Mining the History of Synchronous Changes to Refine Code Ownership

Lile Hattori and Michele Lanza
REVEAL@ Faculty of Informatics - University of Lugano, Switzerland

Abstract

When software repositories are mined, 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 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 Syde tool, which records every change by every developer *in multi-developer projects. This new source of information can augment the accuracy of analyses and breaks new ground in terms of how such information can assist developers. In this paper we illustrate how the information we mine can help to provide a refined notion of code ownership. As a case study, we analyze the developers' activities of the development of a commercial system.*

1. Introduction

To manage the life cycle of software systems, developers use a number of diverse tools, such as software configuration management (SCM), bug trackers, discussion boards, etc. These tools store a large amount of information that is exploited by researchers to understand different aspects of the evolution of projects. Especially SCM repositories 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 that have mined SCM repositories to reveal the nature of software changes [1]–[3] and to understand the correlation between changes and developers' roles [4], [5] is based on largely adopted SCM systems, such as CVS and Subversion. However, any inference derived from such systems is subjected 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 [6]. Contradicting this statement, largely adopted SCM systems are still file-based and do not model the particularities of a programming language. Combined with the *check-out/checkin* protocol, 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 are impossible to be recovered even with elaborate mining and reverse engineering techniques [7].

Since checking code in 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 [8], and even check in partial changes [9] to avoid dealing with merge conflicts.

Modern Decentralized Software Control Management systems (DSCM), such as Git[1], offer additional support for parallel development. In Git the checkout/checkin model is replaced by the clone/pull/push model with which every developer can potentially maintain his own repository by creating a branch from someone else's repository. Some of the consequences are that Git's 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 branches created have to be occasionally merged, and conflicts resolved. Furthermore, DSCMs are also file-based.

Thus, given that the nature of information found in software repositories determines what we can infer from it [10], we believe that studies purely derived from file-based SCM systems are threatened by the loss of information that comes with the underlying model.

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 brings Spyware's [11] change-centric approach to augment the awareness of a team of developers by propagating changes *as they happen* and by warning about potential conflicting

1. See http://git-scm.com/

changes [12]. 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 they happen. We define every change as every saved file that has undergone at least one structural change from the last save action (See Section 3). Hence, the once approximate data about *who* changes *what* and *when* is now accurate and complete; this is what we call *synchronous changes*. 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 the fact that the data is being collected in real time can provide new types of "developer assistance" [13], especially with respect to the collaborative aspects that Syde supports.

For a sample case study, we used Syde to record a period of the development of a commercial system. In this paper we use Syde's change history together with the project's CVS history log to (1) analyze potential conflicts, and (2) understand developers' dynamics and ownership of the code.

Structure of the paper. In Section 2 we describe a number of approaches that bear some similarity with ours. In Section 3 we then detail our approach and its supporting implementation in the form of Syde. In Section 4 we then use Syde to analyze the history of a commercial system, before concluding in Section 5.

2. Related Work

Syde is essentially a chance-centric approach to promote team collaboration, and thus is related to (1) tools that support collaboration and (2) operation-based SCM solutions.

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 [14]. Tool support for collaboration ranges from full-fledged platforms, such as Jazz.net [2] and CollabVS [15], to specific workspace awareness solutions [16]–[18].

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 with support for intra and inter-team collaboration, automation, and traceability of code, tasks and issues. Microsoft's CollabVS extends the Visual Studio by adding communication channels, such as text and audio-video chat, browsing of remote unchecked versions of files, and notification of presence in elements inside a file [15].

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

Palantir is an Eclipse plug-in that addresses direct and indirect merge conflicts [16]. 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. Palantir 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 with the aim of augmenting group awareness [17].

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 [18].

The demand for workspace awareness is becoming urgent as intensive and globally distributed team collaboration becomes the state of pratice. 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. The 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.

On the other end of the spectrum there are language-dependent operation-based SCM systems [11], [19], 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 played or rewound to bring the system from one state to another, and merge conflicts can be resolved with operation-based merge algorithms [20]. 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.* [19], is not a pure operation-based SCM. It is a mixture of state-based and operation-based, which means that it does not record every change made by every developer. Instead,

2. See http://jazz.net

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 contrast, Robbes' Spyware's change-centric solution records every change made by one developer and is able to recover any state of the system [11]. The main restriction of Spyware is that it is a one-developer solution, *i.e.,* it does not support a multi-developer context. OperationRecorder [21] tries to bring Spyware's approach from Smalltalk to Java, however, it presents the same drawback as MolhadoRef: it also loses change information.

