**	**Pacman**	**Speed**	**jArk**	**Pacman**
s = 5	10.0%	29.6%	44.4%	0.70	0.72	0.57
s = 25	32.2%	31.1%	40.5%	1.05	1.08	0.83
s = 125	32.1%	32.5%	40.7%	1.00	0.93	0.63
full memory	31.6%	33.3%	41.7%	1.02	0.81	0.60

number of switches, and the y axis shows the number of files which had that exact number of switches throughout their lifetime. The first three columns show the histogram considering the forgetting effect in order of increasing memory strength, whereas the last column does not consider forgetting. Black bars represent short-term ownership switches, while gray bars represent all ownership switches.

For all three projects, the majority of the files did not have switches: 28 out of 56 files for Speed; 120 out of 172 files for jArk; and 159 out of 223 files for Pacman. We removed them from the histogram in order to increase the resolution of the y axis and hence the legibility.

Relationship between number of switches and memory. If we analyze the files by frequency of switches, the histograms indicate that, overall, a weak memory yields the lowest number of switches. In other words, the number of files with less switches is larger for smaller values of s, while the number of files with more switches is larger for larger values of s. Table **??** confirms these observations: The lowest ratio of overall switches per file is consistently obtained for $s = 5$, although larger values give a close ratio for the Pacman project. The maximum is found for $s = 25$, and it decreases lightly after that.

Speed has a very low number of files with 2 switches or more for $s = 5$. A stronger s noticeably increases the number of files with 2 switches. Pacman and jArk contain a greater number of files with ownership switches, but have an overall smaller ratio of conflicts switches per file. Speed is an overall smaller project (at least the package that we focused on), with twice the number of developers; hence the potential for conflict increases. On the other hand, jArk and Pacman both have several files with a high number of switches, indicating that a small number of files receive the majority of the conflicts.

Relationship between memory and short-term switches. Consulting Table **??**, we observe that for Speed and jArk, the proportion of short-term switches among overall switches increases as the strength of the memory increases. For Pacman, this trend is true for $s = 25$, $s = 125$, and no forgetting, but the value for $s = 5$ is surprisingly the highest. $s = 5$ gives the overall lowest number of switches, but has a comparable number of short-term switches with $s = 125$ or the no forgetting ownership measurement; this gives the impression that ownership switches are increasing. With a closer look at the history, we found that all the 56 short-term ownership switches happened in the first half of the project, on a restricted set of 33 files. The developers of Pacman worked together at the beginning on the model of Pacman, and then split: One stayed on the model, while another developed the view. This close collaboration early on caused a large number of unavoidable short-term switches: On the 24/04/2009, 23 short-term switches occurred on 10 files containing unit tests; 13 short-term switches happened on 7 files on 20/04/2009; and 10 other occurred on 10 files on 06/05/09. This shows that the short-term switches are clustered around specific dates and denote bursts of activity. As such, they are not artifacts of the usage (or non-usage) of the forgetting effect.

