Syde: A Tool for Collaborative Software Development

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

ABSTRACT

Team collaboration is essential for the success of multi-developer projects. When team members are spread across different locations, individual awareness of the activity of others drops due to communication barriers.

We built Syde, a tool infrastructure to reestablish team awareness by sharing change and conflict information across developer's workspaces. Our main challenge is to balance the tradeoff between offering relevant information about the activity of the team and avoiding information overload. The novelty of our approach is that we model source code changes as first-class entities to record the detailed evolution of a multi-developer project. Hence, Syde delivers precise change information to interested developers.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated Environments; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Distributed/Internet based software engineering tools and techniques*; D.2.9 [**Software Engineering**]: Management—*Programming teams*.

Keywords

Collaboration, change, awareness, visualization, Syde.

1. INTRODUCTION

The development of software systems is a collective work that depends on the communication and coordination of teams of developers to deliver the final product. In collaborative software development, informal interactions play an important role, because they keep developers aware of what is happening in the project and ease team coordination – the management of dependencies between tasks [11]. However, when developers do not share the same office, and face-to-face communication is not a part of their daily activities, keeping developers informed of one another's activities becomes a challenge.

Awareness is frequently defined as an understanding of the activities of others to give a context for one's activities [5]. It is becoming an important topic in software engineering [4], especially in the context of global software engineering (GSE), where geographically distributed teams have special needs with respect to awareness [15]. The distance, diverse time zones, different cultures and customized software processes are some of the aspects that directly affect communication and awareness of distributed teams.

A significant effort has been made to address the special needs with respect to awareness by enhancing the coordination capabilities of SCM systems. A coordination drawback shared by current software configuration management (SCM) systems is their strategy of propagation of changes: Only when a developer checks in his changes, will his colleagues have access to them; and only when his colleagues synchronize their code with the repository, will they become aware of new changes.

A number of tools have addressed this problem by providing real-time information of ongoing changes, and alerting developers of emerging conflicts: ProjectWatcher [16], Lighthouse [3] and FASTDash [1] create different visualizations with information collected directly from developers' workspaces. CollabVS [9] takes a wider approach by providing different communication channels, such as instant messaging and videoconferencing. CollabVS and Palantír[14] analyze ongoing changes to alert developers of emerging conflicts.

The main challenge faced by these tools is how to balance the tradeoff between providing relevant information about the activities of the team members, and avoiding overloading developers with irrelevant information. The aforementioned tools have different ways to manage this tradeoff, where the great majority captures changes at the file level. This implies that change information broadcasted to the team is at the level of added and deleted lines. To provide more accurate information (*e.g.,* a couple of lines added and deleted are actually a change on a statement of a method) these tools have to use differencing algorithms to reconstruct and interpret the change to infer its type.

We tackle the challenge of providing awareness information by adopting a change-centric approach [13], which has been widely used to manage inconsistencies among different views of software artifacts [7]. Inconsistency management often focus on the model of the system (*e.g.,* UML model) and is usually associated to the activity of one person [2]. We bring the change-centric approach to a multi-developer context and focus on source code to record the detailed evolution of the system's implementation.

We model object-oriented systems as abstract syntax trees (AST), and changes as tree operations. When a developer changes something inside a class, a tree operation that represents the change is captured and sent to the server. The main difference between our approach and others is that we capture the change at the level of program entities on the developer's workspace, instead of using a differencing algorithm to interpret the change later on. This allows

ICSE '10, May 2-8 2010, Cape Town, South Africa

us to have more precise change information and to detect conflicts by comparing tree operations.

Our tool, *Syde*, provides information of who is changing which parts of the system in real time - synchronous development. It also detects merge conflicts as soon as they arise, and informs the involved developers of them. Syde is an extensible client-server application, where clients are Eclipse plug-ins that both capture changes and show change information as visual cues. Syde's current features allow us to:

- treat textual changes as first class change operations;
- collect change operations in a centralized server;
- broadcast change information to all team members of a project;
- store the changes in a repository for later use;
- access the change history of any package, class, method, or field;
- show the differences between two versions of the program and alert developers of conflicts;
- visualize in real time how the system is evolving through visualizations and visual cues;
- access all of its functionalities from the IDE, rather than a stand-alone tool, to ease its usage.