Our goal with Syde is to port Spyware's approach to a multi-developer context without losing information. Our goal is thus not to replace, but to complement, file-based SCM systems.

3. Syde

Syde is a client-server application that can manage and store object-oriented software systems implemented in Java. The client is an Eclipse plug-in that listens to "build" operations, which are often linked to "save" operations in Eclipse. Every time a developer tries to compile a changed file, Syde'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, then, saves the received information and broadcasts the (successfully compiled) changes to all active client instances. Finally, each client instances of the plug-in shows the broadcasted changes on a view 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 shows the information of others' activities in a view 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 we aim to provide effective collaboration support with minimal, lightweight, and complementary changes to the *status quo* of the developers' settings.
- *Enhance awareness.* Similar to other solutions to augment workspace awareness, Syde informs developers about what changes in the source code are happening and were not checked into the SCM repository yet. A developer can go ahead and request these changes even before they become available through the SCM.
- *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

The overall design and information flow of Syde are illustrated in Figure 1. Syde features the following components:

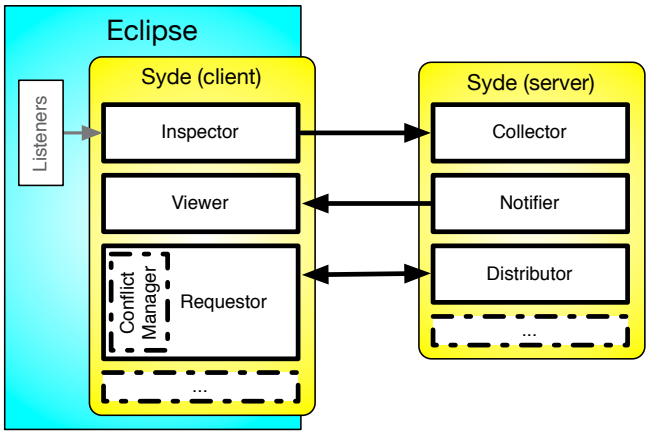

Figure 1. Syde Architecture.

- *The Inspector and the Collector.* Syde's inspector implements listeners to capture from Eclipse's workbench changes performed by a developer. The inspector collects two distinct types of data: the actual changes; and metadata, which 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 a view within Eclipse itself, thus providing awareness of changes to all developers.
- *The Distributor and the Requestor.* Once a developer has become 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.

3.2. Implementation

We implemented Syde in Java with Eclipse. Syde's goal is to complement the workspace awareness offered by SCM