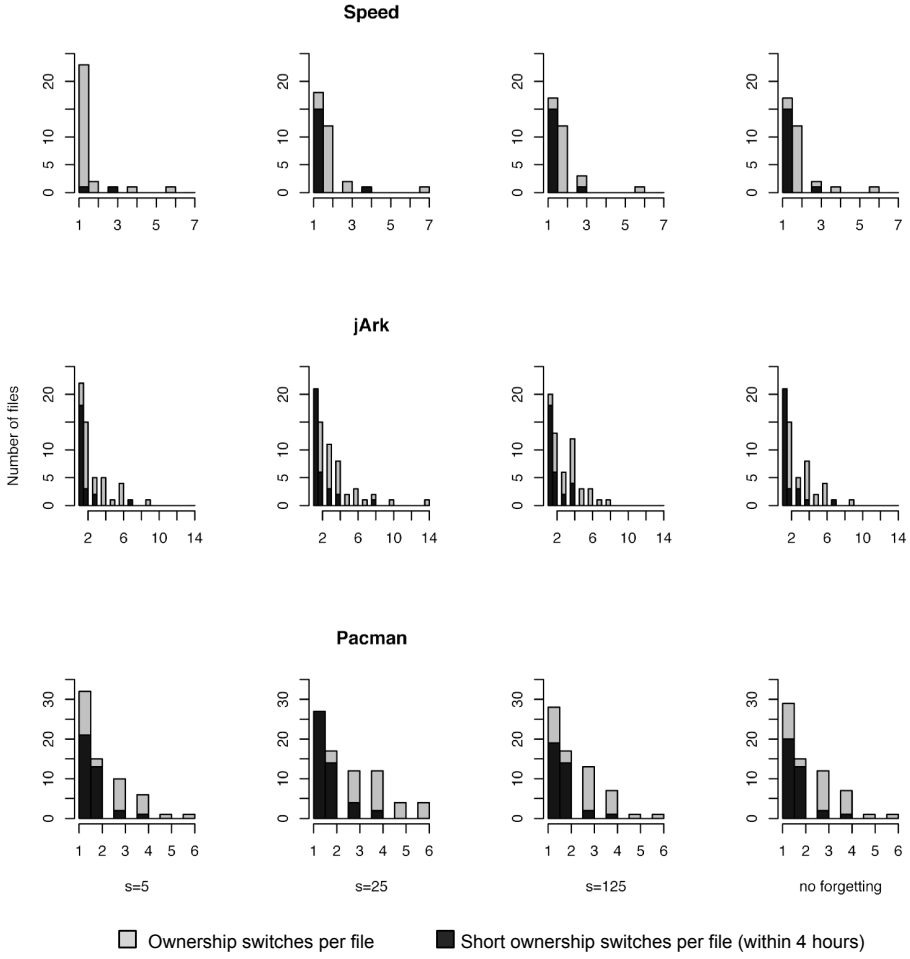

Fig. 13 Histogram of ownership switches per file according to Syde changes. The first three columns consider forgetting with $s = \{5, 25, 125\}$.

We can conclude that the best setting is again the lower memory value ($s = 5$) as it diminishes the overall quantity of switches, even if it is not possible to reduce short-term ones, which hence increase in proportion.

A detailed view on ownership switches. With these observations in mind, we focus on investigating the behavior of the switches for the different rates of forgetting. To do so, we take a close look into a set of classes from jArk. Figure **??** shows a Syde ownership map where, for each class, there is one line per each s value, and the last line for the original measurement (with no forgetting). The main pattern that we observe is that the stronger the memory is, the longer it takes for an ownership switch to happen. Furthermore, when $s = 125$, the behavior is extremely similar to when developers have a perfect memory. In Figure **??**, they are in fact equal (not the case in jArk overall). This might be an indication that $s = 125$ is too high, at least for the context of jArk. Indeed, Figure **??** shows us that developers would have 70%

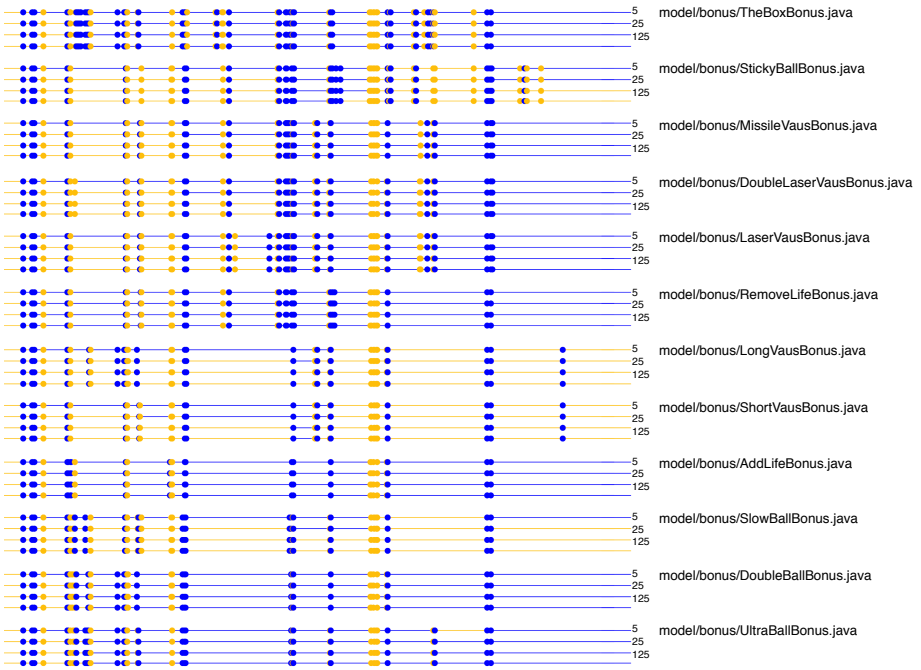

Fig. 14 Ownership map of jArk with forgetting notion.

probability to remember a file 50 days after they changed it. In the context of jArk, which lasted for approximately 35 days, this value is already too high.

