Bachelor Thesis

June 17, 2010

Commit 2.0 for Eclipse

Enriching commit comments with software visualization

Roberto Minelli

Abstract

Widely adopted versioning systems allow developers to write comments at commit time to describe the changes that they have performed. The current limitation is that such comments can only be textual. For example documenting a commit that involves multiple files is difficult: The developer needs to write a text that describes all the modifications. Non descriptive or blank comments drastically limit the documentation of code evolution.

We have ported a prototype for Smalltalk to a mainstream IDE (Eclipse). *Commit 2.0* tackles the problem of the limited support for documenting software changes at commit time, by enriching commit comments with software visualization and floating annotations. Our approach generates interactive visualizations of the changes where the user can insert annotations in the context to which they pertain.

Advisor
Prof. Dr. Michele Lanza
Assistant
Marco D'Ambros

Advisor's approval (Prof. Dr. Michele Lanza): Date:

Acknowledgements

Figure 1. Michele Lanza and Marco D'Ambros

First of all I would like to thank Prof. Dr. Michele Lanza, one of my favorite and most appreciated professors, in the capacity of advisor of my Bachelor project. Three years ago he started guiding me through the world of software systems, sharing some of his precious knowledge in the *Programming Fundamentals I* course and, last semester, in the *Software Atelier IV* course.

Many thanks to Marco D'Ambros, the assistant of the project and of the two courses mentioned above, that stimulates my enthusiasm for doing this work and for being always available, even from South Africa.

I would like to thank my family and friends for their unconditional support and patience, being so understanding during the busy times while I was studying, coding and writing.

Special thanks goes to Alessandro Trombini which has always been available to contain my outbursts in the past three years of Bachelor.

Roberto Minelli
June 17, 2010

Contents

List of Figures

Chapter 1

Introduction

A number of software projects use versioning systems, also known as Software Configuration Management tools (SCM), such as SVN[1] and CVS[2], to handle code evolution. Developers use SCMs as means of synchronization among them and to document changes through commit comments. Current version control tools do not store all the information generated by developers [2].

The support for documenting software changes offered by versioning systems is limited. The only tool that a developer has at commit time to document changes is text. It is often left blank due to lack of time or resources[3]. In addition, commits involving multiple files are difficult to document since this text cannot be related to a specific entity or inserted in a specific context. Sometimes when we browse the history of a software stored in a repository we see comments like "*bugabugabuga*" or "*changes*" or "*arg*"[4].

To address this problem researchers proposed *Commit 2.0* that enriches commit comments with software visualization [1]. They provide a prototype implementation for the Pharo IDE[5] and we ported it to Eclipse. Our tool automatically produces visualization of the changes at different granularity levels: Such views provide a context which makes documentation easier. In general, Software Visualization provides a support to have a comprehensible representation of the structure of and the distribution of efforts among a software system. *Commit 2.0* also supports floating annotations in the visualizations. The user can insert the annotations in the views relating them to their accurate context.

Our tool enriches the documentation process with visualizations. Versioning systems do not support views as comments. We adopted a Posterous[6] blog to store the visualizations. Posterous is a free service that creates instantaneous blog posts upon receiving an email with any number of attachments.

[1]See `http://svncorp.org`
[2]See `http://www.nongnu.org/cvs/`
[3]In the Eclipse project 20% of commit comments is left blank
[4]Real commit comments committed in the Vuze project (5300 out of >13000 are left blank) - `http://www.vuze.com`
[5]See `http://www.pharo-project.org/home`
[6]See `http://posterous.com/`

1.1 Goals

The main goal of the project is to enrich Eclipse to enhance the support given to developers to document changes at commit time with software visualization. The plug-in generates interactive views that can be enriched with floating annotations (like sticky-notes) that improve traditional commit comments.

The main goal is divided in sub-goals:

1. Creation of a model to recognize the structural differences, and not just the textual ones, between two versions of a project.
2. Visualization of the changes in an exhaustive and intuitive way.
   (a) The user has to interact with the visualizations.
   (b) The user can annotate the views.
   (c) The user can upload the visualizations to Posterous.
3. Integration from the IDE to an Eclipse Plug-in.

Figure 1.1 shows the workflow of a typical usage of *Commit 2.0*. A developer codes in Eclipse (1). At the end of the session he invokes *Commit 2.0* (2): the plug-in generates a visualization (3) that the user can edit and enrich with annotations (4). In the end the plug-in publishes the visualizations on the blog (5) and versions the source code in the SCM repository (6).

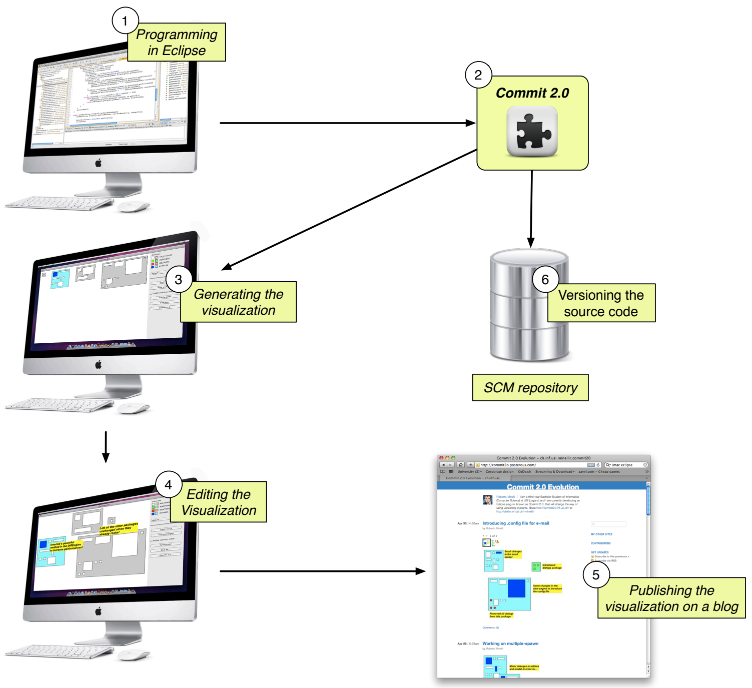

1
Programming in Eclipse
2
Commit 2.0
3
Generating the visualization
6
Versioning the source code
SCM repository
4
Editing the Visualization
5
Publishing the visualization on a blog

Figure 1.1. An overview of Commit 2.0 usage

1.2 Structure of the document

The remainder of this document is organized as follow: We give an overview of the work related to *Commit 2.0* (Chapter 2) and we describe in detail the plug-in itself (Chapter 3). Then we draw the conclusions (Chapter 4) and we introduce possible extensions of it (Section 4.2). The final part of the document (Appendix A) discusses some implementation details.

Chapter 2

Related Work

Commit 2.0 is an approach to enhance the support given to developers, at commit time, to document changes by means of software visualizations. It is related to (1) tools that visualize versioning system data and (2) enhancement of versioning systems and awareness of changes.

2.1 Visualizing Versioning System Data

CVSViewer3D is a tool [3] which extracts, processes and displays information from CVS repositories. The tool relies on existing front-end visualization software (SourceViewer3D). The user can define multiple views of the change history data and at different granularity levels (e.g., file, line of text, method, class).

Chronia uses the mapping between the changes and the author identifiers from CVS log-files to define a measurement of code ownership. The tool presents the Ownership Map visualization [4] that helps understanding when and how different developers interacted in which way and in which part of the system.

As software evolves over time, the identification of expertise becomes an important problem. Alonso *et al.* introduced a method to derive the expertise of the committers from the classification of the source code tree as a path [5]. They also presented a visualization that helps to further explore the repository via committers and categories. They implemented a prototype and validated the approach using the Apache HTTP Web Server project.

In [6] Taylor and Munro combined visualization and animation to study the evolution of a CVS repository. The technique, called Revision Towers, uses colored bars of different thickness and height to represent the size, changes and authors of a piece of code. These bars are animated to show the evolution of the software repository.

Rysselberghe and Demeyer [7] demonstrated how a simple visualization allows researchers to recognize relevant changes. The authors validated the approach on the change history of Tomcat to identify unstable components, coherent entities, design and architectural evolution, and fluctuations in team productivity.