2. SYNCHRONOUS DEVELOPMENT IN A NUTSHELL

Syde provides to the developers of a team the notion of synchronous development, where everyone is aware of the activity of others in real time. In order to achieve synchronous development, Syde extends Spyware's change-based software evolution model (CBSE) [13]. Spyware treats change as first-class entities with the aim of accurately modeling how software evolves. Syde extends its model from a single-developer to a multi-developer approach.

The Model. We model the evolution of a software system as a set of sequences of changes, where each sequence is produced by one developer. A sequence of changes takes a developer's copy of the system from one state to the next by means of semantic operations. These operations are captured from the Eclipse's workbench every time a developer modifies his copy of the system. Syde captures changes is at every save action. Thus, the evolution of a system is the combination of the sequences of changes produced by each individual.

System Representation. We focus on object-oriented systems in Java, thus we store and analyze constructs such as classes and methods, instead of files and lines. Hence, a software system is modeled as an AST containing nodes, which represent packages, compilation units, classes, methods, and fields. Nodes have properties, which vary depending on their type. A class can have a superclass and a set of interfaces; a compilation unit can have a set of imports. Syde provides a unique identifier for each entity and tracks name changes and entity moves, which is not possible in current mainstream SCM systems. These systems track files based on their names, and when someone changes the name of a file, the SCM sees it as a deletion and an unconnected addition. Syde keeps on the server one AST per developer, which reflects exactly the state of the system at a developer's workspace.

Change Operations. In CBSE, change operations represent the evolution of the system, instead of file versions [13]. A change operation is the representation of a change a developer performs in his workspace, *i.e.,* it is the transition of a system from one state to the next. Syde captures two types of change operations, detailed in Table 1: atomic changes, and refactorings [6].

Table 1: Change Operations

Atomic Operations	
Insertion	Insert a node *n* as a child of parent *p*.
Deletion	Delete a node *n* from its parent *p*.
Property Change	Change the value *v* of property *r* of node *n*.
Property Insertion	Insert the value *v* of property *r* of node *n*.
Property Deletion	Delete the value *v* of property *r* of node *n*.

Refactorings	
Rename	Change the name of a node *n* and change all references of this node from the old name to the new one.
Move	Move a node *n* and all its decedents from old parent p_{old} to new parent p_{new}.

Atomic changes are the finer-grained operations on the system's AST. An atomic change contains all the necessary information to update the model. By applying a list of atomic changes in the order they were received on the server, it is possible to generate all the states of the program during its evolution. Although atomic change operations reflect the entire evolution of the system, they can lead to an overwhelming amount of information. In addition, some refactorings that a developer performs are automated by Eclipse and lose their meaning if seen as separated atomic changes. Currently, we capture two refactorings: rename, and move.

Conflict Detection. Syde detects structural conflicts [12] related to the atomic operations previously listed. To detect emerging conflicts, every time a new atomic operation is applied to the AST of a developer, it is compared to the ASTs of the others. The conflicts are classified into two categories: *yellow*, when there are structural differences between two versions of a node, but none of these versions were checked in the SCM system; *red*, when there are structural differences between two versions of a node, and one of them was checked in the SCM system. Syde keeps track of the version of each node, by inspecting the corresponding file's version on the adopted SCM system (e.g., Subversion). The state of a conflict is stored, and at every change on one of the entities involved in the conflict, it is updated. The involved developers are kept up-to-date with the conflict's state until it is solved, and the visual alert is removed from their workspaces.

3. SYDE'S PLUG-INS

Syde uses a client-server architecture, where the server application is located in a centralized server, and the clients are an extensible set of plug-ins. The server is responsible for collecting change information, holding the AST representations of the system, applying conflict detection algorithms, and storing and making available all the information about the evolution and coordination (*e.g.,* merge conflicts) of the system. The plug-ins provide a collection of visualizations that provide team awareness. These plug-ins, depicted in Figure 1, are:

The Inspector (1). It is the main plug-in, responsible for inspecting code changes, translating them into change operations and sending to the server. The Inspector also provides an API that facilitates the creation of plug-in extensions.

