Mining a Software Developer's Local Interaction History

Kevin A. Schneider, Carl Gutwin, Reagan Penner and David Paquette
Department of Computer Science, University of Saskatchewan
57 Campus Drive, Saskatoon, SK S7N 5A9 Canada
{kas, gutwin, rpenner}@cs.usask.ca, dnp972@mail.usask.ca

Abstract

Although shared software repositories are commonly used during software development, it is typical that a software developer browses and edits a local snapshot of the software under development. Developers periodically check their changes into the software repository; however, their interaction with the local copy is not recorded. Local interaction histories are a valuable source of information and should be considered when mining software repositories.

In this paper we discuss the benefits of analyzing local interaction histories and present a technique and prototype implementation for their capture and analysis. As well, we discuss the implications of local interaction histories and the infrastructure of software repositories.

1. Introduction

We are interested in mining local interaction histories of a software development team to help coordinate their activities and to coordinate the change and use of project artifacts.

A software developer's interaction with a software repository includes editing source code but also involves actions to browse or locate source code. We are interested in recording and analysing this interaction, which we refer to as the developer's *local interaction history*. Our principle motivation is to use this information to support awareness in team based software development.

Developers normally change a local copy of the software under development. Periodically, the developer will synchronize their changes with the shared software repository. Although a portion of the developers' interaction with the local software artifacts may be recorded for the purpose of undoing changes and for recovering from previously saved versions, the interaction is not recorded in the shared repository and is incomplete when considering awareness support.

In our approach, as a developer changes software artifacts the different versions are recorded in a shared 'shadow' repository and analysed with respect to the structure of the software. Hierarchical containment of language entities (the structure of the software) is modeled separately so that we can track changes across the language entities. For example, we can track changes to a *method* across *classes* and *packages*. We use this strategy to monitor API (application programming interface) change and usage.

Mining local interaction histories has a number of potential applications, including:

- **Coordinating team member activities.** Monitoring changes to an API and monitoring API usage may be useful in supporting team awareness during software development. (The focus of this paper and our current prototype implementation.)
- **Identifying refactoring patterns.** Analysing local interaction histories may be useful for identifying novel refactoring patterns and coordinating refactorings that affect other team members.
- **Coordinating multiple file undos.** Tracking changes with respect to the structure of a software system may provide software development guidance when undoing a set of changes.
- **Identifying browsing patterns.** Local interaction history includes the developer's searching, browsing and file access activities. Analysing this browsing interaction may be useful in supporting a developer locate technical expertise or exemplars.
- **Project Management.** Recording the changes a developer makes to software with respect to communication logs or project plans may prove to be fruitful for organizing and managing a software project.

The next section discusses background and related work, focusing on coordination and communication issues in software development. Subsequent sections describe our approach and prototype. The implications of mining local interaction histories and the infrastructure of software repositories is discussed with our future research directions in the paper's conclusion.

2. Background and Related Work

Collaborative software development presents difficult coordination and communication problems, particularly when teams are geographically distributed [6, 8, 10, 12, 13]. Even though projects can be organized to make individual developers partly independent of one another, dependencies cannot be totally removed [10]. As a result, there are often situations where team members duplicate work, overwrite changes, make incorrect assumptions about another person's intentions, or write code that adversely affects another part of the project.

These problems often occur because of a lack of awareness about what is happening in other parts of the project. Unfortunately, current development tools and environments do not make it easy to maintain awareness of others' activities [1]. Awareness is a design concept that holds promise for significantly improving the usability of collaborative software development tools.

2.1. Collaboration in Software Development

Collaboration support has always been a part of distributed development – teams have long used version control, email, chat groups, reviews, and internal documentation to coordinate activities and give and gather information – but these solutions generally either represent the project at a very coarse granularity (e.g. CVS [3]), require considerable time and effort (e.g. reading documentation), or depend on people's current availability (e.g. IRC).