Voinea *et al.* proposed an open framework, *CVSgrab*, for visual data mining of CVS repositories [8]. They focused on three aspects: data extraction, analysis and visualization. The tool supports analysis and interactive visualization of software repositories. The user can do querying and filtering and customize the views through a large set of metrics extracted from the CVS data.

Hattori *et al.* redefined the concept of code ownership [9], by enriching CVS data with real-time interaction information extracted from the IDE. They proposed a tool, called *Syde*, that records every change by every developer in multi-developer projects. The authors presented a visualization of the refined ownership.

There is a substantial difference between our approach and the approaches illustrated above. *Commit 2.0* visualizes the changes at commit time enhancing the support for their documentation while the mentioned approaches visualize the data a posteriori to support retrospective analysis.

2.2 Enhancing Versioning and Awareness

In [10] Schneider *et al.* discussed the benefits of analyzing local interaction histories. They argued that developer interaction with the local copy is valuable information when mining software repositories. The authors provided a prototype implementation to capture and analyze these local interactions.

Robbes *et al.* implemented a system to record fine-grained changes in the source code in real-time, instead of reconstructing them from coarse-grained, file-based versioning system archives [12]. This approach was used in a variety of applications [11] that improve versioning systems by refining the code model, while *Commit 2.0* enriches the documentation of code changes with visualization.

Current SCMs are designed to isolate developers from each other. Sarma *et al.* argued that such isolation is both good and bad. With Palantír [13] they wanted to overcome the bad isolation, while retaining the good one. Palantír is a workspace awareness tool that enriches configuration management systems by informing a developer of what other developers change which artifact. It calculates a simple measure of severity of the changes and visualizes the information in a non-obtrusive fashion.

In [14] Lanza *et al.* enhanced the approach proposed by Robbes recording source code changes *as they happen* and broadcasting them to other developers in the team so that they are aware of possibly existing conflicts before committing the code. The authors also proposed three lightweight visualizations to broadcast real time development information to developers.

Knodel *et al.* [15] propose a tool that continuously monitors changes made by developers and provides feedback to them in case structural violations are detected. They validated the approach with six component development teams and they argued that constructive compliance checking helps to reduce the structural violations inserted during the development by 60%.

While the goal of the approaches mentioned above is to provide awareness of changes, the goal of *Commit 2.0* is to document the changes *as they are committed* by means of software visualizations.

Chapter 3

Commit 2.0

In this chapter we describe our approach in detail and we discuss the benefits of visualization over text (Section 3.2.3) as well as pros and limitations of our approach (Section 3.2.3).

3.1 Commit 2.0 in a nutshell

Commit 2.0 is an IDE enhancement which enriches commit comments with visualization, providing a context to changes and a better means to communicate. It is built on top of the Eclipse IDE as shown in Figure 3.1, and does not change the standard commit mechanism. In the current implementation it only supports Subversion (SVN)[1] repositories.

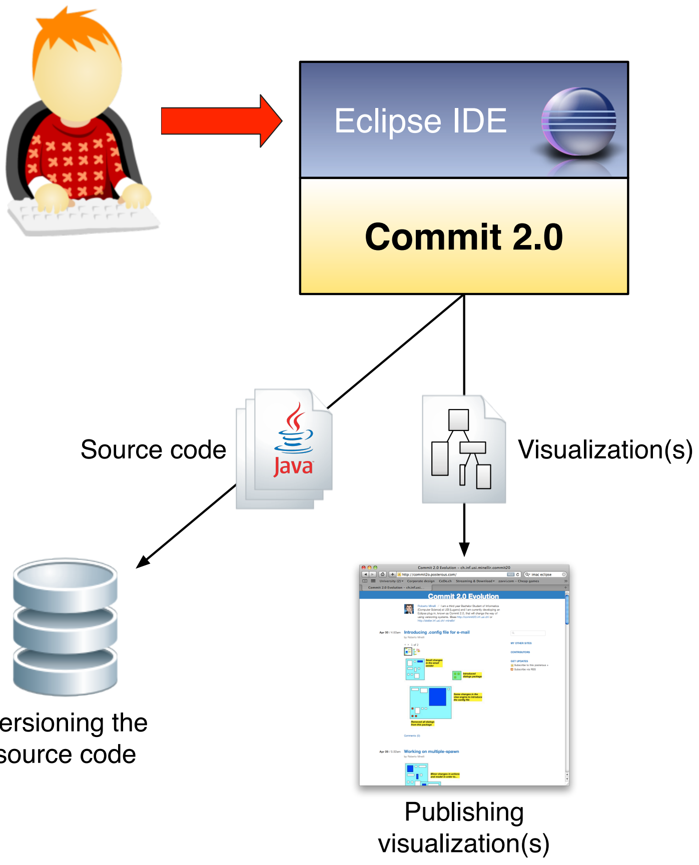


Figure 3.1. An overview of Commit 2.0

[1]See `http://svncorp.org/`

3.2 Visualizations

The aim of our tool is to enrich commit comments with software visualization. In this section we describe the principles behind the visualizations and we give some examples.

3.2.1 Visualization Principles

Our tool provides visualizations at two different granularity levels:

- *Coarse-grained*, concerning the overall structure of the project.
- *Fine-grained*, concerning the inner structure of the packages composing the project.

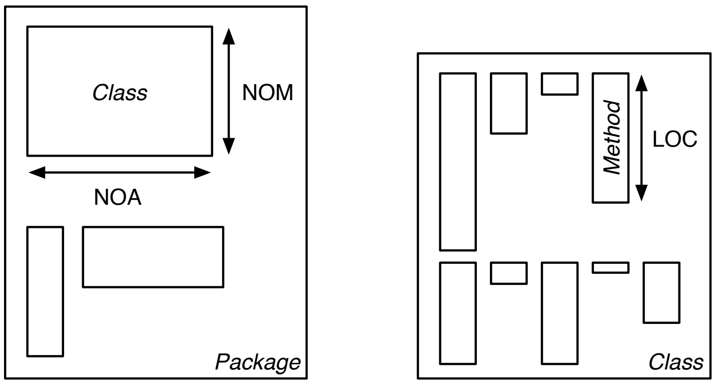


Figure 3.2. Principles of the Coarse- and Fine-grained visualizations

The sizes of the visualizations are mapped to software metrics. Software metrics measure properties of a software system by mapping them to numbers. In both visualizations we distinguish containers, rectangles used to depict entities (packages or classes), and elements, rectangles inside them to depict their content (classes or methods).

The *Coarse-grained* view (Figure 3.2 - left) shows the entire project. A container at this granularity level represents a package, whereas the elements represent classes. The height of a class is proportional to its number of methods (NOM) and the width to its number of attributes (NOA).

In the *Fine-grained* view (Figure 3.2 - right) containers represent classes where the elements represent methods. The width of the methods is fixed while the height is proportional to its number of lines of code (LOC).

All visualizations are generated using a logarithmic scale: This means that, for example, a rectangle (in the coarse-grained view) that has double the height of another one has 10 times more NOM. The metrics can cover a large range of values that might deform the views making most of it unreadable. We adopted a logarithmic scale to reduces this values to a more manageable range.

The generated views respect a color schema where red represents deletion (the corresponding entities have been removed), green represents addition, blue modification and white represents no change. If an entity is modified the container has an indirect change, represented in cyan. In general we use lighter colors for the containers. Gray is the color used for unchanged containers.

3.2.2 Examples

In this section we provide some examples of the principles presented in Section 3.2.1.

