Commit 2.0

Marco D'Ambros, Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano, Switzerland
{marco.dambros, michele.lanza}@usi.ch

Romain Robbes
University of Chile
Chile
romain.robbes@gmail.com

ABSTRACT

Commit comments written by developers when they submit their changes to a versioning system are useful for a number of tasks: Developers write commit comments to document changes and as a means to communicate with the rest of the development team; Researchers mine commit-related data contained in software repositories to support software evolution and reverse engineering activities. However, the support provided by IDEs is restricted in this respect, as they limit the users to use only text to document their changes.

We present *Commit 2.0*, an IDE enhancement to enrich commit comments using software visualization. Commit 2.0 generates visualizations of the performed changes at different granularity levels, and lets the user annotate them.

1. INTRODUCTION

It has become a widely accepted practice for software projects to use a versioning system, such as CVS, SVN, or Git, to manage the evolving code base.

Many versioning systems (*e.g.,* Git, CVS, ClearCase) allow the developers to write a comment at commit time and store it together with the changes. The information contained in such comments is inherently useful: For software development commit comments are used to document changes and as a means of communication and synchronization among the development team. With respect to software evolution, many approaches in the field of Mining Software Repositories (`http://www.msrconf.org`), deal with mining and analyzing this commit related information [1–3].

Given the importance of commit comments data, developers should write meaningful comments which exhaustively document the changes. However, developers do not always document all the changes in the commit comment. This happens for a number of reasons which can vary among software projects, development teams and organizations, because of different practices and different development rules. Still, a common cause is that writing exhaustive comments is time-consuming, and -being the last step of a coding session- the necessary time and energy is not always available. Moreover, for commits with many changes, the developers might not remember all of the modifications. Another problem of commit comments is the lack of context in which changes occur. For example, if a developer changes 5 classes to add a new feature in a system, in the commit comment he can describe the feature and list classes and methods. However, when reading such a comment, it is not clear where the modified classes (and methods) are in the system, which relationships they have and what the magnitude of the change is.

We argue that IDEs should provide means to ease the task of documenting changes in commits. We describe an approach, called *Commit 2.0*, to enrich commit commit comments using software visualization, which provides a visual context to the changes and facilitates their documentation. Since versioning system repository do not support media, but only text, for commenting the changes, *Commit 2.0* uses a blog as alternative repository and persistency mechanism.

Structure of the paper. In Section 2 we provide an overview of our approach. We describe the visualization part of *Commit 2.0* in Section 3 and present an example usage of the approach in Section 4. After discussing the main benefits and the current limitations of the approach in Section 5, we provide implementation details in Section 6. In Section 7 we look at related work, and we conclude in Section 8 by summarizing the contributions of the paper and by presenting how we plan to extend *Commit 2.0*.

2. COMMIT 2.0 IN A NUTSHELL

Commit 2.0 is an IDE enhancement which generates visualizations of the changes at different granularity levels, and lets then the user enrich them with annotations. Figure 1 provides an overview of how *Commit 2.0* works:

- *Commit 2.0* is triggered when the developer wants to commit the code or document previous commits.
- At this point, in addition to the standard dialog where the developer can write the comments, *Commit 2.0* shows also a coarse-grained visualization of the system which highlights the changes. The visualization is automatically generated by comparing the last version of the project in the repository with the locally modified version. The developer can interact with the visualization by inspecting entities, moving figures, zooming in and out and, most importantly, adding annotations. Annotations are rendered as floating text boxes and can be placed by the developer next to a modified entity to detail and comment the corresponding change.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Web2SE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-975-6/10/05 ...$10.00.