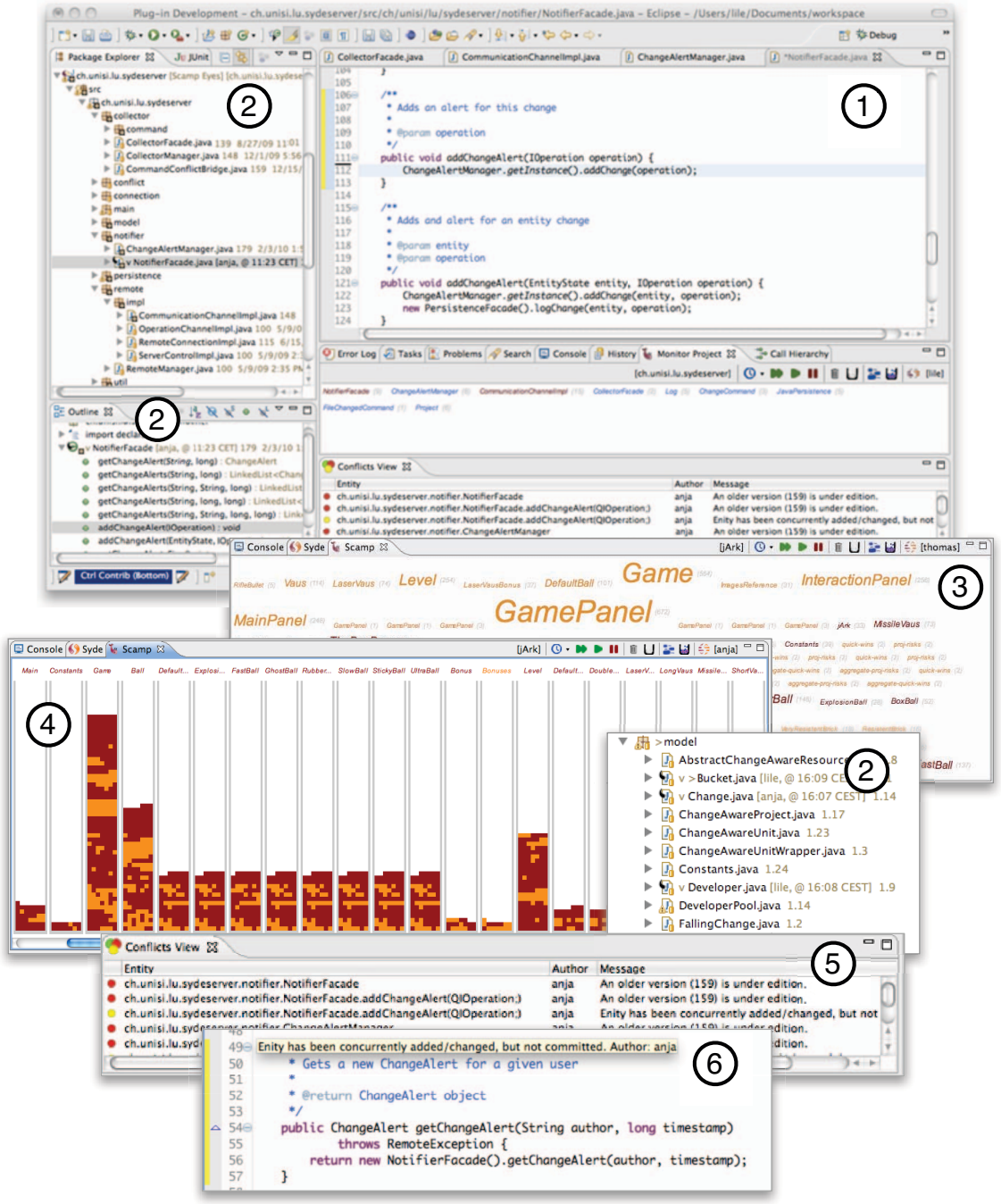

Figure 1: Syde screenshots. 1: The Inspector Plug-in. 2: Scamp Plugin - Decorations View. 3: Scamp Plugin - WordCloud View. 4: Scamp Plugin - Buckets View. 5: The Conflict Plug-in - Conflicts View. 6. The Conflict Plug-in - Annotation on Java Editor

Scamp (2, 3, 4). It provides lightweight extensions to Eclipse to enhance workspace awareness and assist developers to collaborate. Scamp has three different types of visualizations:

Decorations (2). Scamp provides a decoration in the form of small annotations within the Eclipse package explorer and outline view, where the files of the project are displayed. If a developer using Syde and Scamp is changing a class, its representation in the package explorer is annotated in three different ways to express that "something has changed" in that class. Scamp's decorations are: (1) an overlay icon – indicates the classes that have been changed; (2) an arrow – it points up if the class has been changed by the user himself, and points down if the last person who changed it is someone else; and (3) a textual annotation – if someone else is the last person who changed a class, an annotation is displayed after the class name, showing who made the change (username) and the timestamp.