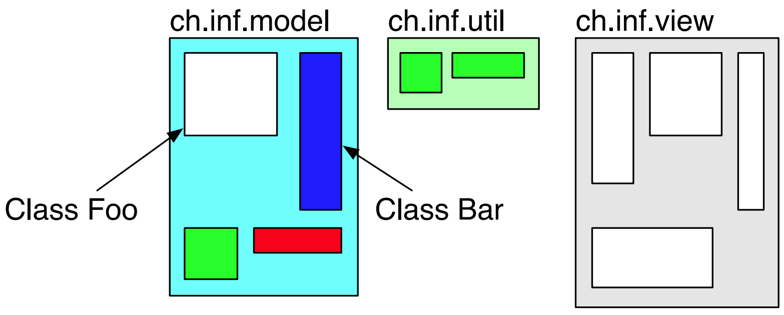


Figure 3.3. A coarse-grained example visualization

Figure 3.3 is an example of a coarse-grained visualization of a system composed by three packages. In the package *model* we can make some inferences about classes Foo and Barfrom from the sizes of the classes. The former has 10 times more attributes than Class Bar while the latter has 10 times more methods than the Class Foo. These inferences are made possible by the logarithmic mapping between the sizes of the elements and the software metrics.

The colors used for the visualizations help to detect what kind of changes happened. Figure 3.3 shows that the *util* Package has been added in this revision and that the *view* Package is unchanged. The *model* Package has indirect changes since some classes are modified (deleted or added) and it is depicted with color cyan.

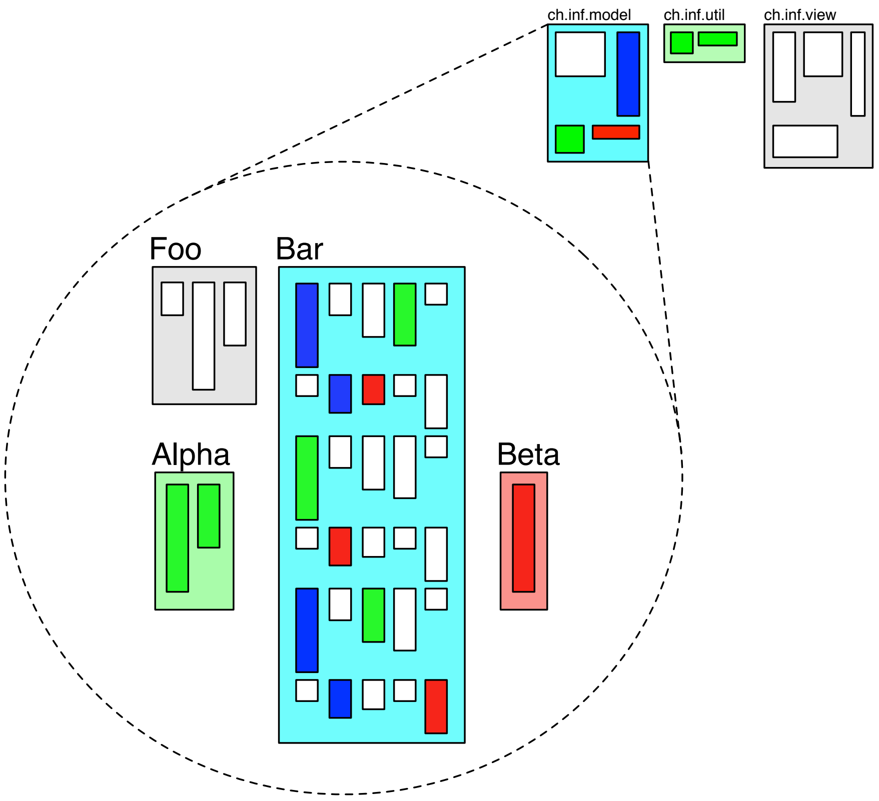


Figure 3.4. A fine-grained example visualization spawns from the model package of Figure 3.3

We provided fine-grained visualizations to spot changes more in detail. The user can spawn the contents of the entities from coarse-grained views. Figure 3.4 shows a spawn of the contents of the *model* package from the visualization in Figure 3.3. This shows that class Bar, with 30 methods, has 10 times more methods than class Foo, having 3 methods, as anticipated before. With the aid of the color schema we can detect the type of changes happened to the classes. Class Foo is unchanged (gray) while Bar contains indirect changes (cyan). Alpha is a class inserted in the current revision (green) while Beta is a deleted class (red).

3.2.3 Discussion

We think that our visualizations provide benefits over text providing a contribution in the context of software changes. Developers can have a complete view of the system (coarse-grained) to see where changes happened and obtain more details using fine-grained visualizations. This, together with annotations, can really push forward the process of documenting changes making developers' life easier and more productive. The interaction plays a fundamental role in the documentation: The user can decide the positioning of the entities and insert in the right context detailed annotations (described in detail in Section 3.4) to document the changes.

- Scalability:
  - Different granularities of the visualizations make the approach scalable.
- Benefits of our visualizations:
  - Are more descriptive than commit comments.
  - Provide a context for annotations.
  - Provide an overview of how the efforts are distributed among the project.
  - Allow to determine if changes happen in a crucial part of the system or not.
  - The color schema provides an intuitive way to spot the kind of changes happened.
- Limitations:
  - The blog used to store visualizations is not an optimal solution.
  - Visualizations are exported as images: There is no possibility to change them afterwards.

3.3 Structure of the tool

Figure 3.5 shows the main window of *Commit 2.0* that is composed of a space for the interactive visualizations (1) and a toolbar (2), featuring the following components:

(a) A package at coarse-grained view with (b) the contextual menu to spawn it.

(c) A class at fine-grained view.

(d) Some annotations.

(e) A preview of the selected color schema and a combo box to change it.

(f) Some controls to interact with the visualizations:

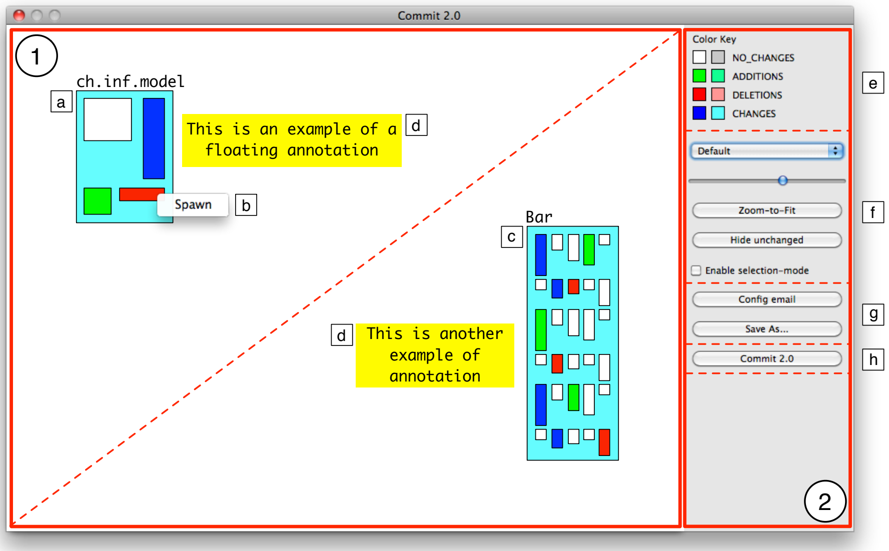


Figure 3.5. An overview of the structure of the tool

- Change color schema
- Adjust the zoom
- Hiding unchanged entities
- Toggle selection mode (to spawn more than one entity)

(g) Buttons to configure the email sender and export the visualizations on a file.

(h) A button to publish the visualizations and version the source code.

3.4 An example: MetricsExtractor

In this section we provide an application of the plug-in in a concrete situation: the implementation, and documentation, of the *MetricsExtractor* of the plug-in itself. It is the module responsible to extract, from the source code of the project, the software metrics used to scale visualizations.

We implement inside Eclipse the *MetricExtractor* class in the *util* package. We re-factor the related classes all around the system and we test our new implementation. We are ready to commit, and document, the changes. We invoke *Commit 2.0*: it automatically generates a coarse-grained visualization of the system, shown in Figure 3.6.

An user can re-arrange (drag & drop) the entities inside the canvas to suit his needs. We can remove the unchanged package, *dialogs*, from the visualization and focus our attention on the documentation of the 4 other packages. In this visualization we add some general annotations for coarse-grained changes: Figure 3.7 shows the result.

As for entities, also annotations can be drag & dropped around the canvas. Once we annotated the coarse-grained view, we found useful to document more in deep the changes in the *model* and in the *util* packages.

