TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Interactive Visualization of CVS Data

By
M.F.T. Ramachers

Supervisors:

Kees Huizing (TU/e)
Huub van de Wetering (TU/e)

Eindhoven, June 2006

Contents

Chapter 1

Introduction

This is the final report to round off the Master Project part (2J042) of the study of Computer Science (Technische Informatica) at the University of Technology Eindhoven (TU/e). All work for this project was done at the department of Mathematics and Computer Science in the research group Visualization during the period February 2005 - June 2006.
Subject of the master project is interactive visualization of CVS data in an Integrated Development Environment (IDE). During the project methods were developed for visualizing (project) data in the NetBeans IDE for the Java programming language. The purpose of the visualization is to help programmers and project managers in their work on large software projects in which a large number of programmers work together on a large number of files.
Chapter 2 describes the problem to be solved during this project in detail. In the chapter thereafter concrete project goals are identified and an analysis is made of the data that is available for visualization. Chapter 4 describes the methods that were used to solve the problem and the design of the visualization tool. Next an implementation of a prototype of the tool, created during the project, is discussed. In chapter 6 the tool is evaluated and compared to another similar tool. Finally, the last chapter contains some conclusions and recommendations for further research.

Chapter 2

Problem description

Nowadays large software projects are common. A large number of programmers together work on a large number of files. Often these projects are distributed, i.e. work on the project is done at several locations around the world. Therefore the files that constitute the software project must be kept at a central location which can be accessed by each programmer. Moreover, the programmers must be able to keep track of each other's changes to the files. Version control systems are used to solve the resulting problems. Such a system stores all files at a central server which can be accessed from client computers used by the programmers. It also keeps multiple versions of each file. When a file is changed at the server, a new version of the file is created. The old version of the file is not deleted but maintained.
Software visualization can also support programmers in performing their tasks by displaying interesting (meta) data obtained from the software project. It can reveal certain problem areas in the software or other areas that require attention that would otherwise have remained unnoticed.
A third tool that can aid programmers is an Integrated Development Environment (IDE). An IDE helps programmers develop software by combining a number of software development tools (such as an editor, version control system client, compiler and debugger) into one consistent user interface. It enables programmers to perform complicated tasks, requiring the consecutive use of different tools, more efficiently.

The subject of this master project is interactive visualization of CVS data in the NetBeans IDE. CVS [6] stands for Concurrent Versions System. CVS is a widely used version control system. It enables developers and authors to record the history of their files. In what follows a few important concepts regarding CVS are described. Figure 2.1 shows a diagram of the terms that are explained. With CVS a repository is created on a server computer. A repository is a directory for a certain software project. The CVS repository stores a complete copy of all the files and directories which are under version control. From a client computer subdirectories can be created and files can be placed in this repository, as on the local hard drive. A difference with the local hard drive, however, is that multiple versions (revisions) of a file are maintained on the server. Normally, one never accesses any of the files in the repository directly. Instead, CVS commands are used to make a local copy (working copy) of a