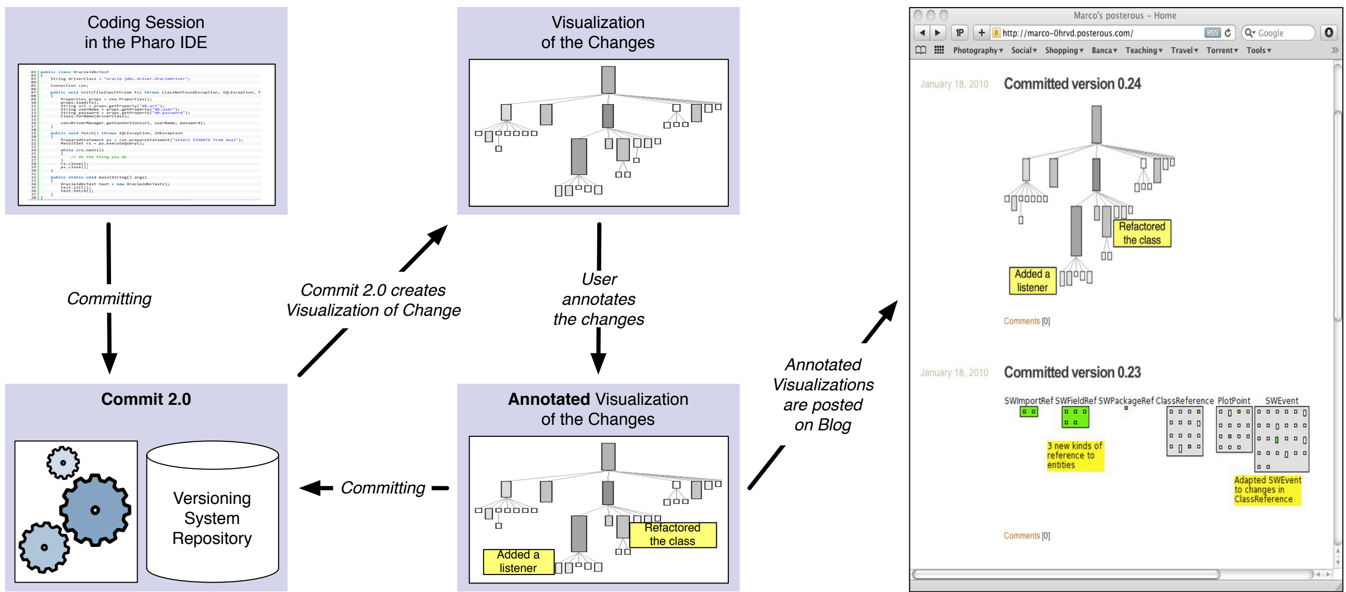

Figure 1: *Commit 2.0* in a nutshell

- The developer can select one or more entities and spawn a fine-grained visualization. As before, the developer can document the changes in the fine grained view by adding an arbitrary number of annotations.

- Once the developer has completed the documentation of the changes, with one annotated coarse-grained visualization and an arbitrary number of annotated fined-grained visualizations, she can complete the commit by submitting the changes, the (traditional) textual commit comments, and the annotated visualizations.

- The code and the (traditional) comments are submitted to the versioning system repository, as without *Commit 2.0*. The annotated visualizations are posted on a Posterous blog (`http://posterous.com`), using the version number as a title.

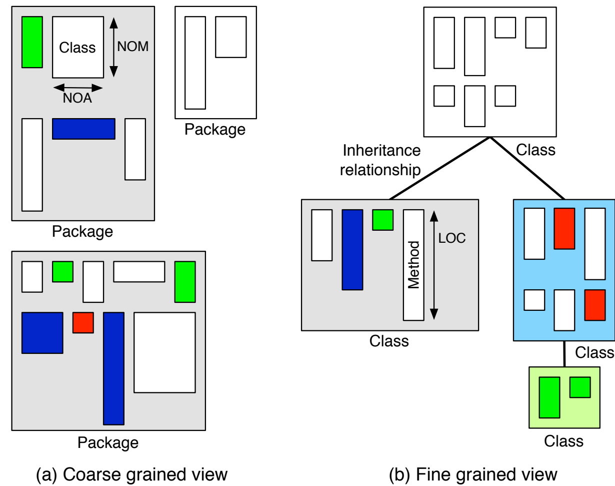

Figure 2: Visualizing changes in Commit 2.0

3. VISUALIZING CHANGES

Figure 2 shows the principles of the visualizations used by Commit 2.0 to depict changes.