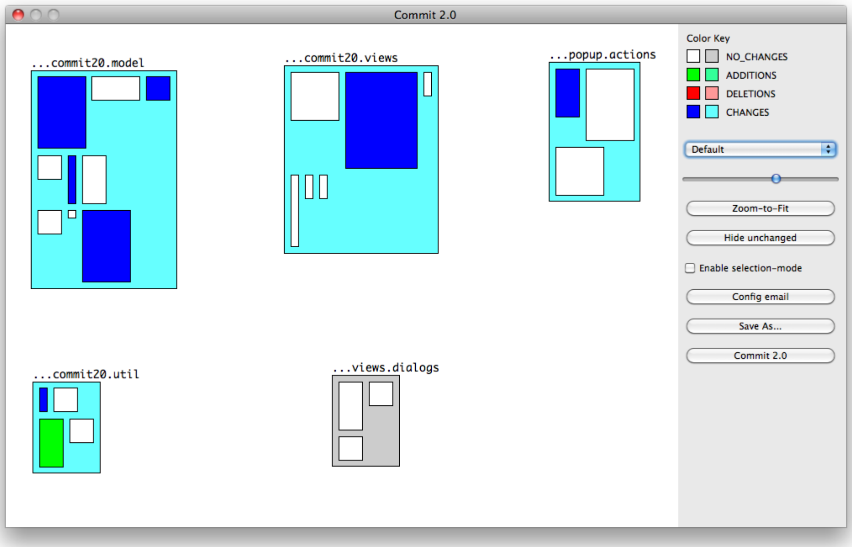
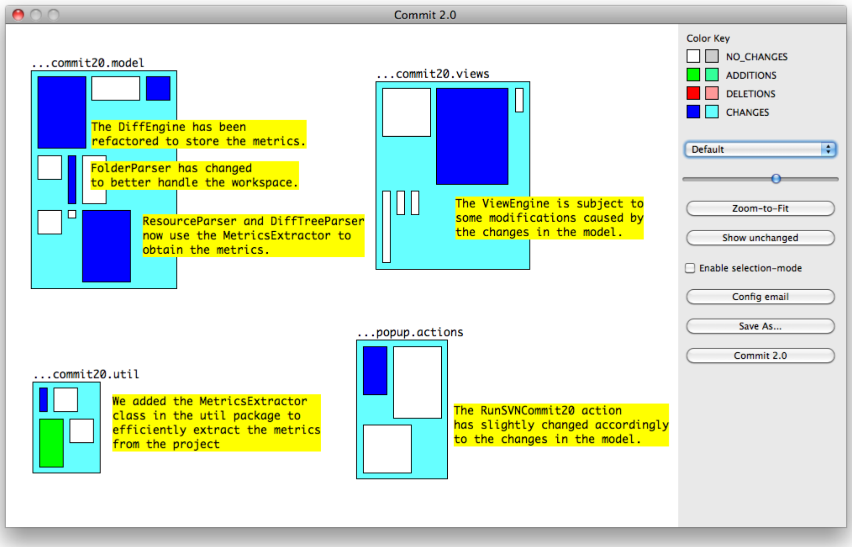


Figure 3.6. Coarse-grained view of the system at commit-time



The DiffEngine has been refactored to store the metrics.

FolderParser has changed to better handle the workspace.

ResourceParser and DiffTreeParser now use the MetricsExtractor to obtain the metrics.

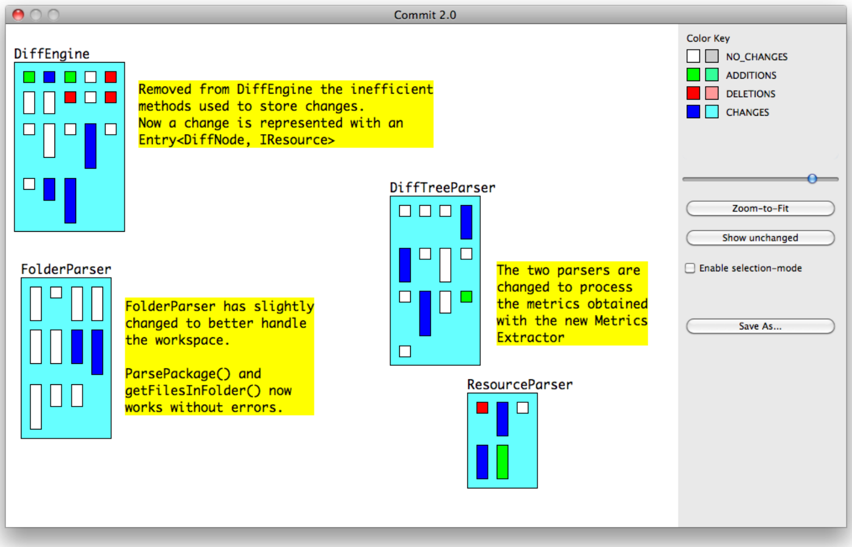
The ViewEngine is subject to some modifications caused by the changes in the model.

We added the MetricsExtractor class in the util package to efficiently extract the metrics from the project

The RunSVNCommit20 action has slightly changed accordingly to the changes in the model.

Figure 3.7. Coarse-grained view with annotations

We spawn the content of the *model*, by right-clicking on it. The plug-in generates a fine-grained view of this package where we can remove unchanged entities and concentrate our attention on the four changed classes. Also in this visualization we can add as many floating annotations as we want. Figure 3.8 shows the results while Figure 3.9 shows the same process applied also to the *util* package.



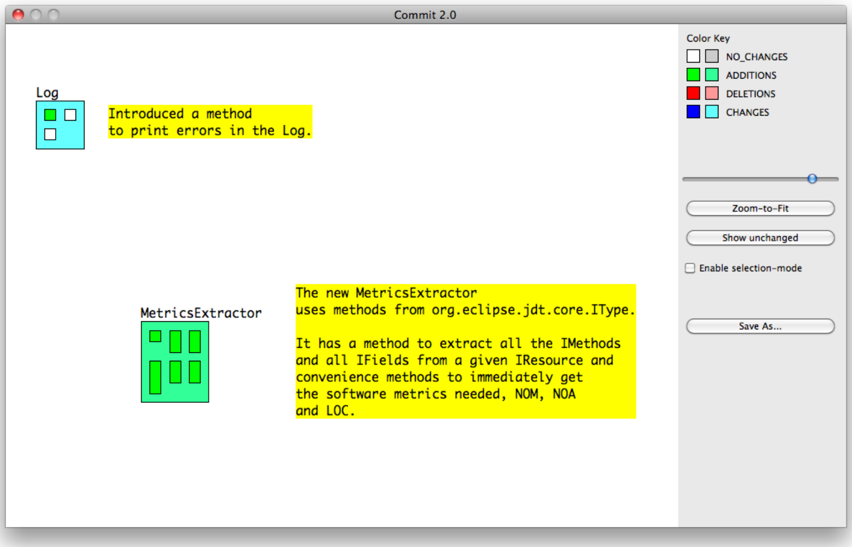
Removed from DiffEngine the inefficient methods used to store changes.
Now a change is represented with an Entry<DiffNode, IResource>

FolderParser has slightly changed to better handle the workspace.

ParsePackage() and getFilesInFolder() now works without errors.

The two parsers are changed to process the metrics obtained with the new Metrics Extractor

Figure 3.8. Fine-grained view of the model package with annotations



Introduced a method to print errors in the Log.

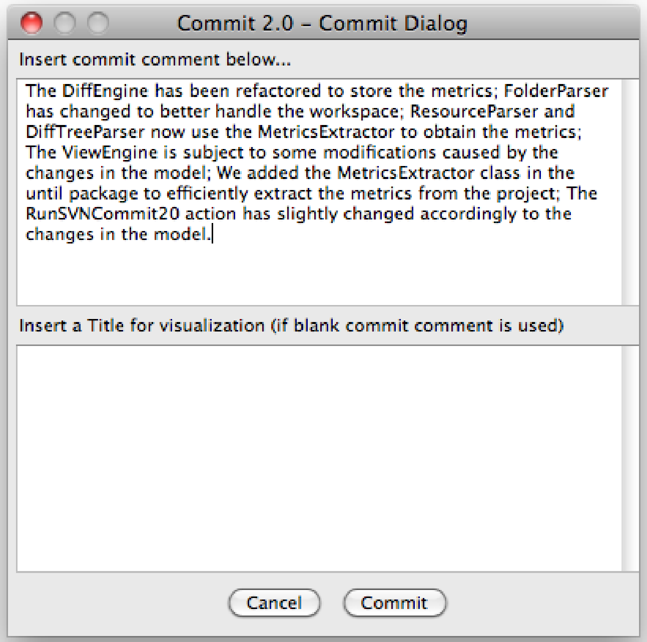
The new MetricsExtractor uses methods from org.eclipse.jdt.core.IType.