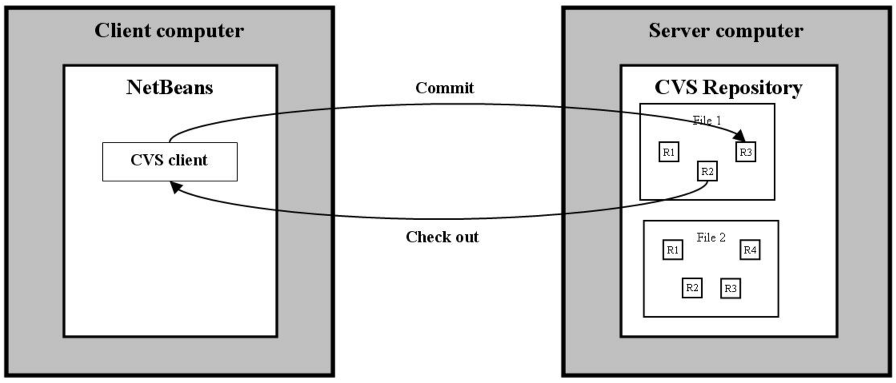


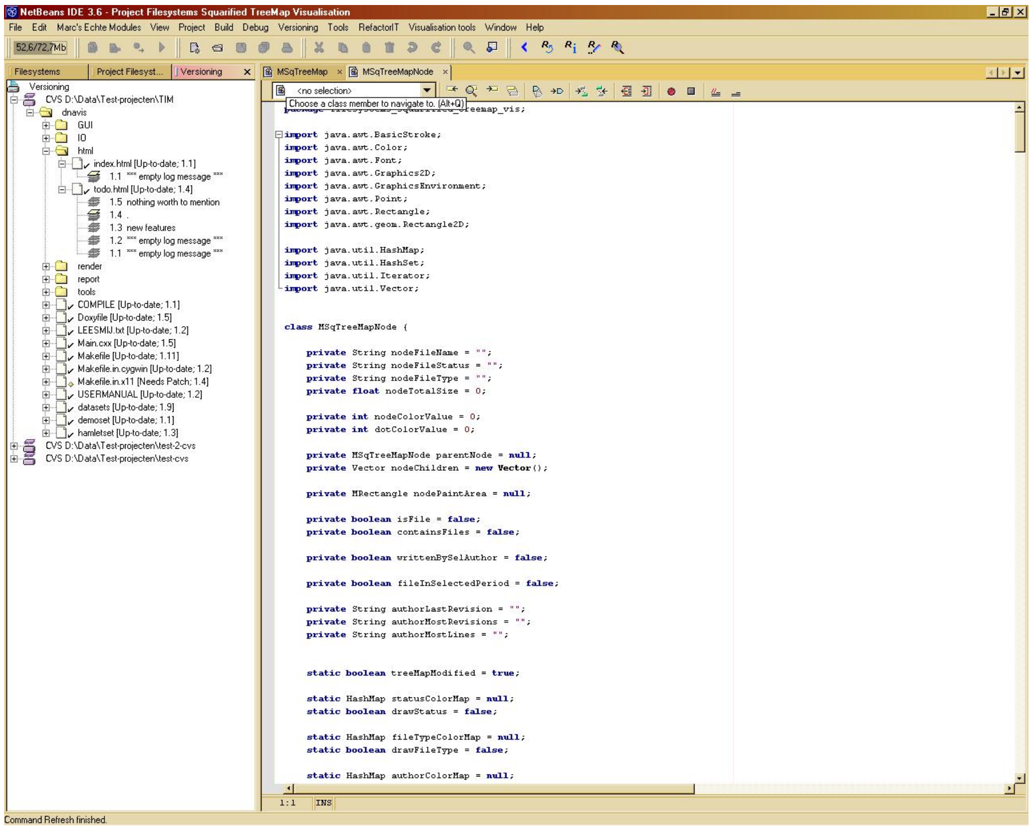
Figure 2.1: Diagram of CVS interaction

file on a client computer. Work is done on that copy. The process of making a working copy is called a "check out". After work has been finished on the working copy, the new version should be copied to the repository on the server. This process is called a "commit". A new revision of the file is created on the server. The previous revisions are also maintained. Therefore a file can have multiple revisions. For each revision additional properties such as author name, commit date and number of lines changed are stored.

The NetBeans IDE [16] is a development environment for Java [18] programs. NetBeans itself has also been written in the Java programming language. A screenshot of the NetBeans IDE is shown in figure 2.2. One of the tools available in NetBeans is a CVS client, which enables the user to perform CVS operations (such as "check out" and "commit") and shows CVS repositories in a treeview. An image of this treeview is shown in figure 2.3. It only visualizes the directory structure of the repository and for each file which revisions it contains. Other data available on the CVS server which might be of interest to the user, e.g. the author of each revision, are not shown. The aim of this project is to make more of this data accessible in NetBeans using interactive software visualization. Here, interactive means the user can select which areas of the data he/she wants to see (in more detail) by selecting these areas in the visualization using a mouse or keyboard.
An important property of NetBeans is that its software structure is modular. Consequently, NetBeans can easily be extended with new functionality such as a new visualization, without modifying existing parts of the IDE.

There are two concrete goals in this project. The first goal is to design an interactive visualization of CVS data available in NetBeans. In the remainder of this report this interactive visualization is called "the visualizer". The second goal is to implement this visualizer as a NetBeans module and consequently to integrate the implementation in the IDE. These goals lead to a number of requirements regarding the visualizer.

1. The visualizer must visualize CVS data obtained directly from the NetBeans IDE.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualisation tools Window Help
52,6/72,7Mb
Filesystems
Project Filesyst...
Versioning
MSqTreeMap
MSqTreeMapNode
<no selection>
Choose a class member to navigate to. (Alt+Q)
1:1
INS

```
Versioning
CVS D:\Data\Test-projecten\TIM
    dnavis
        GUI
        IO
        html
            index.html [Up-to-date; 1.1]
                1.1 *** empty log message ***
            todo.html [Up-to-date; 1.4]
                1.5 nothing worth to mention
                1.4 .
                1.3 new features
                1.2 *** empty log message ***
                1.1 *** empty log message ***
        render
        report
        tools
        COMPILE [Up-to-date; 1.1]
        Doxyfile [Up-to-date; 1.5]
        LEESMIJ.txt [Up-to-date; 1.2]
        Main.cxx [Up-to-date; 1.5]
        Makefile [Up-to-date; 1.11]
        Makefile.in.cygwin [Up-to-date; 1.2]
        Makefile.in.x11 [Needs Patch; 1.4]
        USERMANUAL [Up-to-date; 1.2]
        datasets [Up-to-date; 1.9]
        demoset [Up-to-date; 1.1]
        hamletset [Up-to-date; 1.3]
CVS D:\Data\Test-projecten\test-2-cvs
CVS D:\Data\Test-projecten\test-cvs
```

```java
package filesystems_squarified_treemap_vis;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.GraphicsEnvironment;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.geom.Rectangle2D;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Vector;


class MSqTreeMapNode {

    private String nodeFileName = "";
    private String nodeFileStatus = "";
    private String nodeFileType = "";
    private float nodeTotalSize = 0;

    private int nodeColorValue = 0;
    private int dotColorValue = 0;

    private MSqTreeMapNode parentNode = null;
    private Vector nodeChildren = new Vector();

    private MRectangle nodePaintArea = null;

    private boolean isFile = false;
    private boolean containsFiles = false;

    private boolean writtenBySelAuthor = false;

    private boolean fileInSelectedPeriod = false;

    private String authorLastRevision = "";
    private String authorMostRevisions = "";
    private String authorMostLines = "";


    static boolean treeMapModified = true;

    static HashMap statusColorMap = null;
    static boolean drawStatus = false;

    static HashMap fileTypeColorMap = null;
    static boolean drawFileType = false;

    static HashMap authorColorMap = null;
```

Command Refresh finished.

Figure 2.2: Screenshot of NetBeans IDE

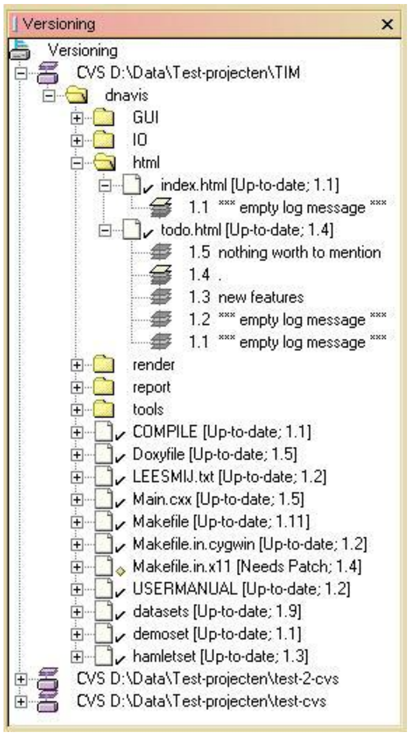
2. The visualizer must be implemented as a NetBeans module, preferably for the latest (stable) version of the NetBeans IDE.