Researchers in software engineering and CSCW have found a number of problems that still occur in group projects and distributed software development. They found that it is difficult to: determine when two people are making changes to the same artifacts [10]; communicate with others across timezones and work schedules [6]; find partners for closer collaboration or assistance on particular issues [12]; determine who has expertise or knowledge about the different parts of the project [13]; benefit from the opportunistic and unplanned contact that occurs when developers are colocated [8]. As Herbsleb and Grinter [8] state, lack of awareness – "the inability to share at the same environment and to see what is happening at the other site" (p. 67) is one of the major factors in these problems.

2.2. Group Awareness

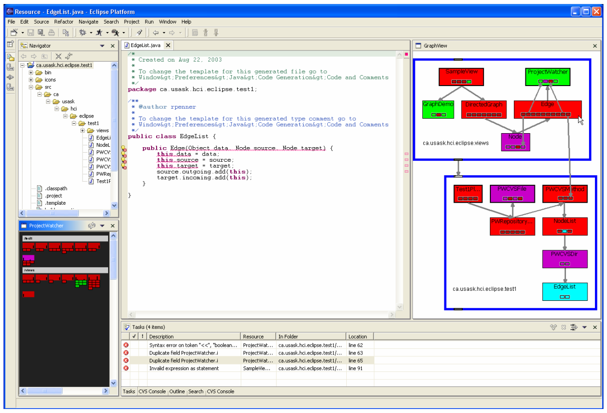
In any group work situation, awareness of others provides information that is critical for smooth and effective collaboration. This is *group awareness*: the understanding of who is working with you, what they are doing, and how your own actions interact with theirs [7]. Group awareness is useful for many of the activities of collaboration – for coordinating actions, managing coupling, discussing tasks, anticipating others' actions, and finding help.

In a software project, knowledge of others' activities, both past and present, has obvious value for project management, but developers also use the information for many other purposes that assist the overall cohesion and effectiveness of the team. For example, knowing the specific files and objects that another person has been working on can give a good indication of their higher-level tasks and intentions; knowing who has worked most often or most recently on a particular file indicates who to talk to before starting further changes; and knowing who is currently active can provide opportunities for real-time assistance and collaboration.

On software projects, awareness information is currently difficult to obtain from development environments: although some of the facts exist (e.g. from CVS logs) there are currently no low-effort means for gathering them. A few research systems do show awareness information (particularly TUKAN [12] and Plantír [11]), but little support exists in more widespread environments.

3. Project Watcher

ProjectWatcher is a prototype system that gathers information about project artifacts and developer's actions with those artifacts, and that visualizes this awareness information in the Eclipse [5] development environment (Figure 1). ProjectWatcher consists of two main parts – the mining component and the visualization plugins.



```
/*
 * Created on Aug 22, 2003
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package ca.usask.hci.eclipse.test1;

/**
 * @author rpenner
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class EdgeList {

    public Edge(Object data, Node source, Node target) {
        this.data = data;
        this.source = source;
        this.target = target;
        source.outgoing.add(this);
        target.incoming.add(this);
    }

}
```

✓	!	Description	Resource	In Folder	Location
		Syntax error on token "<<", "boolean...	ProjectWat...	ca.usask.hci.eclipse.test1/...	line 62
		Duplicate field ProjectWatcher.i	ProjectWat...	ca.usask.hci.eclipse.test1/...	line 63
		Duplicate field ProjectWatcher.i	ProjectWat...	ca.usask.hci.eclipse.test1/...	line 65
		Invalid expression as statement	SampleVie...	ca.usask.hci.eclipse.test1/...	line 91

Figure 1. ProjectWatcher in Eclipse. Visualizations are at lower left and upper right.

The mining component analyzes the source code of a project to produce facts for use by the ProjectWatcher visualization plugin. The mining component gathers information on the structure of the project and also on the current and historical activity of the project team members.

To be able to gather developer activity information, a shadow CVS repository of the project is maintained (Figure 2). User edits are auto-committed to the shadow repository as developers edit source code files. Although Eclipse provides a local history of changes, we require that the changes be available to other developers in the software development team and so publishing them in the shadow repository gives us that facility. As well, we are able to record actions along with changes to software artifacts, and we are able to commit changes at different time intervals.