It has a method to extract all the IMethods and all IFields from a given IResource and convenience methods to immediately get the software metrics needed, NOM, NOA and LOC.

Figure 3.9. Fine-grained view of the utile package with annotations

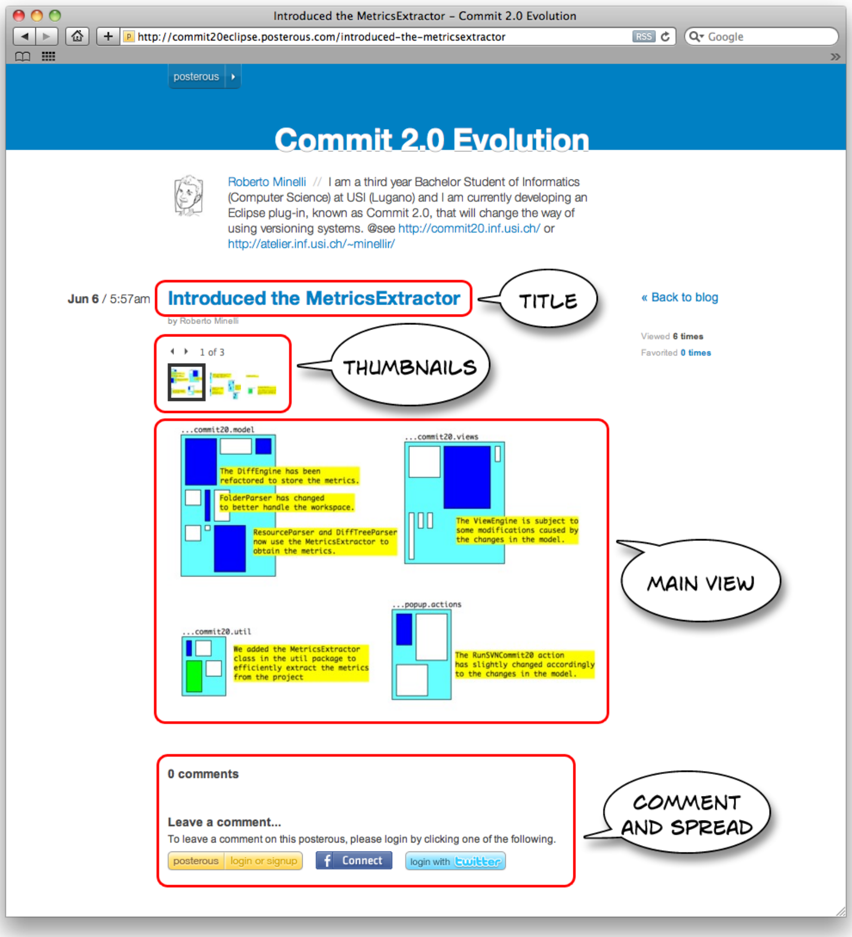
In the end we press the *Commit 2.0* button, situated on the coarse-grained window, to allow the plug-in to version the source code and publish the visualizations. The plug-in shows a window to enter the commit comment and the blog post title (Figure 3.10). A proposed commit comment is generated from the annotations in the coarse-grained view. The traditional commit mechanism is not altered and all the visualizations are published as a blog post as shown in Figure 3.11. At the end of each post there is the possibility to comment the post and to share it on different social networks (e.g. Facebook[2] or Twitter[3]) enhancing collaboration among the developer team.

[2]See `http://www.facebook.com/`
[3]See `http://twitter.com/`


Commit 2.0 – Commit Dialog
Insert commit comment below...
Insert a Title for visualization (if blank commit comment is used)
Cancel
Commit

The DiffEngine has been refactored to store the metrics; FolderParser has changed to better handle the workspace; ResourceParser and DiffTreeParser now use the MetricsExtractor to obtain the metrics; The ViewEngine is subject to some modifications caused by the changes in the model; We added the MetricsExtractor class in the until package to efficiently extract the metrics from the project; The RunSVNCommit20 action has slightly changed accordingly to the changes in the model.

Figure 3.10. The commit comment and blog post title prompt



Roberto Minelli // I am a third year Bachelor Student of Informatics (Computer Science) at USI (Lugano) and I am currently developing an Eclipse plug-in, known as Commit 2.0, that will change the way of using versioning systems. @see http://commit20.inf.usi.ch/ or http://atelier.inf.usi.ch/~minellir/

Jun 6 / 5:57am Introduced the MetricsExtractor

by Roberto Minelli

The DiffEngine has been refactored to store the metrics.

FolderParser has changed to better handle the workspace.

ResourceParser and DiffTreeParser now use the MetricsExtractor to obtain the metrics.

The ViewEngine is subject to some modifications caused by the changes in the model.

We added the MetricsExtractor class in the util package to efficiently extract the metrics from the project

The RunSVNCommit20 action has slightly changed accordingly to the changes in the model.

0 comments

Leave a comment...

To leave a comment on this posterous, please login by clicking one of the following.

Figure 3.11. A Posterous blog post published by *Commit 2.0*

3.5 Discussion

Our approach feeds benefits to the traditional documentation process but has some limitations.

Benefits

- Scalable: We claim that different granularities of the visualizations (coarse- and fine-grained views) allow the visualization of changes in projects with more than a hundred classes, even if we tested it with smaller projects.
- User-friendly: *Commit 2.0* is very intuitive to use.
- Functional: We used it during the development and it fulfills its purposes and functions.
- Unaltered basis: The system does not alter the traditional commit process to version the source code.

Limitations

- Time consuming: It requires developers to spend more time in the documentation process with respect to the traditional form of documentation (commit comments). We argue that this time investment is a valuable contribution in favor of a good documentation of the changes. Our approach facilitates synchronization and collaboration among developers making them more productive.
- Pre-requisites: *Commit 2.0* requires that developers learn the views: They need to understand how colors are used in the visualizations and how software metrics map to the sizes of the entities.
- Blog: The use of a blog, not directly connected with the repository, is a limitation. At this stage of development a user cannot see a connection between a repository revision and a blog post.

3.6 Validation

The main validation approach adopted, as a proof of concept, was using the plug-in to monitor and document the changes of the plug-in itself during the development. The dedicated website[4] shows the evolution of *Commit 2.0*.

We plan to do a user study to see if developers care for such support and to see whether it scales for systems with more than a hundred classes. Moreover this study will test the effectiveness of documenting changes using *Commit 2.0*.

The plug-in is available also on the Eclipse Marketplace[5]. We hope to receive some feedbacks to see what the users find attractive of the approach and what they desire to change or improve.

[4]See `http://commit20eclipse.posterous.com/`
[5]See `http://marketplace.eclipse.org/content/commit-20`

Chapter 4

Conclusions

SCM tools are widely used in software projects. Using a repository to monitor code evolution and as a means of synchronization among developers has become a standard approach in the developing process. Repositories provide a limited support for documenting changes at commit time: a text (commit comment) that a developer writes and it is stored in the repository. Finding a descriptive piece of text for a commit of an entire coding session, possibly involving multiple files, is a non trivial task and often developers, for lack of time or resources, leave these comments blank.

We addressed this problem with *Commit 2.0*: An IDE enhancement that enriches commit comments with software visualization. We implemented a plug-in to port *Commit 2.0* from the Pharo IDE to the Eclipse IDE. It allows the developer to generate a coarse-grained visualization of the changes he/she performed. The developer can re-arrange the position of the entities and add floating annotations in the specific context in which they are needed. If necessary he/she can spawn to a finer granularity level to have a more precise view of the changes. The user can then version the source code as usual and publish all the annotated visualizations on a blog allowing other developers to spot recent changes with a detailed documentation by browsing it.

