Making the Evolution of Graphical Models Visible

Thomas Baar and Issam Bendaas

Hochschule für Technik und Wirtschaft (HTW) Berlin
Wilhelminenhofstraße 75A
D-12459 Berlin, Germany
thomas.baar@htw-berlin.de
issam.bendaas@gmail.com

Abstract

With adoption of the UML and other graphical languages by software industry, graphical models became a cornerstone in today's software development practice. As other artefacts such as program source code, graphical models evolve over time and are, thus, put regularily under version control.

In order to deeply understand the role an artefact plays within a project, it is sometimes helpful to review the history of this artefact. While there are numerous tools available that make it easy for a user to grasp the evolution of textual files (or even portions of it), an adequate support for graphical files has remained to be an area of niche products.

In this paper, we argue that a better support for reviewing the version history of graphical files can facilitate the work with graphical models. In order to support this claim, we implemented a prototypcical tool that can extract and display the version history of any graphical file stored in a GitHub-repository. In addition, users can annotate each version of a file with comments, what turns our tool into a review tool for software projects. Recently, we started to use the tool is a software engineering course to give students better feedback on complex UML models they have to develop iteratively.

Keywords: Version Control, Repository Mining, Human Computer Interface, Graphical Modeling, Textual Modeling

1 Motivation

Software projects as well as other projects, in which electronic documents (artefacts) are produced, take today heavily advantage of version control systems. Systems, such as SVN, Git, Mercurial, are very popular, easy to use and nicely integrated into modern Integrated Development Environments (IDEs), e.g. Eclipse. Version control systems are mandatory for team members to synchronize their work results, but also one-person projects can profit a lot from these systems.

Version control systems offer their users a lot of functionalities. The most important one when working within a team is *to synchronize* the artefacts team members work on. Other basic features version control systems provide are *to revert* an artefact to a previous version and *to see the history of* an artefact or even the history of the whole project (who committed what at which time).

Unfortunately, many of these functionalities work well only for artefacts being textual files. For example, to synchronize the local copy of the project artefacts (called local workspace) with the latest versions of the artefacts stored in the repository requires sometimes *to merge* different versions of the same artefact.

Merging textual files, e.g. program source code, always requires intervention from the user. For all portions that deviate in the files to be merged, the user has to decide whether the variant of the local file or the one of the file from the repository should be taken and marked as

valid. Merging of textual files can easily become an error-prone task, depending on the size and the complexity of the portions that deviates in both files. In order to merge two textual files correctly, the user has to have a mental model of the syntax and semantics of the merged files. For example, if the textual files are program files, the merged file should again be a syntactically correct program. Once executed, the program should behave 'as expected'. To summarize, the merge of textual files is only possible because the user has an intimate knowledge of syntax and semantics of the formalism (e.g. programming language) used within the file.

Merging of graphical files cannot be done the same way as the merge of two textual program files as sketched above. This is, because if a file encodes a graphics it uses an internal format that the user is not familiar with. Even if the graphical file is given in an XML-format — which is used by many UML tools to store UML models — the user would need tool support in order to detect in which parts two versions of a graphical file deviate. Then, tool support is also needed for the merging process itself. Only with adequate tool support the user is able to choose those parts from the two input files that should form the merged graphical file, i.e. the output file.

The differences between textual and graphical files wrt. merging have some far reaching consequences for the way we use them in software development projects. The authors of this papers have never tried to merge two UML models or two models of a graphical domain-specific language (DSL), because they are not aware of any adequate tool support. In contrast to this, the merge of two versions of a program source code file or of a textual DSL model is possible just with the help of a text editor.

To solve the described merge problem, one could propose and/or implement techniques for detecting differences in graphical models, i.e. detecting differences between graphs, and techniques for easy selection of parts of a graph to form a new graph. The resulting implementation of such a research agenda, however, would highly depend on the internal format for representing the graph and, thus, would highly depend on the concrete editor used to create and manipulate the graphical models.