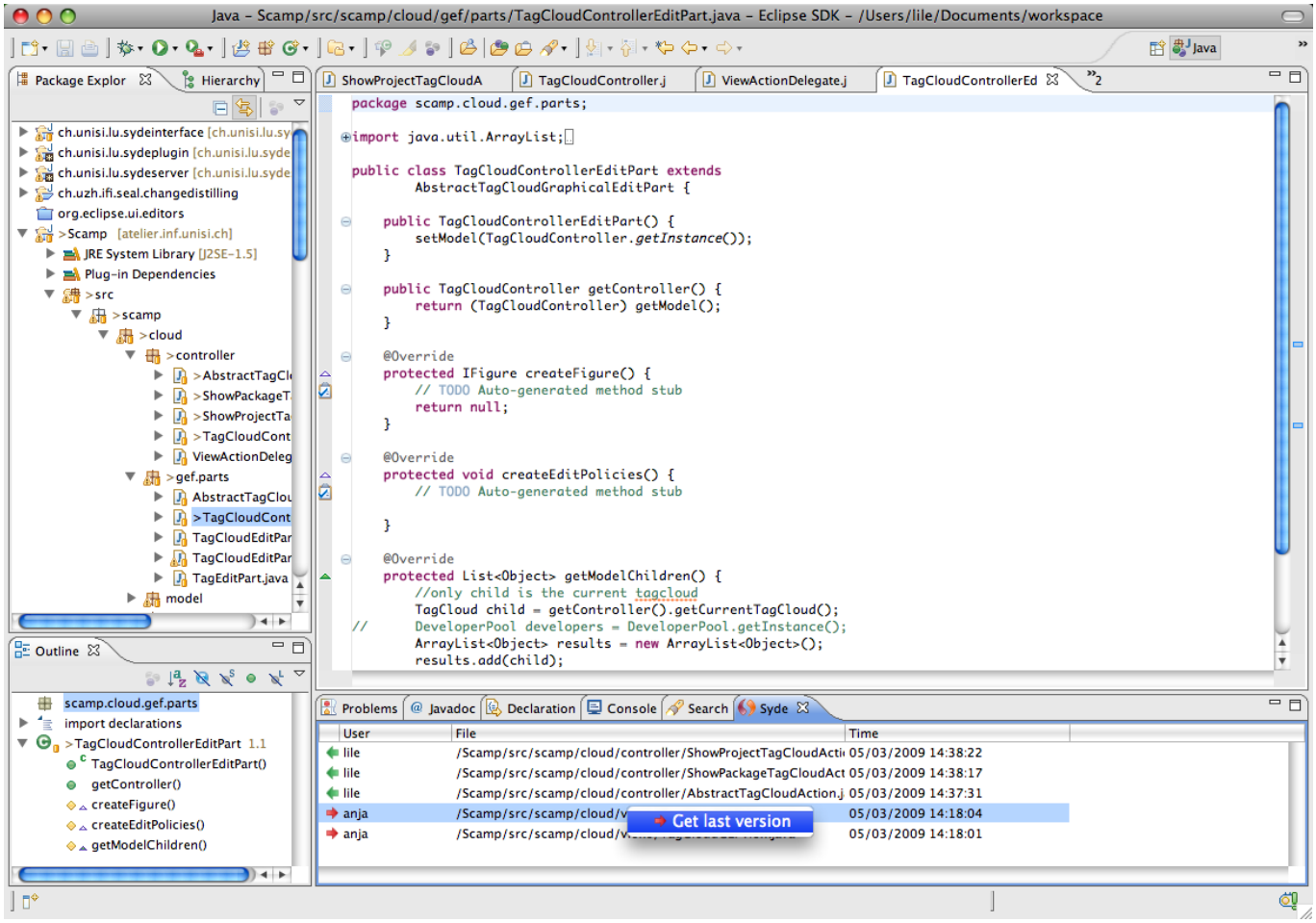

Figure 2. Screenshot of Syde.

systems. However, since its implementation is completely independent from any SCM system, it can also be used by projects that do not use any other version control system. To explain Syde's implementation, we follow the information flow illustrated in Figure 1.

The *Inspector* is located in Syde's plug-in and is responsible for inspecting 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.* [22], 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) preempts 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 in the plug-in, the *Viewer* makes a contribution to Eclipse's workbench by creating a new `ViewPart`, as shown in Figure 2. Finally, the *Requestor* adds the action "Get last version" to Syde's view, which requests from the *Distributor* the last version of the selected file in the view.