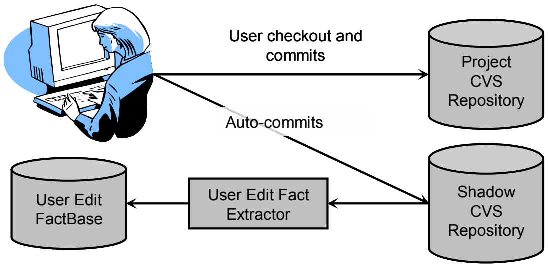


Figure 2. Capturing User Edits. A shadow software repository is used to record the activities of a software developer.

The user edits mining component analyzes the shadow CVS repository to obtain facts about who has been editing the class methods and when. A version of a file is created each time it is auto-committed to the shadow repository. The mining component analyses the differences between versions to track API usage and API change.

The mining component is implemented in two stages and may either be run on the shadow software repository or on the shared software repository (Figure 3). Stage one uniquely names all entities in the project while extracting the entity and relationship facts. This process is accomplished with a TXL program using syntactic pattern matching [2, 4]. At this point, the method call facts are not uniquely identified since we do not have sufficient information to identify which package or class the method being called belongs to. This resolution is accomplished by stage two, the method call resolver.

The method call resolver extracts facts from the project source code and integrates them with the facts extracted from stage one. Next, the method call facts are analyzed to determine which package and class the method that was called belongs to. This process involves resolving the types of variables and return types of methods that are passed as arguments to method calls. The types of all the arguments are identified, and then scope, package, class, and method facts are analyzed to determine which package and class the method belongs to. To resolve calls to the Java library, the full Java API is first processed by the ProjectWatcher mining component (this is only done once for all projects). Not all calls may be resolved, however for our purpose the accuracy of the method call resolver is adequate.

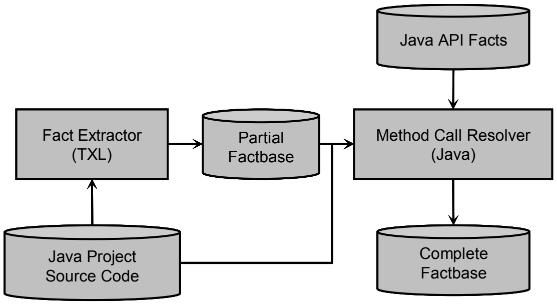


Figure 3. Mining User Edits. In a two stage process, package, class and method facts are extracted and combined with Java API facts. The facts are used by the visualization component to convey API use and API change information.

The complete factbase contains uniquely identified facts indicating all packages, classes, methods, variables, and relationships for a Java project and all user edits. These facts are used by the visualization plugin to show activity and proximity information. The time and space required for fact extraction and factbase storage depends on the size of the code. For example, ProjectWatcher has been tailored for Java, and mining the Java Development Kit 1.4.1 results in 202 package facts, 5,530 class facts, 47,962 method facts, and 106,926 call facts.

4. Awareness Visualization

4.1. Activity Awareness

ProjectWatcher visualizes team members' past and current activities on project artifacts. The visualization uses the ideas of interaction history [9] and overviews: the interaction history is a record all of the actions that a person undertakes with a project artifact (gathered unobtrusively by the mining component as people carry out their normal tasks); the overview representation is a compact display of all the project artifacts, that can be overlaid with visual information about the interaction history. Although some tools such as CVS front-ends do have limited visualization (e.g. by colour on the project tree), our goal here is to collect much more information about interaction, and provide much richer visualizations that will allow team members to gather more detailed awareness information.