Our research goal is rather different! We are looking for pragmatic solutions that make it easier to work with graphical models in software development projects. Since graphical models evolve over time (as any other project artefacts), good tool support for version management is crucial for the acceptance of model-based software development. The above sketched merge-problem has to be solved in close collaboration with the vendors of graphical model editors, since the internal format of the graphical model has to be taken into account. In this paper, we focus on the problem to review the version history of graphical models. Without special tool support, it is very cumbersome to see previous versions of a graphical model. Basically, the user has to do the following steps: (1) find out, which previous versions exist, (2) ask the repository to provide these versions, and (3) open the model with an appropriate editor. Especially the last point can be very time-consuming since different versions of the graphical model have to be opened/closed sequentially as many UML tools, for example, can work at any time only with one model.

In this paper we provide an approach to review the version history in a much more efficient way. Our solution is based on standard formats of graphical files such as .png and .jpg. Consequently, our solution is agnostic to the editor that was used to create the graphical model and is, thus, widely applicable.

We back our approach with a prototypical tool. This tool works on Git-repositories and provides an intuitive interface to view the version history of any graphical file stored in these repositories. As of writing this paper, a severe limitation is that the Git-repository must be hosted by *GitHub*, since we rely on the *GitHub API* to access the repository content.

Our tool does not only allow to view the version history in a very comfortable way, but also allows to add annotations to each of the graphical file's versions. An annotation is basically a layer on top of a file and can contain shapes like circles and rectangles put on a certain position as well as textual comments. The annotations are not stored in the repository itself, but within another (backend) system. Thus, our tool is also a *review tool*: The user can add comments to content stored within a Git-repository without changing (and polluting!) the content of the repository itself!

The remainder of the paper is organized as follows: In Section 2 we review related work and especially describe some commercial online-services, which have very similar goals but which lack the flexibility of our tool. Sections 3 and 4 describe the architecture of the tool and illustrate its usage with a small example. We close in Section 5 with a summary of what has been achieved and give an outlook on future work.

2 Related Work

Mining software repositories (MSR) is a discipline of empirical software engineering with a long history, but with a community being still very active. The main concern is to aggregate information from software repositories, e.g. incident management systems, deployment logs, source code repositories, communication archives, etc. (see [3] for an overview). An important application area is fault and effort prediction based on the current history of a project. As an example, Zimmermann et al. give in [8] a prediction model for future bugs of the tool *Eclipse*.

The MSR community focuses on the automatic extraction and processing of information from several data sources in order to conclude some properties of the developed system wrt. properties such as maintainability, extensibility, reliability. Our goal, however, is to make the access on the version history of graphical files more user-friendly.

Pixelapse [7] and *LayerVault* [5] are two commercial service providers. They offer cloud-services for file version management as well as for project collaboration. Both providers target on non-IT professionals as customers. The main task is to version control graphical artefacts and to review easily the version history of a graphical file. The features these services offer such as graphical file comparison or annotating graphical files with comment annotations have been an inspiration for our tool. The main difference to our approach is that the two services can be used by their customers only for those repositories managed by the services themselves while our tool provides an easy access to any Git repository.

3 Our Approach

In Section 1 we explained a problem many software developers face today when working in projects with graphical artefacts: IDEs, e.g. Eclipse, lack good support for merging and even for reviewing graphical artefacts stored in a repository. Providing such support for graphical models, e.g. models formulated in UML or a graphical DSL, would require an intimate knowlegde of the internal data format used by the corresponding UML- or DSL-editors as argued in Section 1.