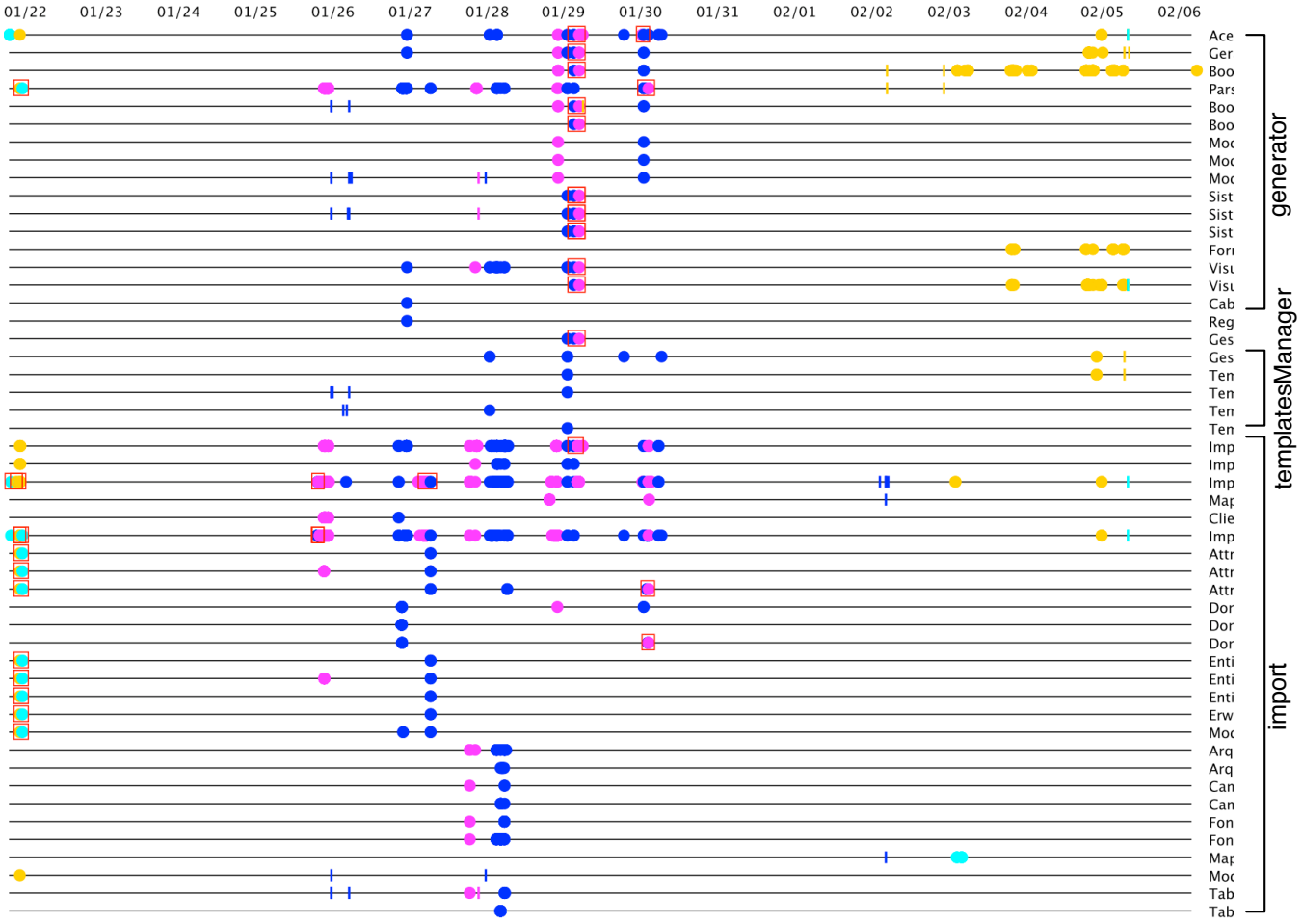

Figure 3. Timeline of changes of Speed project.

3.3. Data

The history log of mainstream SCM systems usually describes which files have been checked in, when and by whom. For example, CVS history log shows: file name, revision, author, timestamp, author's comment, and number of lines of code added and removed. Subversion gives the same information, except for the number of lines of code added and removed.

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

4. Case Study

In [12] we showed an initial analysis of the history of projects developed by a single developer. In the context of this paper 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 Syde's history log of Speed, together with two the corresponding CVS history log.

Speed is a commercial project that is under development at the software factory of CPMBraxis [3]. 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.

To collect the data from Speed, Syde was embedded into the production environment of the software factory, and the project's team was instructed, but not forced to use it for a period of 15 days. Speed has a total of 185 Java files, from which 94 had new versions checked into the CVS repository, and 50 had changes captured by Syde. During a period of 15 days, Syde collected a total of 2,429 changes performed by four developers. The number of new versions checked

3. See http://www.cpmbraxis.com

into the CVS repository that corresponds to these changes was 187.

Figure 3 shows the timeline of changes that Speed underwent during the 15-days period. Each line corresponds to a file that was versioned by Syde (The names of the files were obfuscated for privacy reasons). Each circle represents a change captured by Syde, and each vertical line a CVS check in. Each color represents a different developer. Finally, red rectangles indicate potential merge conflicts, which in the case of Figure 3 appear if two developers changed the same file within a period of 2 hours.

The four developers did not work in parallel during the whole period. Instead, they alternated, two by two, the periods of active development. This happened because the developers work in more than one project at a time, and the time that they dedicate to each project depends on deadlines. In addition, some developers reported that they forgot to use Syde for certain periods of time.