The coarse-grained view shows all the packages in a system as rectangle figures, and within each rectangle all the classes belonging to the corresponding packages are also depicted as rectangle figures. The width of the rectangles representing classes is proportional to the number of attributes, and the height to the number of methods. The size of the package figures is set to fit all the class figures in it.

The fine-grained visualization is a graph whose nodes represent classes and edges represent inheritance relationships. Within each node, all the methods belonging to the corresponding class are represented as rectangle figures, where their height is proportional to their number of lines of code (LOC). We do not directly map any class metric on the size of class figures. However, since the size of method figures is proportional to LOC, and the size of class figures is set to fit all contained method figures, the size of classes is proportional to the weighted number of methods, where the weighting factor is LOC. The fine-grained view is generated from a selection of packages and/or classes in the coarse grained view. Since the fine-grained view does not visualize packages but only classes and methods, selecting a package in the coarse-grained view is the same as selecting all the classes belonging to that package.

In both views, the following color scheme is applied to highlight the changes: Red represents deletion (the corresponding entities have been deleted), green represent addition, blue modification and gray represents indirect changes (if an entity is modified the container entity has an indirect change). When changes concern outer entities (*e.g.,*

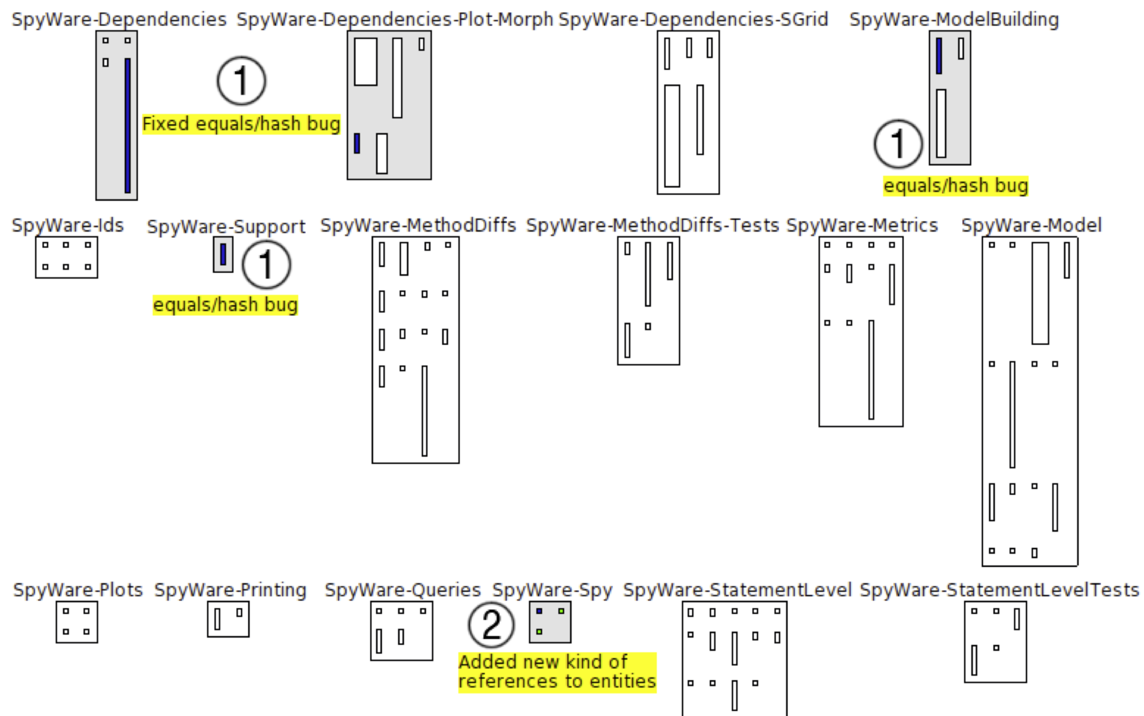

Figure 3: Example of a coarse-grained, annotated *Commit 2.0* change visualization