Our approach makes it easier to review graphical artefacts stored in a repository, no matter which editor was used to create the graphical artefacts. Thus, our approach is applicable for many projects. However, the user has to pay a price for such flexibility: Whenever a graphical model (stored as file in an editor-specific internal format) is committed to the repository, the user has to commit the graphical model exported as a .png- or .jpg-file as well. Note that this

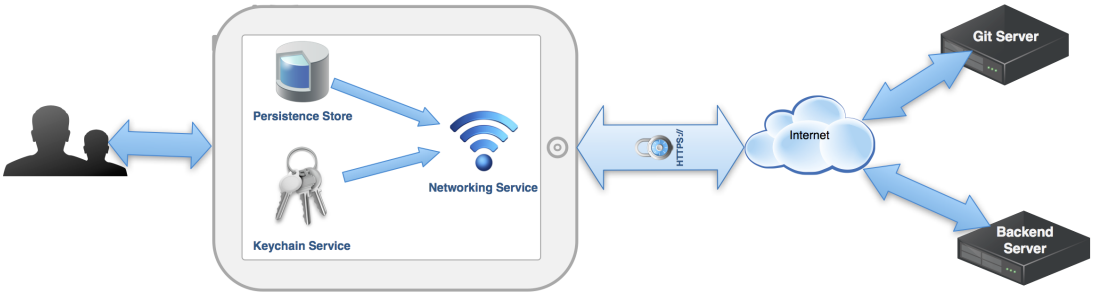

Figure 1: System architecture

assumption can be easily met since editors for graphical model usually offer an export-function to .png- or .jpg-files. Moreover, this generation step can be automated easily.

We have implemented our approach in form of a prototypical mobile app running on the operating system *iOS* (running on Apple products such as *iPad*, *MacBook*, etc.). The app supports project teams to work together on graphical artefacts. The app is designed as a review tool spezialized on graphical files and allows members of a project team to capture and communicate their ideas within and beyond the team.

3.1 Working with Git-Repositories

Our solution aims at extending the traditional functionalities of Git-systems [6]. There are two ways to work with Git-tools. One is to use a command line interface (CLI), where the user issues Git-commands to program the interactions with the Git-system in the form of successive lines of text. The other is the GUI-oriented frontend client with graphical interface for browsing the history and more. The GUI alternative provides a more comfortable and easy way to use Git for most users.

Our solution is a client application that connects to the version control system, pulls the resources of a repository and synchronise them with the user repository stored locally. Beside the effortless backup of all file history, our app paves the way to a more organized management of different versions of graphical files.

Beside the adaptation of Git's tradition workflows, our application offers a feedback workflow, which can be used to share annotations and feedbacks related to a single file without touching the repository on which this file is located. Because of the clear separation of primary artefacts (stored in the repository) and secondary artefacts such as comments (stored at another backend server, created by our app), the original repository is protected from being polluted with information that are valid and interesting only within a certain context.

3.2 System Architecture

Figure 1 shows the system architecture on a very coarse level: The user interacts with the app and must authenticate herself. The app communicates not only with a Git-server, which manages the central Git-repository, but also with a backend-server, whose task is to store all annotations and comments made by users on artefacts stored in the repository.

In Figure 1, the iPad represents the component structure of the application and emphasizes the three main *design factors*:

- *Security* - the user has to authenticate herself to the app as well as to Git-server and backend
- *Persistence Store* - information from external servers are cached locally at the machine on which the app runs
- *Connectivity* - the external servers (Git, backend) are queried according to the interactions of the user with the app

3.3 Features

The app offers in its current state the following features:

Multi-user support Using user's GitHub credentials, the app can be used by more than one user on one device.

Fetching repository information Once the user logged in, the app checks whether the user has saved her own repositories locally, so there is no need to connect to the server to fetch the repositories. Otherwise, the app calls the GitHub API to get a fresh copy of the user's repositories and list them.

Multiple user repositories Using the app, the user can work with multiple repositories at the same time. The app manages all credentials for accessing each of the used repositories.

Filtering files based on file-format The current version of the app can handle only two types of file format: *.jpg* and *.png*. Using the filter functionality, the user can define certain files within the repository as her scope of interest.

Showing all versions in timeline mode for a given file To see the file history, the user firstly selects the file of interest and can then navigate between the file's versions in slide mode.