3. The visualizer must be implemented in Java. This requirement is derived from the previous requirement.

4. The visualizer must provide ways for the user to select which areas of the data he/she wants to see (in more detail) interactively.

5. The user interface of the visualizer should be intuitive to use.

Requirement 5 mentions that intuitive user interaction with the visualizer is considered to be important. This is because the visualizer is not meant to be a specialistic tool requiring a lot of training.



```
Versioning
Versioning
CVS D:\Data\Test-projecten\TIM
    dnavis
        GUI
        IO
        html
            index.html [Up-to-date; 1.1]
                1.1  *** empty log message ***
            todo.html [Up-to-date; 1.4]
                1.5  nothing worth to mention
                1.4  .
                1.3  new features
                1.2  *** empty log message ***
                1.1  *** empty log message ***
        render
        report
        tools
        COMPILE [Up-to-date; 1.1]
        Doxyfile [Up-to-date; 1.5]
        LEESMIJ.txt [Up-to-date; 1.2]
        Main.cxx [Up-to-date; 1.5]
        Makefile [Up-to-date; 1.11]
        Makefile.in.cygwin [Up-to-date; 1.2]
        Makefile.in.x11 [Needs Patch; 1.4]
        USERMANUAL [Up-to-date; 1.2]
        datasets [Up-to-date; 1.9]
        demoset [Up-to-date; 1.1]
        hamletset [Up-to-date; 1.3]
CVS D:\Data\Test-projecten\test-2-cvs
CVS D:\Data\Test-projecten\test-cvs
```

Figure 2.3: Example of NetBeans versioning explorerview. It shows a number of CVS repositories in a treeview.

Chapter 3

Problem analysis

In this chapter the problem described in chapter 2 is further analyzed to find out exactly which data is to be visualized by the visualizer. As mentioned in chapter 2, the visualizer is to be implemented as a NetBeans module. Therefore only NetBeans users will use it. Two primary user groups of NetBeans can be identified:

1. Programmers
2. Software project managers

Programmers use the IDE to write Java programs. Software project managers use the IDE if they want to inspect the project by obtaining an overview of the files that constitute the project. To make the visualizer useful for as many users as possible, it is decided to create a visualization targeted at both user groups.

A small survey was conducted to find out which questions the visualizer should answer about CVS data. A number of questions were gathered and passed on to a few NetBeans users. The users were asked whether they considered the questions to be important and whether they had questions of their own about CVS data. The outcome of this process was a list of questions considered to be interesting to answer using a visualization. This list is given below.

- Questions a programmer may have concerning CVS data:
  - How important is this file with respect to the evolution of the repository?
  - Who knows most about this file/directory?
  - Who introduced this bug in this file?
  - Which are the old parts of the project, which the new parts?
  - What is the probability that there will be a new revision of this file soon?
  - Do I have the latest version of this file?
- Questions a project manager may have concerning CVS data:
  - What part of the total work did this person do and when did he perform it?

– What is the productivity of each author over time?
– During which time periods was most of the work done on the project?
– Who worked hardest during a certain time period?

Answering the questions mentioned above is chosen to be the concrete goal of the visualizer.
Many of the questions (of both user groups) can be parameterized in two ways:

1. The question can be applied to a restricted set of files.
2. The question can be applied to a restricted time period, in effect restricting the set of revisions under consideration.

The visualizer must provide ways to answer these parameterized questions. It must also enable the user to interactively set the parameters to the desired values.

3.1 Data analysis

To answer the questions mentioned above meta data needs to be extracted from the files and revisions in the repository and from the local working copies of these revisions. Here, a file is defined as the basic unit of directory structure on the CVS server. Multiple versions of a file can be present on the server, one version for each time the file is committed. Each version constitutes a revision. The relationships between files, revisions and working copies are shown in figure 3.1.

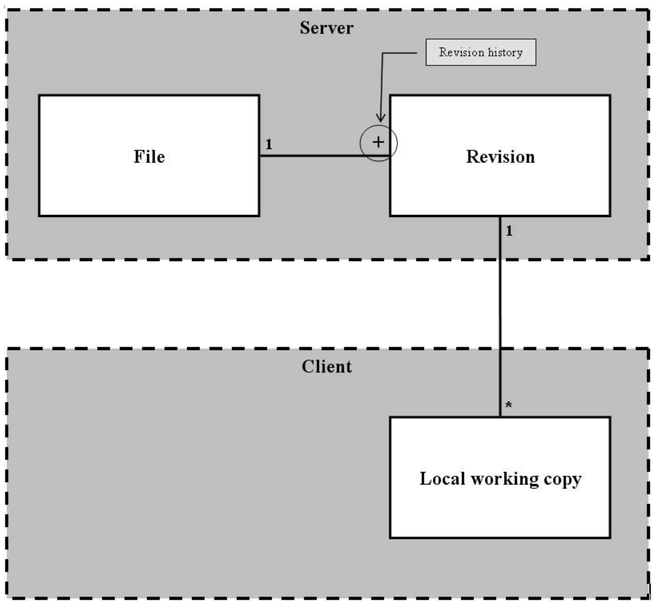


Figure 3.1: Diagram explaining CVS file concepts

Because of requirement 1 in chapter 2, the CVS data must be obtained directly from the NetBeans IDE. The following types of CVS data are available in the IDE:

1. Data related to the directory structure of the repository
2. Data related to the revision history of each file in the repository
3. Meta data concerning each revision in the repository
4. Meta data concerning a local working copy of each file in the repository
5. The text content of each revision in the repository

The data related to the directory structure of the repository consists of information concerning the contents of each directory, i.e. which subdirectories and files it contains.
The data related to the revision history of each file in the repository consists of information concerning which revisions each file has.
The (meta) data that is available on the server for each revision of a file is given below:

- Revision number
- Log message
- Revision date
- Author of revision
- Number of lines changed

The "log message" attribute contains a short message written by the author of the revision. It usually describes which modifications were made to the file with respect to the previous revision, the reason why these modifications were made and the effect of the modifications. This attribute is unstructured, which means it is very hard to automatically derive useful information from it. The "revision date" indicates when the revision was created. "Author of revision" contains the name of the author who created the revision. Finally, the "number of lines changed" attribute indicates how many lines were changed with respect to the previous revision of the file. This attribute is not available for the first revision of a file.
For a local working copy of each file, the following meta data is available:

- File name
- File size
- Date of last modification
- CVS file status

The "CVS file status" attribute indicates the status of the local working copy. The NetBeans versioning module discerns nine different values for this attribute. The two most common values of this attribute are "Up-to-date" and "locally modified". The value "Up-to-date" means the working copy is a check out of the last revision of the file on the server and that it has not been modified after

check out. The status “locally modified” means the working copy has been modified after check out and has not yet been committed. A description of the other possible values can be found at [9].
Note that no information can be retrieved about different working copies on different client computers. Therefore, such data cannot be used in answering the questions.
Finally, the text content of a local working copy of a file is also available in the IDE. The text content of any revision in the repository can be obtained by checking out a working copy of that revision. However, it is decided to ignore the text content of revisions. The reason for this is that a number of visualization tools which focus on this data already exists, e.g. standard diff visualization tools and CVSscan [19]. Therefore, only the first four types of data are used in the visualizer.

3.2 Mapping questions to data

The questions a programmer may have concerning CVS data, are all related to files and, hence, can be properly answered by visualizing certain attributes for each file. These attributes can be taken from the data available for a working copy of each file (see above) or, because each file has one or more revisions, from the data related to the revision history of the file or the data available for the corresponding revisions. A file can have more than one revision. Hence, the data of the revisions needs to be aggregated when being mapped to a file. This can be done in several ways, e.g. by taking the average value of a certain attribute of each revision or by taking an attribute value from one specific revision of the file. The methods to aggregate the data which are considered most useful in answering the questions mentioned above are used in the visualizer. The following (derived) attributes are considered to be most relevant to visualize for each file in answering the questions:

For the question “*How important is this file with respect to the evolution of the repository?*”:

- Number of revisions
- Number of authors
- Minimum number of lines changed aggregated over all revisions
- Average number of lines changed aggregated over all revisions
- Maximum number of lines changed aggregated over all revisions
- Total number of lines changed aggregated over all revisions
- File size of working copy
- File type

For “*Who knows most about this file/directory?*” and “*Who introduced this bug in this file?*”:

- Author of last revision

- Author who committed most revisions
- Author who changed most lines

During the project three criteria are defined in determining who knows most about a file: 1) the person who last changed the file, 2) the person who changed the file most often and 3) the person who changed the file the most. Though other criteria might be defined, the three criteria mentioned here are considered to be the most important.
In determining who introduced a bug in a file, it is assumed that it is known whether the bug has been present in the file only recently or for a long time. In the former case it is assumed the author of the last revision introduced the bug. In the latter case it is assumed the author who changed the file most often or who changed the file the most introduced the bug.
For these reasons, the three attributes mentioned above are considered important to visualize for both user questions.

For "*Which are the old parts of the project, which the new parts?*" the following attributes are considered to be most relevant to visualize:

- Date of first revision
- Date of last revision

For "*What is the probability that there will be a new revision of this file soon?*":

- Date of last revision
- Number of lines changed of last revision

During the project one criterion is defined to determine the probability that there will be a new revision a file soon: a large recent change to a file is assumed to indicate a high probability for a new change since bugs might have been introduced in the last change. Therefore the attributes mentioned above are considered most relevant to visualize to answer this question. Other criteria might be defined, e.g. by looking at the frequency of changes to a file in the past.

Finally, for the question "*Do I have the latest version of this file?*" the following attribute:

- CVS file status

The questions a project manager may have concerning CVS data are related to the amount of work done. Two definitions of "amount of work done during a certain time period" are considered: 1) the number of revisions that were committed during that period and 2) the number of lines that were changed accumulated over all revisions that were committed during the period. Both are directly related to revisions. Therefore, the "questions a project manager may have concerning CVS data" are directly related to revisions. These questions can be answered by visualizing certain attributes for each revision. The following attributes are considered most relevant to visualize for each revision in answering the questions:

For the question "*What part of the total work did this person do and when did he perform it?*":

- Revision date
- Author of revision
- Number of lines changed

For "*What is the productivity of each author over time?*":

- Revision date
- Author of revision
- Number of lines changed

For "*During which time periods was most of the work done on the project?*":

- Revision date
- Number of lines changed

Finally, for the question "*Who worked hardest during a certain time period?*":

- Revision date
- Author of revision
- Number of lines changed

Visualizing these attributes for each revision may also help in answering most of the questions a programmer may have concerning CVS data.

Chapter 4

Visualization design

Section 3.2 mentions that:

- Certain attributes must be visualized for each file in the repository.
- Certain attributes must be visualized for each revision in the repository.