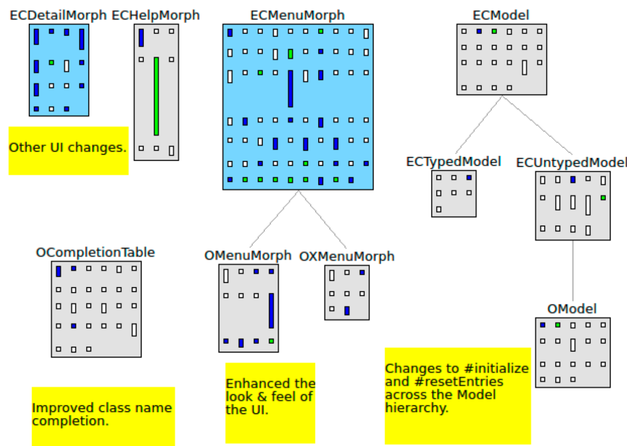

Figure 4: Example of a fine-grained, annotated *Commit 2.0* change visualization

packages in the coarse grained view and classes in the fine grained one), we use lighter color, *i.e.,* light green for addition and light red for deletion. Finally, in the fine grained view we represent modification of classes (the outer entities) with light blue: Only structural modifications are considered, *i.e.,* additions or removal of instance variables.

Example: Coarse-grained view Figure 3 shows an example coarse grained view obtained from the Spyware software system [5]. We see that 5 packages were modified, mostly because the contained classes were modified (1), or added (2). Further, looking at the annotations added by the developer, we see that the changes fixed a bug (1) and added a new feature (2). The visualization allows us to localize the changes, *i.e.,* understand which classes/packages were modified to fix the bug and add the feature.

Example: Fine-grained view Figure 4 shows a visualization of fine-grained changes in OCompletion (see next section). Light blue classes had attributes added or removed. The view provides an insight on the impact of the changes on the system, in terms of changes to its behavior.

4. AN EXAMPLE: OCOMPLETION

OCompletion is a code completion tool based on an optimistic completion algorithm [6]. It shows a tiny menu under the cursor when the user is typing in order to complete the identifiers the user is entering. Unlike other code completion tool, it is always active and does not need to be explicitly invoked. Pressing the tab or enter key allows one to complete the text being entered with one of the proposed identifiers. Being included in the Pharo IDE, OCompletion is continuously maintained. In this example, we show how one can use *Commit 2.0* to document the changes between versions 35 and 41 of OCompletion.

Step 1: Choosing the versions *Commit 2.0* in integrated in Monticello, Pharo's source code control system. The browser shows a list of versions, their author, time stamps and commit comments, as shown in Figure 5. After selecting a version, the *Commit 2.0* button becomes active. When clicked, it opens a menu to select the previous version against which to document the changes. When the user selects the