Adding annotations The user can add comments and annotation to each version of a file. From the file version slide show, the user can choose the version she wants to annotate and by a long press gesture she is able to add an annotation at the place she wants. Annotations can be place at any position of the annotated file and are usually shared with other members of the team.

4 Demo Example

We want to illustrate our tool with a small example on how a UML model can evolve. The example presented here shows a design refactoring and is based on the refactoring *Encapsulate Classes with Factory* as described in the excellent textbook of Kerievsky [4].

Figure 2 shows how the user of our tool sees the evolution of a UML model. In the upper part, the user sees the first version of the design. In the lower part, the tool is shown after the user has moved by a swipe gesture to the next version.

The first design is a common situation in OO design, where commonalities among classes have been already centralized in form of a superclass (`AttributeDescriptor`) and this superclass also declares all access methods available to clients (here, indicated by class `Client`). Unfortunately, a client still has to create instances of the subclasses by itself and uses the constructors of every subclass. This is a design smell since all subclasses have to be public, what has severe drawbacks wrt. the maintability of these subclasses: if these classes were invisible outside the package, one could easily change or even delete them, as long the contracts to clients are kept.

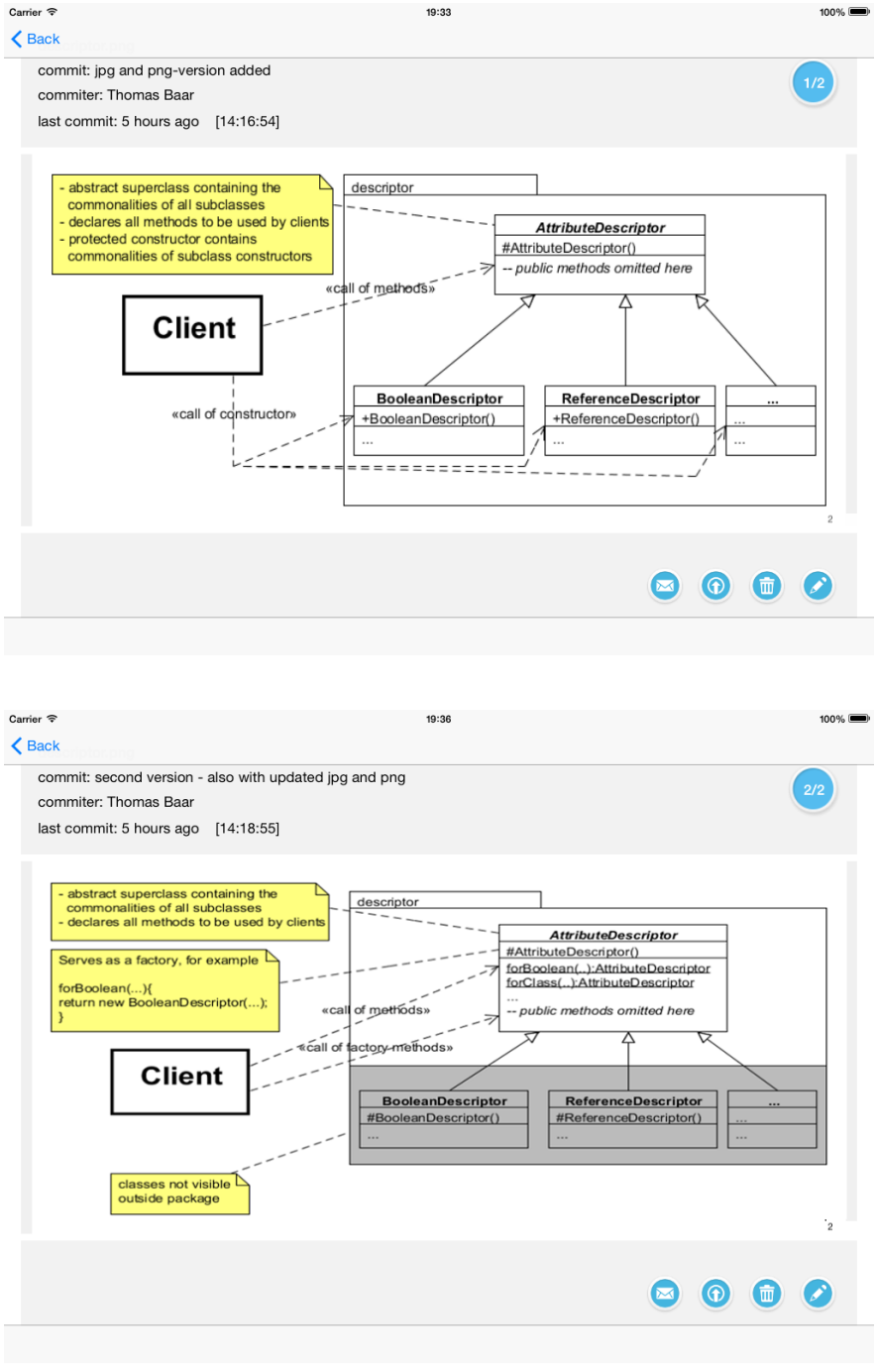

Figure 2: Screenshots from browsing two versions of a UML model

In the second design (after the refactoring) shown in the lower part, all subclasses and also their constructors are only visible within the same package. The class `AttributeDescriptor` has been extended with multiple factory methods (`forBoolean(..)`, `forClass(..)`, ...) and serves now as a factory for clients and can create instances of any subclass. The clients do not know the subclasses any longer.

5 Conclusion and Future Work

Christopher Alexander, who is admired by the software community for his pioneering work on architecture patterns [2], argues that a yard or a terrace built north to a house will be used only little: *'People are phototropic, biologically, so that it is often comfortable to place yourself where the light is.'* [1, p. 111] Likewise, a technology, which is poorly supported by tools, is adopted by users only slowly, because usage of this technology is generally avoided for the same reasons as people avoid to meet at wrongly placed terraces.