Conclusions. As previously stated, the strength of memory is a subjective value that depends on the various characteristics of each project. Therefore there is an optimal range of values for each project, but there is no optimal value for all projects. In the case of jArk a short strength of memory is more suitable, since it is a project with a short duration, and two developers with no clear division of tasks, thus with dynamic characteristics. In the case of Speed, even if there is a clear division of tasks, there are more developers involved, which raises the number of conflicts; it is also more suitable to consider a lower strength of memory, at least until the project grows in size. In the case of Pacman, a low value of memory reduces the overall number of conflicts, but keeps the number of short-term ownership switches nearly constant, and is hence preferable as well. Even if the three projects tend to benefit from the same memory settings, it is too early to generalize beyond them.

7 Threats to Validity

7.1 Threats to Construct Validity

Construct validity refers to the extent with which our variables are correctly measured. We identified two potential threats to construct validity:

Syde Change Recording. Although Syde checks for compilation errors when a source code is changed, we do not compute the structural differences from two subsequent versions. As consequence, any edit to a file is considered as a change, including addition or deletion of comments and blank lines.

Syde records every change made by a developer as long as he is connected to the Syde server. Speed was monitored with an early version of Syde with limitations. The history log collected from Speed is not complete, because some of the developers reported that they forgot to connect to Syde a couple of times. We minimized this issue by offering the option to automatically connect to the server, however this initial version of Syde did not enable automatic connection by default. This early version was also missing a buffer in the plug-in to save the changes performed while the developer is offline and to send them to the server when he connects. Thus a number of changes may have been lost, which may have influenced the accuracy of our measure.

The jArk and Pacman projects were monitored with a later version of Syde, featuring both auto-connect enabled by default and a buffer to record offline changes. This allowed us to record a large portion of the changes that would have been lost with the previous one. There is however a slight probability that the offline buffer was full, leading to the loss of a few changes.

SCM Usage. Syde was used since the beginning of the implementation phase of Speed, but CVS was only adopted four days later (01/26). This fact could have influenced ownership in the beginning of the project. This was a decision taken by the team and hence beyond our control. Pacman and jArk adopted Subversion at the beginning of the project, so this threat does not apply for them.

7.2 Threats to Internal Validity

Internal Validity refers to the validity of our causal conclusions. We identified two potential threats:

Developer behavior under observation. Since developers knew they were observed, they may have altered their behavior in ways that we cannot predict. For instance, they may have committed more – or less – often than usual. Since we monitored our subjects for relatively long periods of time using non-intrusive tools (SCM change logs and Syde), we think they had time to get used to it; hence that effect should be weak.

Match of ownership measurements with developers' opinion. Ideally, we should have collected the developers' opinion on how much they know about a particular file they edited. This collection should have been done at fixed intervals throughout the data collection process, so we would have points in time to compare our findings with developers' opinion. However, we did not collect this information, and showing the map to developers several months after the data collection requires a perfect memory from them to be able to check whether the map conforms to their notion of ownership.

7.3 Threats to External Validity

External Validity refers to how much our results can be generalized to other circumstances. We identified two potential threats to external validity:

Number of systems. We monitored three projects – one industrial and two student projects –, featuring a total of 8 developers over a combined time of 28 weeks. This is still a relatively low number of projects and a short period of time (although two of the projects were monitored from start to completion). Moreover, all the projects were implemented using the same toolset: the Java programming language and the Eclipse IDE. These restrictions prevent us from deriving stronger conclusions at this time.

Styles of Developers. Another aspect to be considered is that developers might present diverse patterns on saving and compiling, which could influence the results of code ownership measurement, since it is based on the number of changes each developer produced. We believe the usage of an IDE such as Eclipse, which outlines the errors still present in the code, encourages one to compile more often, thus mitigating this threat. In the same fashion, developers have different patterns of SCM usage. These can be influenced by the development process adopted by the team: Agile development encourages developers to check in their code frequently, while more traditional processes encourage developers to maintain the repository consistent, which may delay their check-ins.

Since we monitor eight developers, we do not know if we account for all the variability, even though we did notice large differences in SCM usage behavior with respect to the actual number of changes performed, which comforted our opinion that our ownership metric is more resilient than the one based on SCM system usage.