The project underwent a period of intensive development, in which two developers (pink and dark blue) worked in parallel on many different files. In most cases, none of them checked in the changed code after the intensive period of work. In some cases, the second file for example, there was even a third developer making changes one week later. This scenario seems susceptive to merge conflicts that will arise when developers try to check in the modified files.

In the following we analyze the research question previously presented.

4.1. Characterizing Code Ownerships with Syde

Code ownership 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 not all developers know each artifact of the system, and can be used to answer questions such as "who should fix this bug" [23] or "who should I ask about artifact XY" [24]. In this study, we consider the definition of code ownership described by Gîrba *et al.* [25]: "*Based on the number of lines of code added and removed extracted from the CVS log, the owner of a file is the one who owns the greater percentage of lines over the total number of lines of a file. In this case, the total number of lines of a file is approximated with information extracted from CVS.*"

This measurement technique can be effective when developers check in their files approximately with the same frequency. If however, 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.

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. We define code ownership as: *The owner of a file* f, own_f, *is the developer who has performed the greater number of small changes* c *on it. A developer becomes the owner of a file at the moment he performs* c+1 *changes in relation to the previous owner.*

Figure 4 shows code ownership schemas for Speed project. The top schema is based on our definition of ownership, thus it uses exclusively information contained in Syde's logs. In this schema, every colored circle is a change. The larger circles indicate when a new owner is assigned to a file, from where the line of the corresponding file is colored with the developer's color. Figure 4 *a* shows a magnified view of code ownership changes with Syde's logs. The overview suggests that developers dark blue and pink are the owners of the majority of the files. More specifically, by the last day (Feb 6) dark blue owns 34, pink owns 9, yellow owns 6, and cyan owns 1 out of 50 files.

The second schema is based on the definition by Gîrba *et al.*. Each tick represents one CVS commit, and the larger ones indicate when a new owner is assigned to a file, from where the line of the corresponding file is colored with the developer's color. Figure 4 *b* shows a magnified view of code ownership changes with CVS logs. This schema shows that the developers who own more files are dark blue (10) and yellow (7). Cyan developer owns only 2 files, while pink does not own any file. In addition, 31 files (a significant number) remain without classification because of the complete absence of a commit containing them.

Finally, the third schema shows the delta between the two techniques, illustrated by a red line where they diverge. In this schema, only the circles and ticks that represent ownership changes are shown. Figure 4 *c* illustrates how the differences are highlighted.

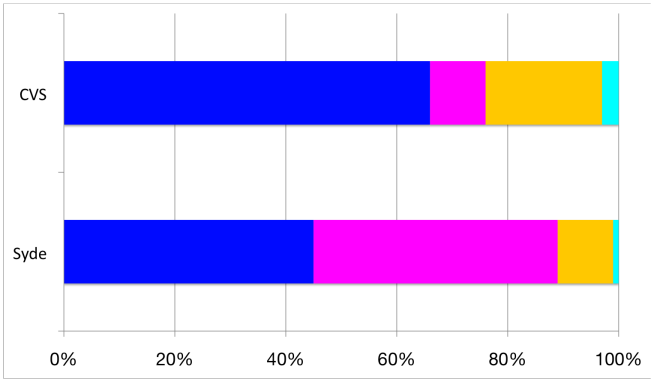

Figure 5. Distribution of changes per developer for Syde and CVS logs of Speed project.

It is evident from the difference between the first two illustrations that these developers do not check in their code frequently, nor do they present a common behavior. Figure 5 and Table 1 reinforce this observation. They show