WordCloud View (3). It displays the names of the classes present in a project. The number of changes that have been performed on each class is used as size metric, while the order indicates the recency of the changes, with most recently changed classes at the top. Each word in the cloud is colored according to the developer who made the most recent change to the class in question. Clicking on a word will take the user to the source code.

Buckets View (4). It displays the effort spent on each class that was recently changed. The classes are displayed as "buckets", which are progressively filled with single changes depicted as small squares. The color of each change denotes the developer responsible for it. Changes follow a chronological order, thus older changes are at the bottom of the bucket, while newer changes appear at the top. Each bucket has the corresponding class name colored according to the developer who owns the code. Ownership in this case is defined as the developer who has performed the greatest number of changes [8].

The Conflict Plug-in (5, 6). It displays information about emerging conflicts as developers' copies of the system become inconsistent with one another. Conflict alerts are shown on a view (5) and as annotations on the left side of the Java Editor (6). Red conflicts are considered severe, because they involve at least one version of an entity that is outdated according to the SCM system. Yellow conflicts are considered moderated, but they can easily become severe if the involved developers do not try to resolve them before checking in their code to the SCM. A developer can request to Syde the other version of the entity that has a conflict with his own, and the Conflict Plug-in will start a semi-automated process to resolve the conflict and merge the two versions.

4. TOOL INFORMATION

Implementation. Syde has been developed over the last two years as part of a Ph.D. thesis. It is free and open software written in Java. It is an evolving prototype that has been incrementally adopted and assessed through a number of case studies.

Experience. Syde has been used in a variety of contexts. The Inspector plug-in was used by an industrial team [1] and the history was used to analyze code ownership [8]. The Scamp plug-in was used by two teams of students to develop their course's project collaboratively, and we assessed the usefulness of the plug-in through a qualitative study [10]. We have been using Syde on itself throughout its entire development, which allow us to incrementally enhance its features.

Tool Availability. Syde can be obtained at `http://syde.inf.usi.ch` and through the Eclipse updater by entering `http://syde.inf.usi.ch/update`

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990).

5. REFERENCES

[1] J. T. Biehl, M. Czerwinski, G. Smith, and G. G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems)*, pages 1313–1322. ACM, 2007.

[1] A team of four developers at the software factory of CPMBraxis Inc.

[2] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *ICSE 2008: Proceedings of the 30th international conference on Software engineering*, pages 511–520. ACM, 2008.

[3] I. da Silva, P. Chen, C. V. der Westhuizen, R. Ripley, and A. van der Hoek. Lighthouse: Coordination through emerging design. In *Proceedings of ETX 2006 (OOPSLA Workshop on Eclipse Technology eXchange)*, pages 11–15. ACM Press, 2006.

[4] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*, pages 81–90. IEEE Computer Society.

[5] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of CSCW 1992 (ACM conference on Computer-supported Cooperative Work)*, pages 107–114. ACM Press, 1992.

[6] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[8] L. Hattori and M. Lanza. Mining the history of synchronous changes to refine code ownership. In *Proceedings of MSR 2009 (6th IEEE Working Conference on Mining Software Repositories)*, pages 141–150. IEEE CS Press, 2009.

[9] 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)*, pages 178–187. IEEE CS Press, 2008.

[10] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Proceedings of CSMR 2010 (14th IEEE European Conference on Software Maintenance and Reengineering)*. IEEE CS Press, 2010. Accepted.

[11] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[12] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[13] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, Jan. 2007.

[14] A. Sarma, D. Redmiles, and A. van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proceedings of FSE 2008 (16th ACM SIGSOFT International Symposium on Foundations of software engineering)*, pages 113–123. ACM Press, 2008.

[15] A. Sarma and A. van der Hoek. Towards awareness in the large. In *ICGSE 2006 (Proceedings of the IEEE international conference on Global Software Engineering)*, pages 127–131. IEEE Computer Society, 2006.

[16] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings of MSR 2004 (1st International Workshop on Mining Software Repositories)*, pages 106–110, 2004.