4.1 Contributions

The main contributions of this work can be summarized as:

- The porting of *Commit 2.0* for Eclipse, initially developed for the Pharo IDE[1].
- The implementation inside, and full integration with, a mainstream IDE.
- The introduction in Eclipse of a visual approach to support the documentation of software changes at commit time.
- The automatic generation of visual differences between two versions of a Java Project, Package or Class.
- The validation of our approach, as a proof of concept, on the plug-in itself.

[1]The overall work was more challenging in Java than in Smalltalk: The reflection property of Smalltalk (the process by which a computer program can observe and modify its own structure and behavior) supports Marco D'Ambros *et al.* [1] to easily detect differences between two versions of a project and implement the first prototype.

4.2 Future work

This version of the tool has some limitations that can be removed and allows a wide number of extensions.

- *Repository interface*: In the current implementation the plug-in supports only Subversion (SVN). In future versions we would like to introduce support for different versioning systems such as CVS and Git[2].
- *Interaction*: We want to give more freedom to the user editing the visualizations. We want to add support for relating annotations to entities. A possibility would be drawing arcs from a comment to one or more entities to which this comment is related and then use this relations to automatically propose to the user a more descriptive and exhaustive comment than the traditional commit comment he/she would write.
- *Versioning the visualizations*: All the visualizations that the user creates or spawns are exported in a blog, to have a visual track of the evolution of the software system. In following releases we want to version also the visualizations. We can create an ad-hoc folder in the root of the project containing the views and version this folder.
- *Enhancing versioning systems*: The long term goal would be enhancing versioning systems to support visualizations as comments. If versioning systems would natively support this feature we could remove the temporary blog to store visualizations and get rid of visualizations from the repository (as proposed above).

4.3 Epilogue

In this thesis we have ported a prototype for Smalltalk to a mainstream IDE (Eclipse) to reach a larger audience. *Commit 2.0* is an approach to tackle the problem of limited support for documenting changes at commit time, by enriching commit comments with software visualization.

There are many steps to take in this direction since *Commit 2.0* is open to a wide number of extensions, as discussed in Section 4.2.

We believe that in the following years this area will be a hot research topic and our *Commit 2.0* is making headway in the field.

[2]See `http://git-scm.com/`

Appendix A

Commit 2.0: Implementation

In this appendix we describe some implementation details. Section A.1 shows the procedure of selecting the resources to commit. Section A.2 describes the process of obtaining the remote files. The parsing procedure is described in Section A.3 while Section A.4 shows how the plug-in computes the structural differences between two projects. Section A.5 illustrates some details of the visualizations while Section A.6 describes how to interact with them. Section A.7 shows how the visualizations are published and Section A.8 describes how the plug-in versions the source code.

A.1 Selecting resources

Eclipse's plug-ins operating on the Workspace have to deal with *org.eclipse.core.resources*. We can do an analogy between the Eclipse Workspace and the File System. In the Workspace there are files and folders organized in the same way of a File System. There can be subdirectories and hidden files.
The classes in the package mentioned above are used to programmatically manipulate the files in the Eclipse Workspace as a user does in a traditional file system: copy, paste, move, rename, get the contents and so on. In addition to files and folders in the workspace there are also projects (special kind of folders) and a resource called workspace root, analogous to the root directory of your File System. *Commit 2.0* uses all of them to correctly handle the resources the user wants to process.

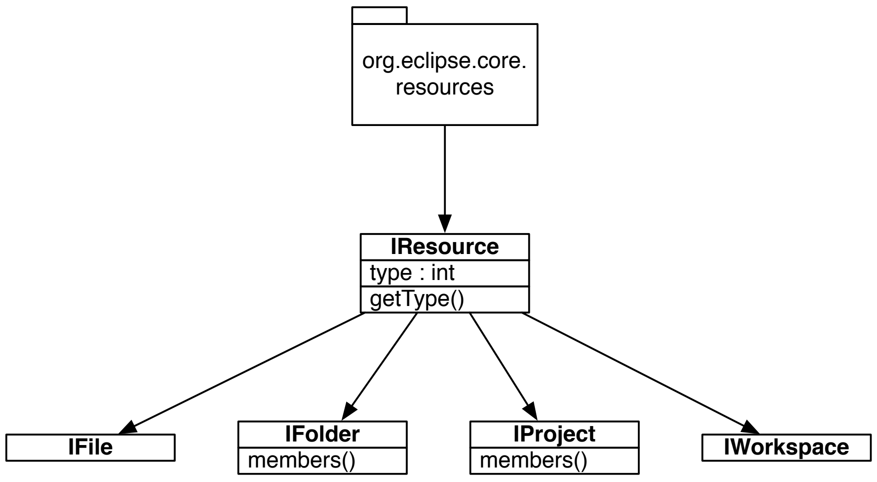


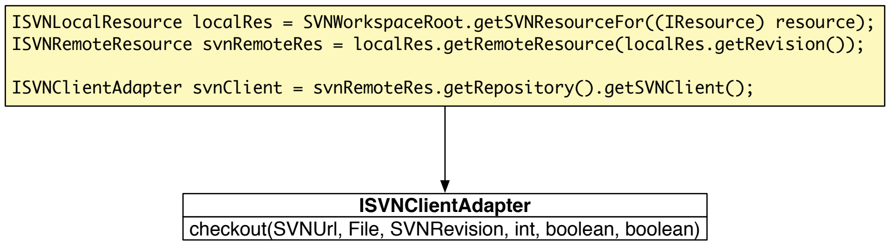
Figure A.1. An overview of the workspace resources

Figure A.1 shows the architecture of the package used by Eclipse to handle the Workspace files. *Commit 2.0* verifies which type of resource the user selects. It is able to operate on single packages

(IFolder) or entire projects (IProject). Once this is determined, the plug-in acts in a dedicated manner for different kind of choices generating either coarse- or fine-grained views.

A.2 Getting remote files

To get the remote files the plug-in executes a *SVN Checkout operation* by means of a *ISVNClientAdapter* provided by *org.tigris.subversion*. Figure A.2 shows briefly how to get one instance of it and which method it is used to perform the needed operation.



```
ISVNLocalResource localRes = SVNWorkspaceRoot.getSVNResourceFor((IResource) resource);
ISVNRemoteResource svnRemoteRes = localRes.getRemoteResource(localRes.getRevision());

ISVNClientAdapter svnClient = svnRemoteRes.getRepository().getSVNClient();
```

Figure A.2. An overview of the SVNClientAdapter

The parameters used in the checkout methods are, in order, the SVNUrl (1) of the resource we want to checkout, obtained invoking the *getUrl()* method on a *ISVNRemoteResource*, the File (2) in which the remote resources will be checked out, the revision (3) to which we want to check it out, the depth (4), as an integer (i.e. *DEPTH_INFINITY* constant from *ISVNCoreConstants*) and two booleans for keeping the locks (5) and for forcing the checkout (6).

To optimize the performances, since checking out big projects may require long time, if the user selects directly only a package this specific package will be checked out. If the user selects a project, instead, the entire project will be checked out. Even in this situation there is an optimization: If the user wants to spawn one or more entities the communication with the repository does not take place anymore. The plug-in obtains the differences from the project already checked out.

A.3 Parsing procedure

Figure A.3 represents the overall parsing process together with some relevant methods defined in the *FolderParser* class: One of the classes responsible for the parsing.

The input that comes from the selection described in Section A.1 is processed and modeled by the FolderParser class. This class first determines whether it has to work on a project or on a package, and if it is the second case whether the user needs a coarse- or fine-grained visualization since the parsing process is different. The two parameters of the methods on top of Figure A.3 are, respectively, the left (local) and the right (remote) terms of comparison. The FolderParser class defines some convenience methods to manipulate and browse the Workspace. Most of them uses the *members()* method, introduced in Figure A.1, to get the contents of a directory (or project).

The parsing procedure produces an object of type *DiffEngine* for each parsed package. A *DiffEngine* object has the structure illustrated in Figure A.4, having a name and some fields to store the modified resources within the package.