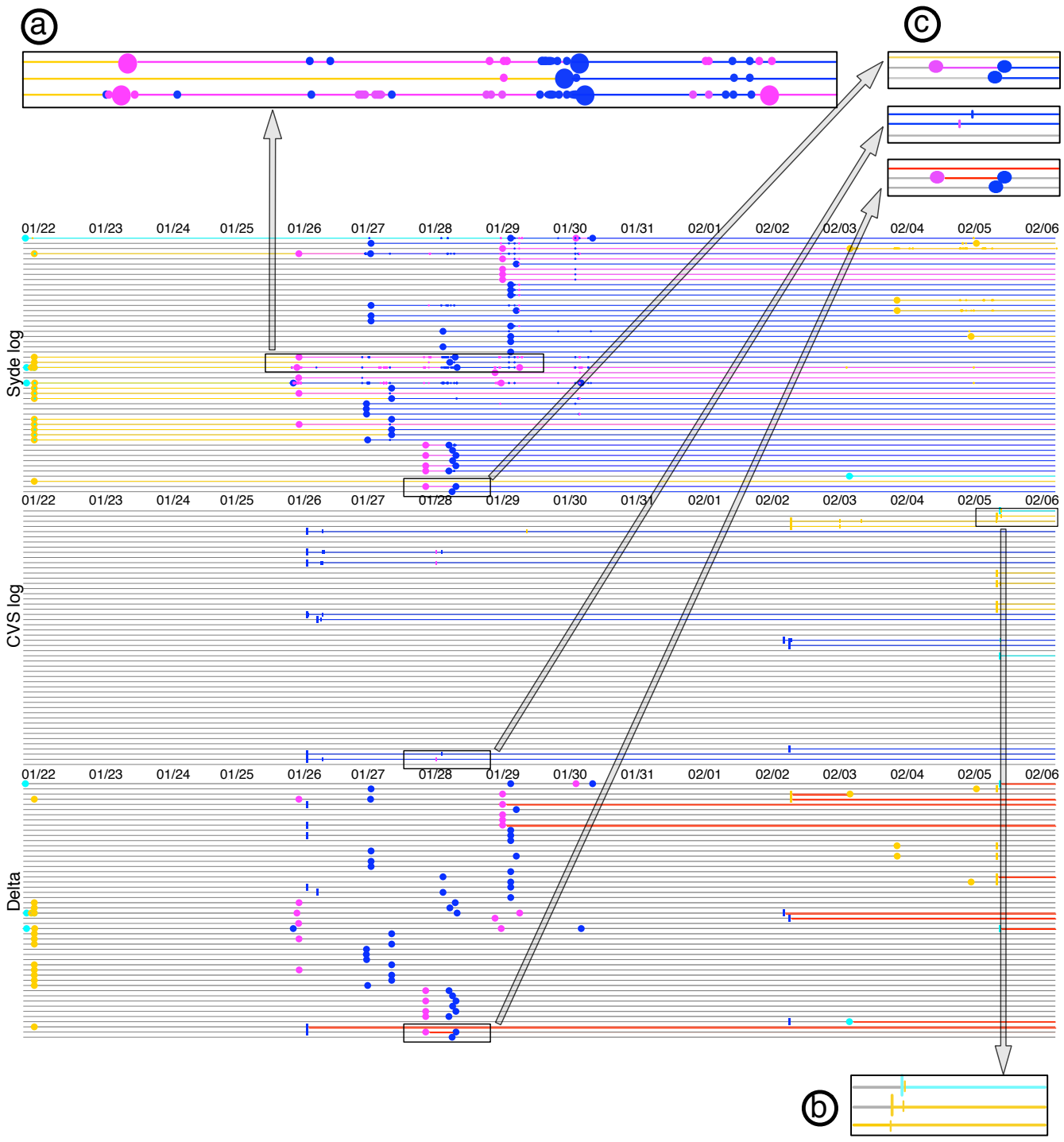

Figure 4. Ownership of Speed project measured with both definitions, and the delta between them.

	dark blue	pink	yellow	cyan
Syde (2,429 changes)	45%	44%	10%	1%
CVS (187 commits)	66%	10%	21%	3%

Table 1. Distribution of changes per developer for Syde and CVS logs of Speed project.

the distribution of changes per developer for both Syde and CVS. The absolute values of small changes and CVS commits are, respectively, 2,429 and 187. Developer dark blue appears as the most active, performing 45% of small changes and 66% of commits. However, developer pink, who appears as the second most active in relation to small changes (44%), has only made a few commits (10%) and does not even appear as owner in the second illustration. Consequently, he is the one responsible for the majority of diverging deltas (7 out of 12).

The illustration and the corresponding analysis evidence that our definition of code ownership is more suitable in the context of Speed. 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 this was an initial analysis of Syde's log generated in a multi-developer context, further investigation is needed to support our suggestion.

4.2. Threats to validity

Syde records every change made by a developer as long as he is connected to Syde server. 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 try to minimize this issue by offering the option to automatically connect, however we do not force developers to use it. In the future, we will add a buffer in the plug-in to save the changes performed while the developer is offline, and send to the server when he connects. In addition, Syde was used since the beginning of the implementation phase of Speed, but CVS was only adopted four days later (02/26). This fact could have influenced the ownership variation in the beginning of the project.