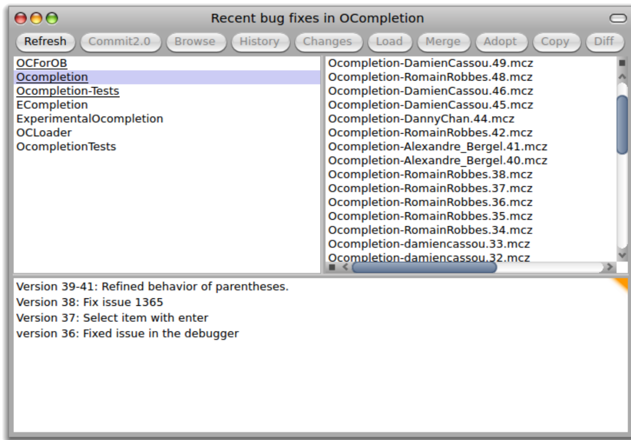

Figure 5: Integration of *Commit 2.0* with Monticello

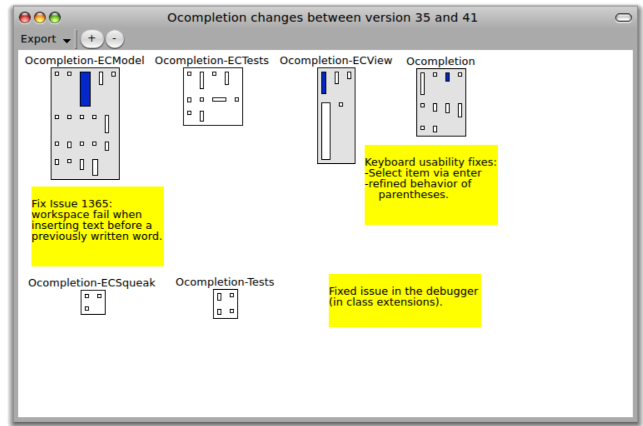

Figure 6: Coarse-grained annotation of changes

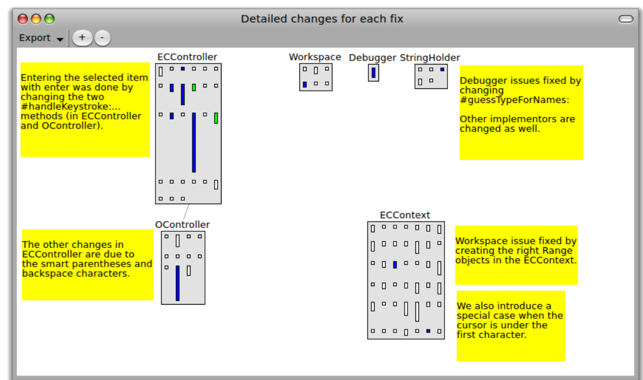

Figure 7: Detailed comments on each change

second version, *Commit 2.0* import both of the version in a single annotated moose model, and opens change-aware visualizations. In our case, the user first selects version 41 of OCompletion as the last version, and then version 35 as the first version as the period to document. One can document several versions at once if the delta between them is small. In parallel *Commit 2.0* starts to build the email it will send to Posterous, putting as default subject the last version's number, and as default text the commit comment.

Step 2: Documenting the changes Between version 35 and 41, several bug fixes and enhancements were integrated in OCompletion. First, a bug fix related to the use of OCompletion in the debugger was integrated in version 36. Second, the ability to select a menu item with the enter key, in addition to the tab key, was included in version 37. Then, in version 38, another bug fix was integrated. Finally, in versions 39 to 41, incremental work proceeded in order to treat character input in a more intuitive fashion. All of these changes are scattered through the code base. The developer adds annotations near the changed entities to document the location and intent of each change. The developer is free to move the graphical elements as he sees fit, in order to better group related changes together.

In our example, the developer enters 3 annotations in the coarse-grained views, describing the location of the changes in the system. The changes in version 37 and 39-41 are grouped as they concern the same portion of the code. The annotations are shown in Figure 6. Note that the developer also elected to change the title of the window, as a higher-level change description.

After this, our developer annotates the fine-grained visualization. This one features only the changed classes and their methods. The developer once again uses the placement of the entities and the annotations to show the extent of each change to each method. One change in class ECContext, comprising two methods, is documented with two annotations. Other changes are documented by one annotation each, but the spacing of the entities clearly shows that some changes span two or three classes, unlike others. All in all, the process takes a handful of minutes.

Step 3: Posting to posterous Once the windows are ready, the developer clicks the window button on the top right corner. This summons a window-specific menu that contains an "add to email" item. Upon selecting this menu item,a screen capture of the window is taken and added to the email *Commit 2.0* is composing. Clicking on the send button asks for the developer's SMTP password, and then sends the email to the Posterous email account. Posterous generates a blog post with an image gallery, as shown in Figure 8. Since the developer was documenting the changes to several versions at once, he also changed the title of the blog post to better reflect the changes. From then, other developers will see the new post in their RSS reader's RSS feed. If set up accordingly, Posterous can also post to web sites such as twitter, flickr or Facebook – so that fans of a tool's Facebook page may be notified of the latest changes.