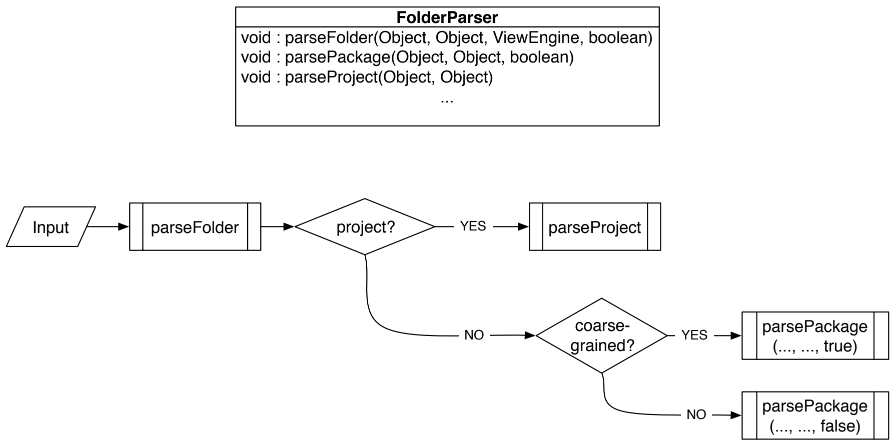

FolderParser
void : parseFolder(Object, Object, ViewEngine, boolean)
void : parsePackage(Object, Object, boolean)
void : parseProject(Object, Object)
...
Input
parseFolder
project?
YES
parseProject
NO
coarse-grained?
YES
parsePackage (..., ..., true)
NO
parsePackage (..., ..., false)

Figure A.3. An overview of the parsing procedure

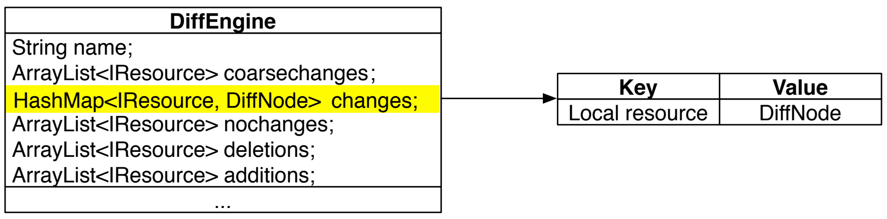

DiffEngine
String name;
ArrayList<IResource> coarsechanges;
HashMap<IResource, DiffNode> changes;
ArrayList<IResource> nochanges;
ArrayList<IResource> deletions;
ArrayList<IResource> additions;
...
Key
Value
Local resource
DiffNode

Figure A.4. An overview of a DiffEngine object

Figure A.4 highlights the unique relation used to store fine-grained changes. At this granularity level are needed both the *IResource*, to extract the metrics later on, and the DiffNode (discussed more in depth in Section A.4) used to store the changes of the corresponding *IResource* (a class).

A.4 Differencing engine

The differencing engine relies on *org.eclipse.compare*[1]. The package provides support for performing structural and textual compare operations on arbitrary data and displaying the results. It uses a *org.eclipse.compare.structuremergeviewer.Differencer*, a generic two-way or three-way differencing engine used by calling one of the *findDifferences()* methods and passing in the objects to compare. The initial rough approach, to understand how this engine works, was visualizing the *DiffNode* using a *StructureDiffViewer*[2] that allows to easily browse the *DiffNode* created after the comparison.

Figure A.5 shows the result of the expansion of a *DiffNode* generated from the comparison of two classes. From this analysis we could determine how the differencing engine constructs this trees and I how we can efficiently extract the changes from them. They are parsed from the plug-in in order to construct the *DiffEngines* described in Section A.3.

[1]See `http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/compare/package-summary.html`

[2]A diff tree viewer that can be configured with a *IStructureCreator* to retrieve a hierarchical structure from the input object (an *ICompareInput*) and perform a two-way or three-way comparison on it.

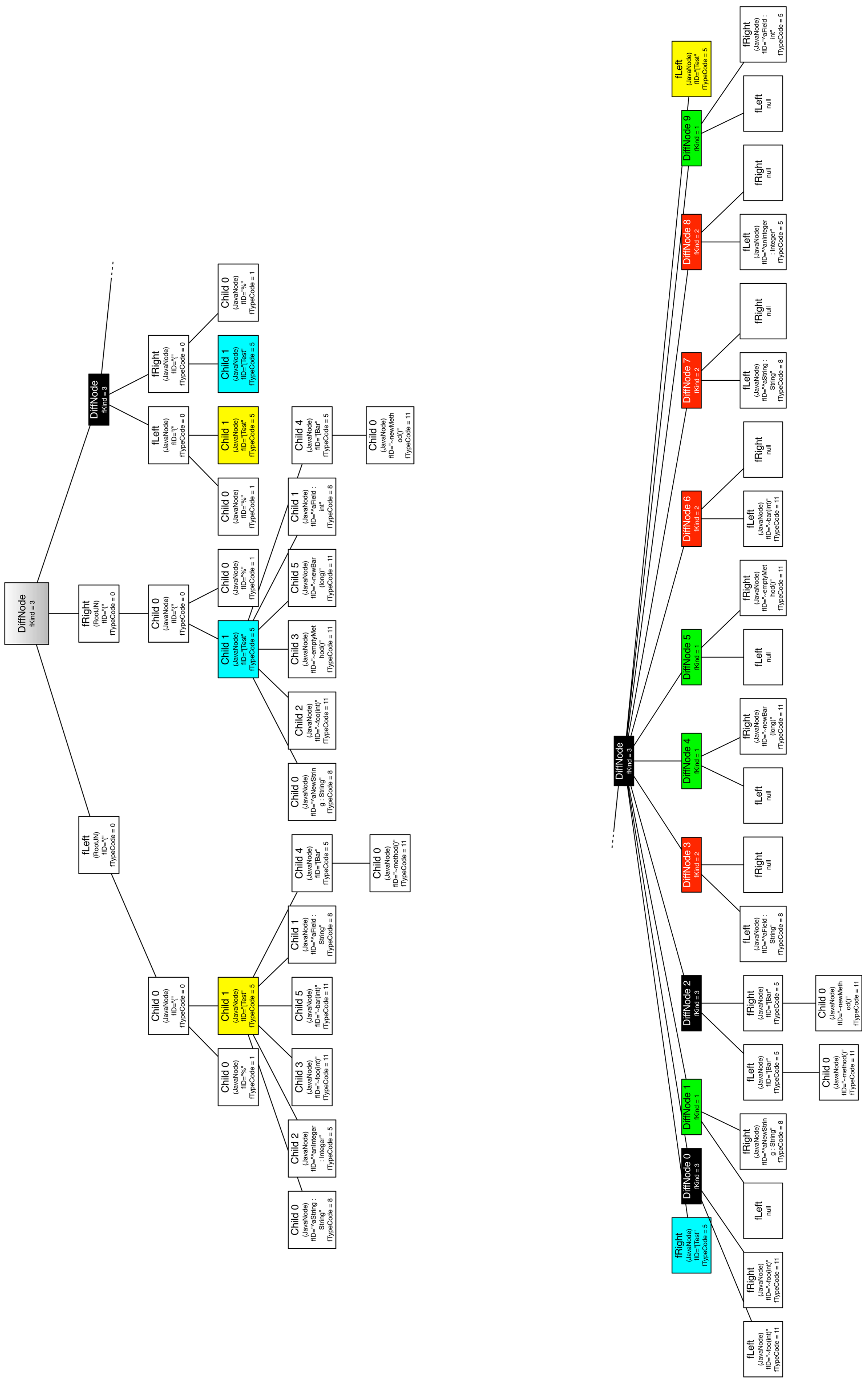


Figure A.5. A DiffNode in details

A.5 Generating visualization

The plug-in allows the user to generate visualizations at two different granularity levels: Coarse- and fine-grained. Coarse-grained views show packages as containers and classes inside them. The height of the classes is proportional to the number of methods (NOM) and the width to the number of attributes (NOA). Fine-grained views show classes as containers and methods inside them. The width of the methods is fixed while the height is proportional to the number of lines of code (LOC). For more information on visualization principles see Section 3.2.1.