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.

Finally, the short period of usage of Syde and the fact that we only discuss one system, are evident restrictions that prevent us from deriving stronger conclusions at this time.

5. Conclusion

In this paper we have presented a novel type of software repository that stores every change performed by every developer in a multi-developer project. The new 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 [11], in which every small 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 to Syde. 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 applied Syde's log to determine code ownership and to compare the result with the one produced exclusively with CVS log. To achieve that, we used the data collected by Syde, and the CVS log from the development of a commercial system for a period of 15 days. During this period, four developers were active, but not working exclusively on this project. The results showed differences between the two classifications, especially when active developers did not check in their changes frequently. Based on this finding, we suggest that our code ownership classification is more accurate than the one proposed by Gîrba *et al.* [25].

As future work on code ownership, we intend 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 did not perform as many edits as the first one.

Syde's log opens new perspectives for investigation of other mining techniques that have exclusively used CVS or Subversion logs. Moreover, it provides fine-grained information that can be retrieved and analyzed "on the fly" to help developers while they are working.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990). We would like to thank CPMBraxis and its professionals for using Syde and providing useful feedback to us.

References

[1] Q. Tu and M. Godfrey, "The build-time software architecture view," in *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 398.

[2] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293.

[3] A. T. T. Ying, R. Ng, M. C. Chu-Carroll, and G. C. Murphy, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[4] O. Baysal and A. J. Malton, "Correlating social interactions to release history during software evolution," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 7.

[5] L. Yu and S. Ramaswamy, "Mining cvs repositories to understand open-source project developer roles," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 8.

[6] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "Impact of software engineering research on the practice of software configuration management," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 4, pp. 383–430, 2005.

[7] R. Robbes and M. Lanza, "Versioning systems for evolution research," in *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*. IEEE CS Press, 2005, pp. 155–164.

[8] R. E. Grinter, "Supporting articulation work using software configuration management systems," *Comput. Supported Coop. Work*, vol. 5, no. 4, pp. 447–465, 1996.

[9] C. R. B. de Souza, D. Redmiles, and P. Dourish, "Breaking the code, moving between private and public work in collaborative software development," in *GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*. New York, NY, USA: ACM, 2003, pp. 105–114.

[10] R. Robbes, "Mining a change-based software repository," in *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*. ACM Press, 2007, p. 15.

[11] R. Robbes and M. Lanza, "Spyware: A change-aware development toolset," in *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*. ACM Press, 2008, pp. 847–850.

[12] L. Hattori and M. Lanza, "An environment for synchronous software development," in *Proceedings of ICSE 2009 (31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track)*. IEEE CS Press, 2009, pp. xxx–xxx.

[13] A. Zeller, "The future of programming environments: Integration, synergy, and assistance," in *Proceedings of FOSE 2007 (2nd Conference on the Future of Software Engineering*. IEEE CS Press, 2007, pp. 316–325.

[14] A. Sarma, D. Redmiles, and A. van der Hoek, "Empirical evidence of the benefits of workspace awareness in software configuration management," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2008, pp. 113–123.

[15] R. Hegde and P. Dewan, "Connecting programming environments to support ad-hoc collaboration," in *Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE CS Press, 2008.

[16] A. Sarma, G. Bortis, and A. van der Hoek, "Towards supporting awareness of indirect conflicts across software configuration management workspaces," in *Proceedings of ASE 2007 (22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 94–103.

[17] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a software developers local interaction history," in *MSR '04 : Proceedings of the 1st International Workshop on Mining Software Repositories*, 2004, pp. 106–110.

[18] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson, "FASTDash: a visual dashboard for fostering awareness in software teams," in *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2007, pp. 1313–1322.

[19] D. Dig, K. Manzoor, R. Johnson, and T. N.Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436.

[20] E. Lippe and N. van Oosterom, "Operation-based merging," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 5, pp. 78–87, 1992.

[21] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," in *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*. New York, NY, USA: ACM, 2008, pp. 31–34.

[22] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[23] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 361–370.

[24] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of ICSE 2004 (26th ACM International Conference on Software Engineering)*. IEEE CS Press, 2004, pp. 563–572.

[25] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 113–122.