ProjectWatcher plugins use the extracted fact base to create a visual model of what each developer is doing in that

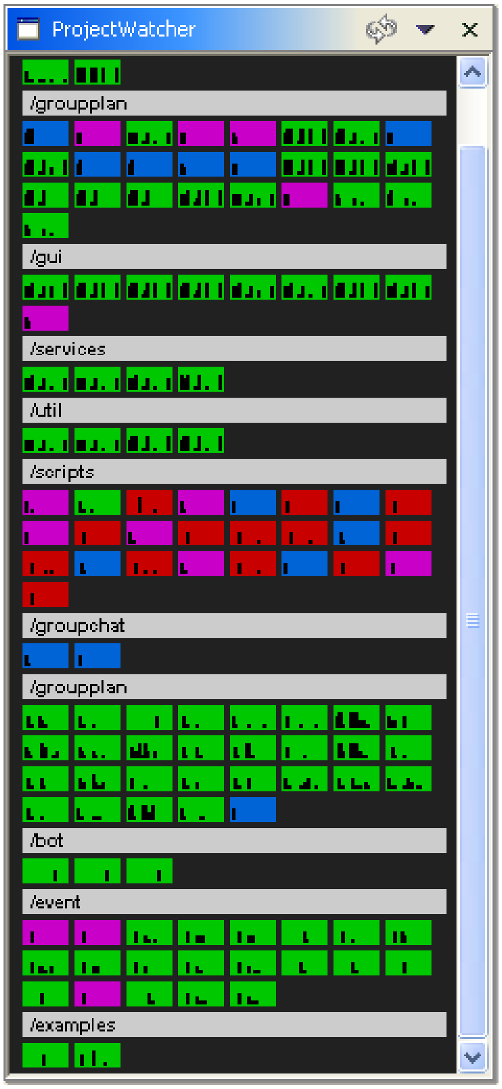


Figure 4. Project overview plugin showing packages (grey bars) and classes within each package (coloured blocks). Colour indicates who edited the class most recently. Black marks inside class blocks chart edits since project start.

project space. In the overview plugin (Figure 4), project artifacts are shown in a simple stacked fashion that displays packages, files, classes, and methods. Artifacts are always stacked by creation date, so that their location in the overview can over time be learned by the user. On this basic (but space-saving) representation, we overlay awareness information. First, each developer is assigned a unique colour, and this colour can be added to the blocks in the overview based on a set of filters. Common filters include who has modified artifacts most recently, or modified them most often. Second, we show a summary of the activity history for each artifact with a small bar graph drawn inside the object's rectangle; bars represent amount of change to the class since its creation. Finally, more information about an artifact can be obtained by holding the cursor over a rectangle: for example, the name of the class and a more detailed bar graph, along with details about the state of the class compared to the CVS repository.

4.2. Proximity Awareness

Following on from a basic understanding of others' activities is the question of proximity – that is "who is working near to me?" in terms of the structures and dependencies of the software system under development.

The notion of distance to another person has not been studied extensively, although it has been explored previously in Schümmer's TUKAN [12]. We have developed a visualization tool (Figure 5) that makes it easier to see proximity-based groups. Once actions are mapped to the dependency structure, the graph is presented in visual form with people's locations and proximities made explicit.