In the past, when applying the model-based approach on some real projects, we were frustrated about the fact, that our IDE (*Eclipse* in our case) was not able to display the version history of model files in a user-friendly way. For this reason, we have implemented our own tool that matches our expectations.

Currently, our tool has several limitations that we plan to overcome in the near future. Firstly, the tool should be available also for other platforms, not only for iOS. Secondly, we want to make the tool applicable for any Git-repository, not only for the ones hosted on GitHub.

Another severe limitation seems to be the applicability of our tool only to files that are stored in the graphical file formats *.jpg* or *.png*. Artefacts such as UML diagrams, however, are stored within a repository usually in a format, that is specific to the editor used to create the artefact. For example, the open-source tool *UMLet* stores UML-models in *.uxf*-files, but *.uxf*-files are — intentionally (!) — not understood by our tool. To circumvent this problem, the user has whenever committing a new version of a UML-diagram also to commit a *.jpg*- or *.png*-version of this diagram.

An important application area for our tool is teaching. Students should gain a deeper understanding for modeling by *seeing the evolution of models*. In many textbooks, only the final model describing a certain problem is presented.

We have started to use the tool in software engineering courses, in which UML is taught. In these courses, students are obliged to develop a complex UML model in an iterative way. If the artefacts produced by students are put under version control, then the results of each iteration can commented easily by the lecturer using our tool. We also plan to gain experience from applying the tool in a non-CS course, in which students produce many graphical artefacts, e.g. in fashion design.

References

[1] C. Alexander. *The Timeless Way of Building*. Number Bd. 8 in Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. Oxford University Press, 1979.

[2] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, August 1977.

[3] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of software maintenance, held with the 2008 IEEE International Conference on Software Maintenance, Beijing, China*, pages 48–57, 2008.

[4] J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley, 2004.

[5] LayerVault. Homepage. https://layervault.com.

[6] J. Loeliger and M. McCullough. *Version Control with Git.* O'Reilly, 2nd edition, 2012.

[7] Pixelapse. Homepage. https://www.pixelapse.com.

[8] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.