5. DISCUSSION

The main benefit of Commit 2.0 is that it provides a visual context to changes, which eases both their documentation and comprehension, *i.e.,* understanding the rational behind the changes. The visualizations can be used by a development team as a mean to communicate. However, the usage of Commit 2.0 requires an investment of time greater than just write text comments. We argue that this investment

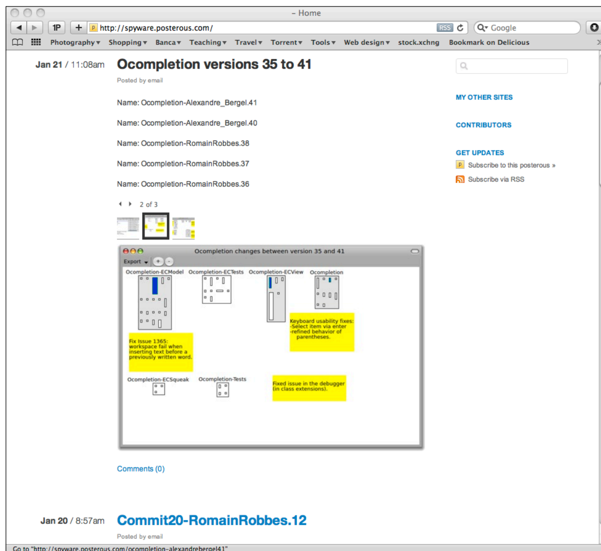

Figure 8: Email-generated posterous blog post

is worth as it produces better documentation for changes. To help the developers spotting all the changes and to make the approach scalable, the views in Commit 2.0 are fully interactive, allowing the user to inspect entities, moving them around and zooming in and out. Moreover, Commit 2.0 allows not only the visualizations but also any UI element of the application to be annotated and documented, *i.e.,* posted to the blog. Finally, the visualizations are kept simple so that they are easy to learn and understand.

On the downside, Commit 2.0 is a prototype which was never tested with large software systems. A second limitation consists in the storage of the annotations: they are only stored in the visualizations, but not in the versioning system repository. Also, they are not linked to the entities they refer to. Moreover, the visualizations are stored as scalar images (png or jpg), while a vectorial format (e.g. SVG) is not supported in the current implementation. Another limitation is that the code repository and the blog are separated and developers have to look at them with different tools.

Usage of Posterous. We decided to post the annotated visualization on a blog because, to our knowledge, no versioning system supports the use of images to be attached to commit comments. Using a blog has several advantages:

- We did not have to change anything in the versioning system and it is applicable to any versioning system.
- Since every blog post has a permalink, every commit has a permalink too, which can be used to access the changes and understand them, for example before checking out the code.
- Developers can comment and discuss the changes by commenting the blog posts.
- Developers and project managers can subscribe to the RSS feed of the blog and be automatically notified about changes in the system, with details about them (the visualizations). Therefore, Commit 2.0 and RSS feed can be used as a visual monitoring system for the software project.
- The blog can be integrated with Facebook or other social networking web application, to create a software project development community composed of developers, project managers, testers and people interested in the development of the project.

Posterous allows us to post the visualizations easily, by sending e-mails with the visualizations as attachments. Moreover, Posterous allows multiple accounts (multiple emails) to post. In this way, the identity of the developer who commits the code is preserved.

6. COMMIT 2.0 IMPLEMENTATION

Commit 2.0 is developed in Smalltalk and is available for the Pharo Smalltalk IDE (`http://pharo-project.org`). *Commit 2.0* leverages the following technologies:

Monticello is a Smalltalk-specific SCM system. It versions packages, classes and methods instead of directories, files and lines. Hence the differences between two versions are of a much higher level than for a conventional SCM system. We extended Monticello so that *Commit 2.0* can easily access its code repositories, at the click of a button.

Moose is a reverse engineering platform. It can automatically import Monticello versions of Smalltalk code in its meta-model. We extended Moose to import both a version and the differences with a previous version, enriched with annotations describing the changes.

Mondrian is a visualization framework used by Moose. It defines a domain-specific language to easily build visualizations. We built two Mondrian scripts in order to show our change-enriched versions.

SMTPClient is a simple library to send emails. *Commit 2.0* uses it to communicate with posterous.

Posterous is a web 2.0 blogging platform. When sent an email, posterous generates a blog post based on the email's content and title.

In a way, *Commit 2.0* is merely the glue between these components.