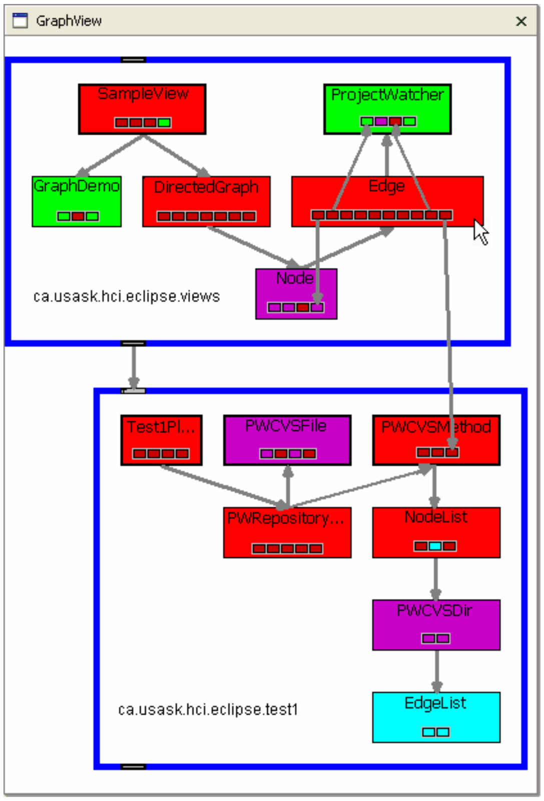


Figure 5. ProjectWatcher graph view

5. Conclusion

We have presented a system for mining local interaction histories to help address some of the awareness problems experienced in distributed software development projects. The system observes a software developer's activities in a software development environment and records those actions in relation to the artifact-based dependencies extracted from source code. Visualization plugins represent this information for developers to see and interact with. Although our prototypes have limitations (particularly in terms of project size), they can provide developers with much-needed information about who is working on the project, what they are doing, and how closely linked two developers are.

Our experience suggests a number of directions for mining software repository research, including:

- **Content.** Research on awareness often monitors a software development teams' interaction with a shared software repository. Unfortunately, the granularity of check-in and check-out is usually too coarse to adequately monitor change. This suggests that the content of shared software repositories should also include local interaction histories.

- **Rapid incremental processing.** For our purposes it is important that the computation of source facts and their resolution be relatively efficient to support interactive visualizations.

- **Robustness.** Our analysis may process source that is currently being edited and so the source may not be well-formed. We require that fact extraction and resolution needs to support analysis under ongoing change.

Our future plans with the system involve both improvements and new directions. With the current system, we plan to continue refining our representations and filters to determine how the information can be best presented to developers. We currently visualize source code that is in the process of being edited, and therefore the source code may be inconsistent, incomplete and frequently updated. We are investigating techniques for improving the robustness and performance of the mining component and visualizing partial information given these circumstances.

Longer range plans involve extensions to the basic ideas of project artifacts and interaction histories. We plan to extend our artifact collection to include entities other than those in source code. Many other project artifacts exist, including communication logs, bug reports and task lists. We hope to establish additional facts to model these artifacts and to use the new artifacts and their relationships in the awareness visualizations.

We can also extend our use of the interaction histories to other areas. For example, recording developers' interaction history and extracting method call facts from the source code provides us with basic API usage information. We can present this information in a future plugin to provide awareness of technology expertise. A developer wishing to know how to use a particular Java API feature may be presented with a list of developers who have used the feature frequently or recently. Alternatively, the visualization plugin may present this information overlaid on the project's dependency structure.

Acknowlegment

The authors would like to thank IBM Corporation for supporting this research.

References

[1] M. C. Chu-Carroll and S. Sprenkle. Coven: brewing better collaboration through software configuration management. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97. ACM Press, 2000.

[2] J. R. Cordy, T. R. Dean, A. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[3] CVS. Concurrent Versions System. Available online at http://www.cvshome.org/.

[4] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. Malton. Using design recovery techniques to transform legacy systems. In *ICSM*, pages 622–631, 2001.

[5] Eclipse. Available online at http://www.eclipse.org/.

[6] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The geography of coordination: dealing with distance in r&d work. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 306–315, 1999.

[7] C. Gutwin and S. Greenberg. A descriptive framework of workspace awareness for real–time groupware. *Computer Supported Cooperative Work*, 11(3):411–446, 2002.

[8] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, pages 63–70, 1999.

[9] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *Proceedings of CHI'92*, pages 3–9. ACM Press, 1992.

[10] R. E. Kraut and L. A. Streeter. Coordination in software development. *Communication of the ACM*, 38(3):69–81, 1995.

[11] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proceedings of ICSE 2003*, pages 444–454, 2003.

[12] T. Schümmer. Lost and found in software space. In *Proceedings of the 34th HICSS*, 2001.

[13] B. Zimmermann and A. M. Selvin. A framework for assessing group memory approaches for software design projects. In *Proceedings of the conference on Designing interactive systems*, pages 417–426. ACM Press, 1997.