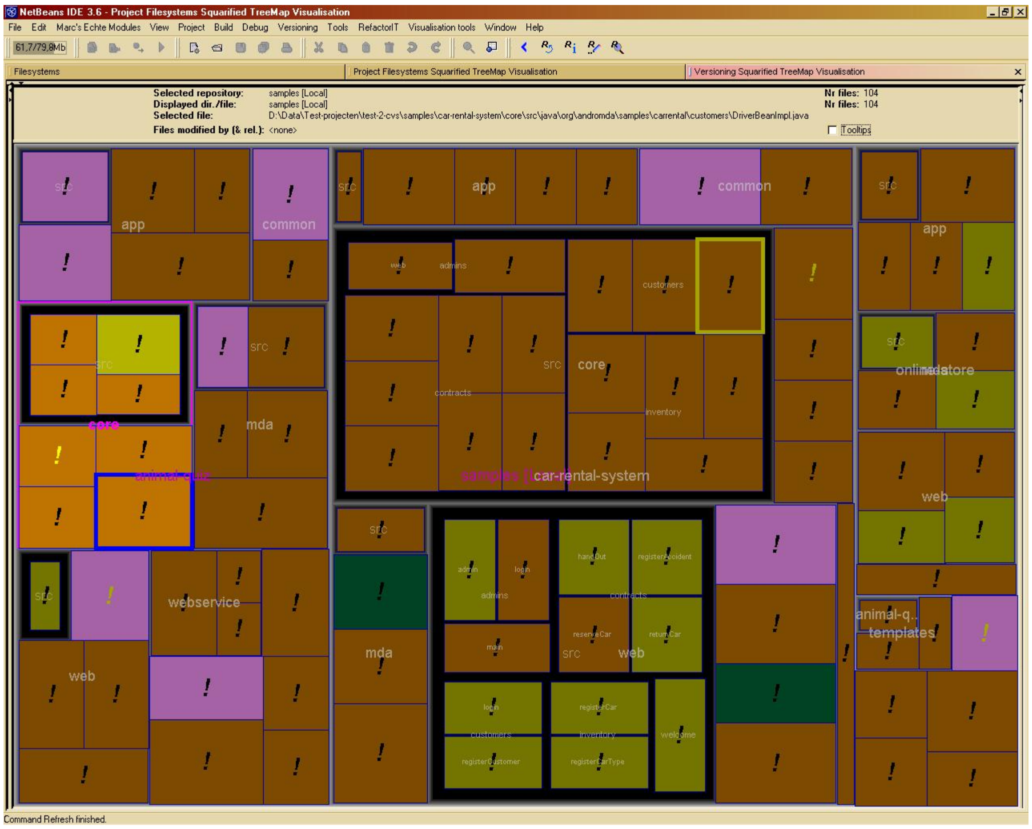
Therefore, to conveniently visualize these attributes, each file and revision must be represented by an entity (node) in the visualizer. Because the user must be able to find a specific file node, it is also desirable to visualize the directory structure of the repository. Furthermore, many of the user questions mentioned in chapter 3 are related to the evolution of the repository over time, i.e. the revision history of the files. Therefore, additionally the visualizer must visualize time related data. Data related to directory structure is tree shaped. Hence at least two dimensions are needed to properly visualize this data. Another dimension is needed to visualize data related to time. A 3D visualization can be used to display both types of data simultaneously. However, 3D visualizations are usually less clear than 2D visualizations due to occlusion problems [20]. Often, a great amount of user interaction is needed to accurately view the data. The user must manipulate the 3D view e.g. by means of rotation, translation and zooming to see the data he/she is interested in. This way the user cannot get a clear overview of the data, which is important in answering the questions mentioned in chapter 3. Therefore it is decided to create two coupled 2D visualizations, one for each type of data. Here, coupling means the visualization components are two views on the same data: they always display the same data set, each from its own perspective. If the selected data set in one component changes due to user interaction, the other component is automatically updated to also display this new data set.

4.1 Directory structure visualization

As mentioned above the directory structure of the repository needs to be visualized. It is important that a large number of files can be displayed simultaneously in the visualizer. This way the user can get an overview of the entire repository, also when the repository is large. This requires efficient usage of screenspace. Therefore a treemap [11] is chosen to display the directory structure of the repository. A treemap does not waste a large amount of screenspace such as e.g. a node-link diagram does. A treemap can display the directory structure of the repository, allocating a node for each file in the repository. Additionally it can display attributes for each node using two basic visual properties of the node: the node color and the node size (area). A standard treemap, however, often has thin elongated rectangles. As a result, rectangles are difficult to compare and to select. To solve this problem a special version of a treemap is used in the visualizer, called a squarified treemap [13]. A squarified treemap is a treemap with a lay-out in which the rectangles approximate squares.

4.1.1 Attribute visualization using glyphs

As mentioned above visualization of the directory structure of the repository is done by using a squarified treemap. The nodes in the treemap represent files. An example of the treemap view is shown in figure 4.1.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualisation tools Window Help
61,7/79,8Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Versioning Squarified TreeMap Visualisation
Tooltips
src
app
common
core
mda
webservice
web
onlinestore
animal-q...
templates
samples [car-rental-system
customers
contracts
inventory
admins
Command Refresh finished.

Selected repository: samples [Local]
Displayed dir./file: samples [Local]
Selected file: D:\Data\Test-projecten\test-2-cvs\samples\car-rental-system\core\src\java\org\andromda\samples\carrental\customers\DriverBeanImpl.java
Files modified by (& rel.): <none>
Nr files: 104
Nr files: 104

Figure 4.1: Example of squarified treemap view. The color of an exclamation mark indicates the value of one of the attributes concerned with "number of lines changed" for the corresponding file.