7. RELATED WORK

Visualizing Versioning System Data. A number of approaches were introduced to visualize versioning system data. Xie *et al.* presented CVSViewer3D [13], a tool which extracts, processes, and visualizes information from CVS repositories. The visualization allows users to define multiple views of the change history data and to look at it at different granularity levels. Girba *et al.* defined a measurement for code ownership based on information extracted from a CVS repository and presented the Ownership Map visualization [1]. The visualization, which displays the history of versioned files, using different colors to represent different authors, is helpful to understand when and how different developers interacted in which way and in which part of the system. Taylor and Munro [10] used visualization together with animation to study the evolution of a CVS repository. The technique, called revision towers, allows the user to find out where the active areas of the project are and how work is shared out across the project. Rysselberghe and Demeyer [11] used a simple visualization of CVS data to recognize relevant changes in the software system such as: (1) unstable components, (2) coherent entities, (3) design and architectural evolution, and (4) fluctuations in team productivity. Voinea and Telea [12] proposed the CVSgrab tool

which supports querying, analysis and visualization of CVS based software repositories. Their tool allows the user to produce views, to interact with them, to do querying and filtering and to customize the view through a rich set of metrics computed from the CVS data.

The difference between these approaches and Commit 2.0 is that they visualize the data a posteriori to support retrospective analysis, while Commit 2.0 visualizes changes at commit time to support their documentation.

Enhancing Versioning and Awareness. Researchers proposed several approaches to enhance IDEs and the way they monitor the evolution of source code. In [9] Schneider *et al.* argue that local interactions, *i.e.,* the way developers interact with their local copies of the source code, are a valuable source of information which should be considered when mining software repositories. The authors proposed a technique and a prototype implementation to capture and analyze such local interactions. Robbes *et al.* developed an approach to record fine-grained source code changes as they happen, instead of recovering them from coarse-grained, file-based versioning system archives [7]. The approach was used in a variety of applications [5]. These approaches improve versioning systems by refining the code model, while *Commit 2.0* enriches the documentation of code changes with visualization. *Commit 2.0* could be applied on the refined changes these approaches produce.

Awareness of changes is also an active research topic. Sarma *et al.* developed Palantír [8], a workspace awareness tool that complements configuration management systems by (1) informing a developer of which other developers change which other artifacts, (2) calculating a measure of severity of those changes, and (3) visualizing the information in a non-obtrusive manner. Lanza *et al.* [4] enhanced the approach of Robbes *et al.* to record source code changes as they happen and broadcast them to other developers of the team: Developers are aware of potential conflicts before committing the code. While the goal of these approaches is to provide awareness of changes the aim of *Commit 2.0* is to better document the changes as they are committed.

8. CONCLUSION

In this paper we have proposed a visual approach, and the corresponding implementation, to support the documentation of software changes at commit time. Our technique generates coarse grained and fine grained visualizations of the changes, and let the developer enrich them with annotations. Since images are not supported as comments in versioning systems, we create an alternative repository by means of a blog, where we store the annotated visualizations. Such blog, besides serving as a communication mean for the development team, acts as an entry point allowing us to leverage the web 2.0 technologies.

Future Work. In this paper we presented a first prototype of our Commit 2.0 tool, in which the annotated visualizations are posted on a Posterous blog. In the future we plan to investigate both web 2.0 technologies to fill the gap between the blog and the IDE, and IDE enhancements to support and version the visualizations. As part of our future work, we also plan to conduct a user study to evaluate the effectiveness of documenting changes with Commit 2.0.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

9. REFERENCES

[1] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[2] 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.

[3] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR 2008)*, pages 99–108. ACM, 2008.

[4] 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)*, pages xxx – xxx. IEEE CS Press, 2010.

[5] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, Switzerland, Dec. 2008.

[6] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd ACM/IEEE International Conference on Automated Software Engineering)*. ACM Press, 2008.

[7] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering)*, pages 27–41, 2007.

[8] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 444–454. IEEE Computer Society, 2003.

[9] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings of the First International Workshop on Mining Software Repositories (MSR 2004)*, 2004.

[10] C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.

[11] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

[12] L. Voinea and A. Telea. An open framework for cvs repository querying, analysis and visualization. In *Proceedings of the 2006 international workshop on Mining software repositories (MSR 2006)*, pages 33–39. ACM, 2006.

[13] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of cvs repository information. In *Proceedings of WCRE 2006*, pages 231–242. IEEE CS, 2006.