If the user selects a Java Project the plug-in automatically generates a coarse-grained view of the system. To do that the *ViewEngine* receives all the information computed from the core of the application and produces a visualization powered by the *Standard Widget Toolkit* (SWT)[3]. To do so it parses one package at the time to determine which classes are contained in it and computing the needed software metrics. In particular, for the coarse-grained views, number of attributes and the number of methods. These metrics are obtained using methods from the *org.eclipse.jdt.core.IType* package.

Analogously, for the fine-grained views, the model is passed to the *ViewEngine* that parses all the classes within a given package to compute the lines of code of each method and represent the entities. A fine-grained view is automatically generated if the user selects a package instead of a project.

A.6 Interacting with the visualization

The visualizations generated from the plug-in are interactive. The user can move around the entities by simple drag and drop gestures, zoom in and out using the scroll wheel (or moving the slider on the GUI, see Figure 3.5), change the color schema and show or hide the unchanged entities that are not relevant to document the changes.

The visualizations can be enriched with floating annotations. These annotations scale, together with the entities, according to the zoom level and can be edited or deleted.

We provided to the user some zooming functionalities. He/she can select the best level of zoom for his layout with a *zoom-to-fit* functionality. A *best-fit* function that suggests the user the best position for the entities together with the right quantity of zoom will be added in future releases.

[3]See `http://www.eclipse.org/swt/`

A.7 Publishing the visualization

Once the visualizations are generated and annotated the plug-in publishes them on a blog in order to have them in a convenient and easily accessible place for all developers of the project. For this release a blog is used since currently no SCM support visualizations for comments. We used Posterous, a service that creates blog posts upon receiving emails. For this purpose we implemented an e-mail sender using the *JavaMail (javax.mail)* library. Figure A.6 illustrates an overview of the publishing procedure.

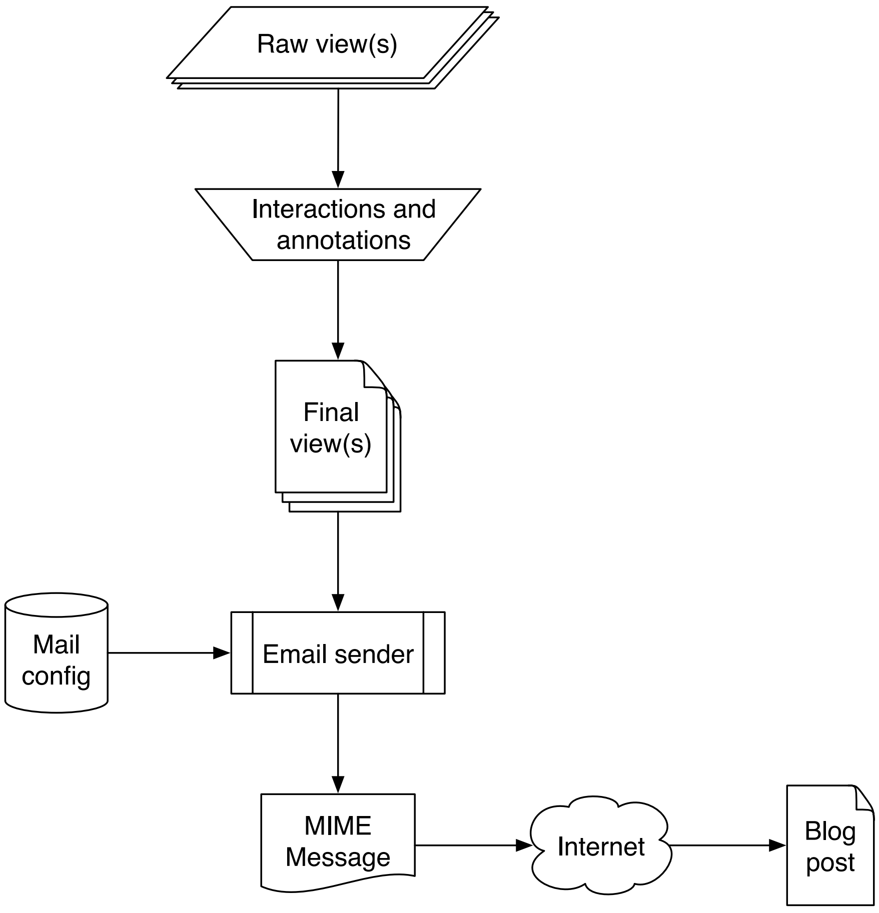


Figure A.6. An overview of the publishing procedure

A.8 Versioning the source code

The whole *Commit 2.0* procedure does not change the standard commit mechanism. While the visualizations are published on the blog, the source code is versioned on the SCM repository. To accomplish this task we implemented the commit operation on top of *Subclipse* [4], a well known Eclipse plug-in, to support Subversion in the Eclipse IDE. We plan to remove the dependency with this plug-in.

[4]See `http://subclipse.tigris.org/`

Bibliography

[1] Marco D'Ambros, Michele Lanza, Romain Robbes, *Commit 2.0*. In Proceedings of Web2SE 2010 (1st International Workshop on Web 2.0 for Software Engineering), pp. 14 - 19, IEEE CS Press, 2010.

[2] Romain Robbes, Michele Lanza, *A Change-based Approach to Software Evolution*. In Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 166, pp. 93 - 109, January 2007. Elsevier Science Direct, 2007.

[3] Xie, Xinrong and Poshyvanyk, Denys and Marcus, Andrian, *Visualization of CVS Repository Information*, In Proceedings of WCRE 2006, pp. 231–242, IEEE CS, 2006.

[4] Tudor Gîrba and Adrian Kuhn and Mauricio Seeberger and Stéphane Ducasse, *How Developers Drive Software Evolution*, Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 113-122, IEEE Computer Society Press, 2005.

[5] Alonso, Omar and Devanbu, Premkumar T. and Gertz, Michael, *Expertise identification and visualization from CVS*, Proceedings of the 2008 international working conference on Mining software repositories (MSR 2008), 2008, pp. 125-128, ACM.

[6] Taylor and Munro, *Revision towers*, In Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis, 2002, pp. 43-50, IEEE Computer Society.

[7] Los Alamitos CA, Van Rysselberghe, Filip and Serge Demeyer, *Studying Software Evolution Information By Visualizing the Change History*, Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04), pp. 328-337, IEEE Computer Society Press, 2004.

[8] Lucian Voinea and Alexandru Telea, *An open framework for CVS repository querying, analysis and visualization*, Proceedings of the 2006 international workshop on Mining software repositories (MSR 2006), 2006, pp. 33-39, ACM.

[9] Lile Hattori and Michele Lanza, *Mining the History of Synchronous Changes to Refine Code Ownership*, Proceedings of MSR 2009 (6th IEEE Working Conference on Mining Software Repositories), pp. 141-150, IEEE CS Press, 2009.

[10] Kevin Schneider and Carl Gutwin and Reagan Penner and David Paquette, *Mining a Software Developer's Local Interaction History*, Proceedings of the First International Workshop on Mining Software Repositories (MSR 2004), 2004.

[11] Romain Robbes, University of Lugano, Switzerland, *Of Change and Software*, 2008.

[12] Romain Robbes and Michele Lanza and Mircea Lungu, *An Approach to Software Evolution Based on Semantic Change*, Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering), pp. 27-41, 2007.

[13] Anita Sarma and Zahra Noroozi and André van der Hoek, *Palantír: Raising Awareness among Configuration Management Workspaces*, Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pp. 444-454, IEEE Computer Society, 2003.

[14] Michele Lanza, Lile Hattori and Anja Guzzi, *Supporting Collaboration Awareness with Real-time Visualization of Development Activity*, Proceedings of CSMR 2010 (14th IEEE European Conference on Software Maintenance and Reengineering), IEEE CS Press, 2010.

[15] Jens Knodel and Dirk Muthig and Dominik Rost, *Constructive Architecture Compliance Checking – An Experiment on Support by Live Feedback*, Proceedings of ICSM 2008, pp. 287-296, 2008.