Section 3.2 describes which data attributes should be visualized for each file. In a squarified treemap each node has two basic visual properties to which attributes can be mapped: the node color and the node size (area). However, in this project more than two attributes need to be visualized simultaneously. A number of attributes for each file mentioned in section 3.2 is concerned with "number of lines changed". By simultaneously visualizing one of these attributes and two other attributes some of the user questions can be answered more effectively, i.e. more information that can help in answering the question is provided to the user simultaneously which makes it easier for the user to determine the answer. E.g., in answering the question "*How important is this file with respect to the evolution of the repository?*" at least three attributes are relevant: the number of revisions the file has, the number of authors that have worked on the file and the average number of lines changed aggregated over all revisions. Only visualizing the first two attributes answers this question less accurately. Therefore, the visualization of the treemap nodes is extended by adding a colored shape (glyph) to each node to display the "number of lines changed" attributes. The shape is positioned over the center of the node. Two criteria are defined in choosing the form of the glyph:

1. Adding the glyph to the nodes should not impair the visualization of the directory structure by the treemap.

2. The area of the glyph should be large enough to clearly show its color value.

Criterion 1 is important, because after adding the glyph to the nodes the user should still be able to clearly discern the directory structure of the repository. The second criterion is considered to be important, because it was decided to use the color value of the glyph to represent the value of the "number of lines changed" attributes. Because the user must be able to see how large the change to a file was to determine whether this file is of interest to him/her, the color value of the glyph must be clearly visible. The size of the glyph could also have been used for this purpose. However, the maximum size of the glyph is dependent on the size of the treemap node, which is used to represent another attribute value. To avoid dependencies in the treemap in the visualization of different attributes, the size of the glyph is not used to represent an attribute value at all.
A number of geometric forms, a.o. circles and squares, were tried as glyphs. They all failed to meet criterion 1. After adding such a glyph the nodes seemed to consist of two separate subnodes obscuring the directory structure of the repository.
Text symbols were also tested as glyphs. Adding a text symbol to the center of a node does not have the disadvantage mentioned above. The brightness of the color of the symbol can be varied depending on the attribute value to be displayed, a dark symbol representing a small attribute value, a bright symbol a large value. When displayed in bold, the area of text symbols is large enough to clearly show a particular color value. In this way text symbols also meet the second criterion. Therefore a text symbol is chosen as additional glyph.

In this project it is assumed that large changes to a file are more of interest to the user than minor changes. A large value for one of the "number of lines changed" attributes indicates a large change to a file at some point in time. Therefore it is decided to only display values for these attributes that are above a user defined threshold value. Hence, the new glyph should only be visible if the value to be displayed is above a certain threshold.

Considering the above, it is decided to use an exclamation mark as additional glyph. The exclamation mark, in contrast to other text symbols, emphasizes that an important (large) change has taken place in the file. Furthermore, a bold exclamation mark has a good shape to display a color value.

4.1.2 Treemap visualization in the visualizer

Now three attribute values can be displayed simultaneously by each node. As mentioned above all "number of lines changed" attributes are mapped to the exclamation mark (color). The remaining attributes mentioned in section 3.2 are mapped either to the node color or to the node size. A number of configurations was tested in mapping attributes to properties. The following mapping is found to be satisfactory in answering the user questions mentioned in chapter 3.
Attributes which are mapped to the node color:

- Date of first revision
- Date of last revision
- Number of revisions
- File type
- Author of last revision
- Author who committed most revisions
- Author who changed most lines
- CVS file status

In mapping file types, author names and CVS file states to the node color three colormappings are used respectively. Each of these colormappings can be modified by the user.
Attributes which are mapped to the node size:

- File size of working copy
- Number of authors

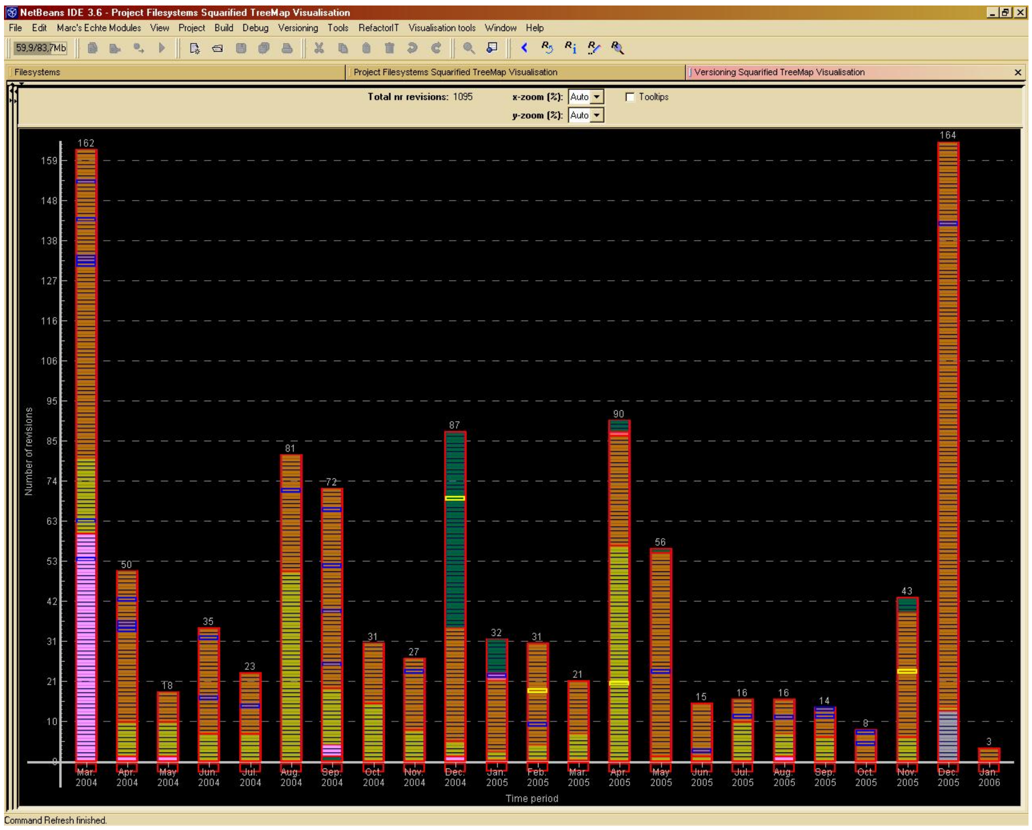
Finally, attributes which are mapped to the glyph color:

- Minimum number of lines changed aggregated over all revisions
- Average number of lines changed aggregated over all revisions
- Maximum number of lines changed aggregated over all revisions
- Total number of lines changed aggregated over all revisions
- Number of lines changed of last revision

The user can obtain detailed information concerning the file nodes using tooltips.