8 Conclusion

In this article we have used the logs of a novel type of software repository to determine code ownership and to compare the result with the ownership computed exclusively with SCM-level logs. The new repository stores every change performed by every developer in a multi-developer project. The repository is managed by Syde, a client-server application built with the goal of augmenting workspace awareness on a multi-developer environment. The foundation of Syde is Spyware's change-centric approach [**?**], in which each individual code edit is saved and can be recovered in the future.

Similar to mainstream SCM systems, such as CVS, Syde produces history logs containing useful information about changes, which can be mined in the same context as the widely mined CVS logs. The fundamental difference is that Syde's logs are the result of continuous edits performed by developers, who do not need to stop their work to submit the changes. In contrast, CVS logs are the result of explicit check-ins of changes, which can vary according to team culture, developer habits, and the likelihood of merge conflicts. Hence, we argue that Syde's logs reflect what happened in the past more accurately than the ones provided by mainstream SCM systems.

We mined Syde's log to determine code ownership and compared the result with the one produced exclusively with CVS or SVN logs. We defined a new ownership measurement based on the frequency with which developers change the code of each file; we subsequently refined the new measurement to add the notion of memory loss on the definition of code ownership. That is, a developer who has performed the majority of code edits of a file, but has not touched it for a long period (when the file underwent significant changes), starts to lose knowledge of it. In the meantime, the developer who performs the recent changes becomes more knowledgeable, even though he may not have performed as many edits as the first one.

To validate the Syde ownership measurement, we used the data collected by Syde, and the CVS/SVN logs from the development of three distinct projects: Speed, for a period of 15 days; jArk and Pacman for a period of 5 weeks. We monitored eight developers for a total of 28 man/weeks, or 7 man/months.

We compared the results of the variants of ownership with the help of the Ownership Map, a visualization introduced by Gîrba *et al.* [**?**], that we extended to fit our data. The results showed differences between the two classifications, especially when active developers did not check in their changes frequently –in one case, a developer did not commit any code for two weeks, significantly skewing the measurement based on SCM data. Based on this finding, we suggest that our code ownership classification is more accurate than the one proposed by Gîrba *et al.* [**?**], as it is less sensitive to the commit habits of developers.

In addition, we suggest that the use of the notion of memory loss when measuring ownership reflects a more realistic scenario than assuming a developer remembers everything regardless of the time passed. We found that models based on smaller memory retention in general satisfied the two heuristics of minimizing the number of ownership switches and of minimizing the number of short-term (possibly spurious) ownership switches. However, it is important to emphasize the subjective nature of forgetting, and thus, that the ideal rate of forgetting for each project is subject to its characteristics.

The ownership maps are a means to investigate the variation of ownership at a fine-grained level rather than a visualization to help developers to detect file owners. The visualization has a number of scalability constraints, such as the number of developers that can be distinguished by different colors, and the increasing difficulty for the human eye to spot an ownership switch as the map is shrunk to show a longer time span within a fixed size.

Therefore, as future work on code ownership, we plan to implement a recommender in the form of an Eclipse plug-in to help developers to locate those who are knowledgeable about an artifact of the system. The recommender should allow a developer to query for experts of a file or a package, and provide a rank of experts. We intend to use the ownership measurements investigated in this work to compute the knowledge that each developer who changed an artifact has at the moment another developer seeks for help. This recommender will be integrated with the existing set of Syde plug-ins.

We intend to investigate other subjective aspects that influence the knowledge of an individual compared to a target group (in our case, a developer compared to a team). Analogous to the notion of memory loss, there is the learning notion, *i.e.*, how does one acquire knowledge of a part of the system. However, we believe that the learning curve is more influenced by individual experience –in general and in the context of a project– than the forgetting curve. That is, an expert is more likely to understand what a feature does than a newcomer. We intend to study this notion and model it in order to refine the ownership measurement.

We believe that the data made available by Syde opens new perspectives for several analyses, such as the understanding of developers' roles and activities, code ownership, detection of unstable code, *etc*. We also believe that since the data is being collected in real time, we can provide new types of "developer assistance" [**?**], especially with respect to the collaborative aspects that Syde supports.

Acknowledgements We would like to thank CPMBraxis and its professionals for using Syde and providing useful feedback to us. We also thank the students that gently let us spy on them.

References