4.2 Revision history visualization

To visualize the evolution of the selected repository over time, a new 2D visualization is designed, called a revision history diagram. When visualizing time related data, time can be mapped to time to create an animated visualization. However, with an animated visualization it would be difficult for the user to get an overview of the entire evolution of the repository. Therefore an animated visualization is not suitable for these purposes and time must be mapped to one of the two visible dimensions of the 2D visualization. When mapping time to one of the visible dimensions two obvious choices for visualization are a curved (smooth) function plot and a bar graph. However, with the evolution of the repository over time the user is interested in the moments in time when a new revision is added to the repository. The evolution of the repository therefore consists of discrete time related data, not continuous data. A curved function plot is used to display continuous data, a bar graph discrete data. Hence, a bar graph is chosen as the starting point of the design of the new visualization. Figure 4.2 shows an example of the revision history diagram view of the visualizer.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualisation tools Window Help
59,9/83,7Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Versioning Squarified TreeMap Visualisation
Number of revisions
159 148 138 127 116 106 95 85 74 63 53 42 31 21 10 0
162 50 18 35 23 81 72 31 27 87 32 31 21 90 56 15 16 16 14 8 43 164 3
Mar 2004 Apr 2004 May 2004 Jun 2004 Jul 2004 Aug 2004 Sep 2004 Oct 2004 Nov 2004 Dec 2004 Jan 2005 Feb 2005 Mar 2005 Apr 2005 May 2005 Jun 2005 Jul 2005 Aug 2005 Sep 2005 Oct 2005 Nov 2005 Dec 2005 Jan 2006
Time period

Total nr revisions: 1095 x-zoom [%]: Auto
y-zoom [%]: Auto
☐ Tooltips

Command Refresh finished.

Figure 4.2: Example of revision history diagram view. The horizontal axis represents time. The bars consist of separate nodes. Each node corresponds to a revision. Revision nodes within in a bar are grouped by author.

As with a standard bar graph the revision history diagram 1) contains a horizontal and a vertical axis and 2) has vertical bars starting on the horizontal axis. In contrast to the bar graph, however, the bars in the history diagram consist of individual nodes. Figure 4.3 shows the general structure of a bar in a history diagram.

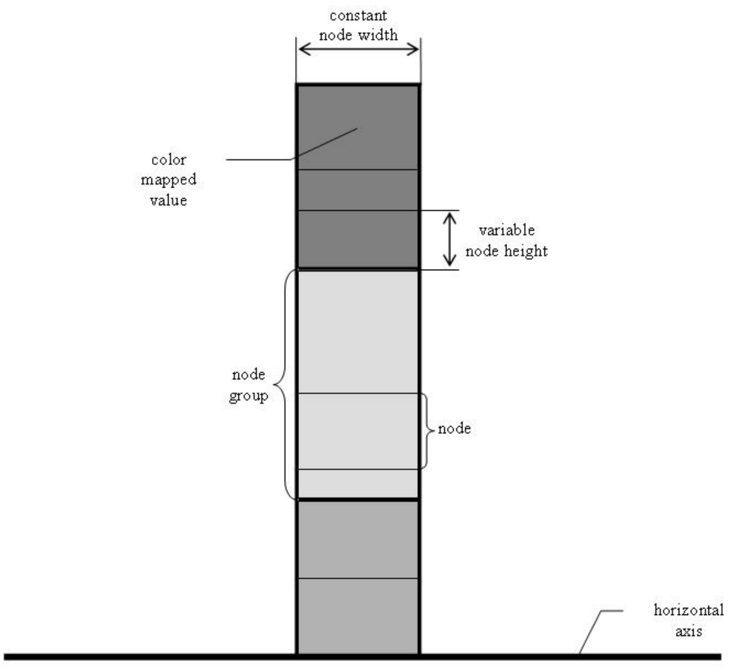


Figure 4.3: Diagram showing the general structure of a bar in a history diagram. The node height and node color can be varied. The node width is constant. Nodes can be grouped within in a bar.

A node is a separate rectangular area within a bar bounded by thin lines (see figure 4.3). These nodes can have a variable height. Therefore, the height of a node can be used to display a certain attribute value of the entity corresponding to that node. The width of the nodes is always constant, equal to the width of the bar. The color of the node area can be varied. Hence, as with the node height it can be used to display a certain attribute value. A final characteristic of the revision history diagram is that nodes can be grouped within a bar.

In the visualizer the nodes of the revision history diagram represent revisions. Figure 4.4 shows the structure of a bar in the revision history diagram of the visualizer.
Section 3.2 mentions three attributes that are considered to be important to visualize for each revision in answering the user questions and, hence, need to be visualized in the history diagram:

- Revision date
- Author of revision

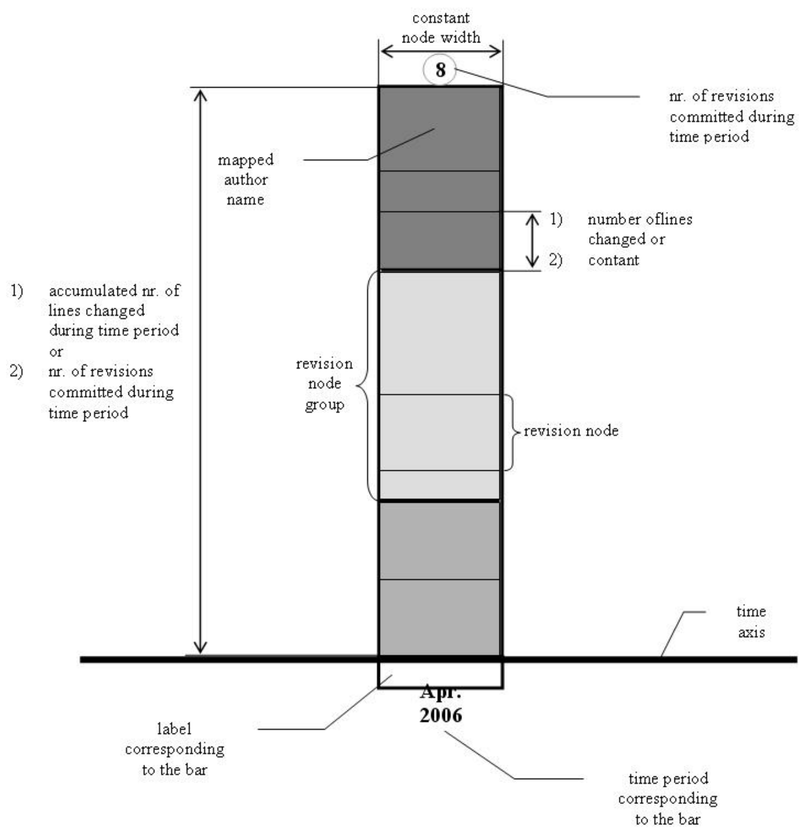

constant node width
8
nr. of revisions committed during time period
mapped author name
1) 2) number oflines changed or contant
1) accumulated nr. of lines changed during time period or
2) nr. of revisions committed during time period
revision node group
revision node
time axis
Apr. 2006
label corresponding to the bar
time period corresponding to the bar

Figure 4.4: Diagram showing the structure of a bar in the history diagram of the visualizer. Each node represents a revision. The node height is either constant or represents the number of lines changed in the corresponding revision. The author name of each revision is mapped to the node color. The node width is constant. Nodes can be grouped within in a bar by author, by file name or by revision date. The label below the bar is used to select the files and revisions from the time period corresponding to the bar (see section 4.3).

- Number of lines changed

In the visualizer the horizontal axis in the revision history diagram represents time. This way the "revision date" of a revision can be visualized by the horizontal position of the corresponding node in the diagram. Each bar contains the nodes of all revisions committed in the time period corresponding to that bar. The size of such a period can be chosen by the user: a day, a week, a month or a year. Hence, the user can to some extent choose how coarse or fine the "revision date" value is displayed in the diagram.
As mentioned above, each node has two basic visual properties to which attributes of a revision can be mapped. These properties are: the node color and the node area, which is proportional to the node height. The attribute "author of revision" is chosen to be mapped to the node color. The colormapping used to achieve this is the same colormapping as used in the treemap to map the "author" attributes to the treemap node color. This way, the user can easily relate author information displayed in the revision history diagram to author

information displayed in the treemap. As mentioned in section 4.1 this colormapping can be modified by the user.
A number of user questions in chapter 3 is concerned with the "amount of work that was done during a certain time period". As mentioned in the previous chapter, two ways to define this amount are considered during the project: the number of revisions that were committed during that period and the number of lines that were changed accumulated over all revisions that were committed during the period. The user must be able to choose which definition to use. Therefore, the node area is taken to be either constant or to represent the "number of lines changed" in the corresponding revision. Consequently, the vertical axis in the history diagram of the visualizer represents either the number of revisions that were committed during a time period or the number of lines that were changed during a time period.
In answering the user questions in chapter 3 the user is interested in the amount of work done by an author compared to other authors. To make comparing the amount of work done by each author easier, revision nodes inside a bar are grouped by author. Other grouping criteria the user can choose from in the visualizer are "by file name" or "by revision date".
The user can obtain detailed information concerning the revision nodes using tooltips.

Note that nodes in the treemap represent files, nodes in the history diagram represent revisions. Because a file can have multiple revisions, one node in the treemap can correspond to several nodes in the diagram.

4.3 User interaction with treemap and history diagram

As mentioned in the previous sections the treemap and the history diagram are two views on the same data. The two views are always shown in a consistent state. This requires updating both views whenever one of them changes as a result of user interaction with the visualizer.
User interaction with the treemap and the history diagram does not affect which repository/data is being visualized. It can only change which part of this data is currently displayed in the views or which part of the data is marked as selected. The visualizer ensures that both views always display the same part of the data and show the same part of the data marked as selected.
There are four forms of user interaction with the treemap and the history diagram:

1. File/revision history selection
2. Time period file/revision selection
3. Zooming
4. Tooltips display

The following paragraphs describe each form of user interaction and how the views are kept consistent after performing it.

File/revision history selection

There are two types of file/revision history selection, which differ from each other only in their duration. The first type, temporary selection, is initiated by brushing a node. A node is brushed by moving the mouse cursor over that node. The second type, persistent selection, is initiated by left clicking a node. The user can brush or click file nodes in the treemap or revision nodes in the history diagram. Brushing or clicking a file node results in selecting the corresponding file and all revisions corresponding to that file. Brushing or clicking a revision node results in selecting the corresponding revision, the file corresponding to that revision and all other revisions corresponding to that file. Consequently, brushing or clicking a node always results in selecting the corresponding file and its revision history.
Temporary selection, initiated by brushing a node, is canceled as soon as the mouse cursor is no longer over the node. Persistent selection, initiated by left clicking a node, is only canceled if the node is clicked again or if another node is clicked. Table 4.1 shows the state transitions of a node after brushing or clicking the node or removing the mouse cursor from the node.

	no sel.	**temporary sel.**	**persistent sel.**
brushing	temporary	temporary	persistent
clicking	—	persistent	temporary
remove cursor	—	no	persistent

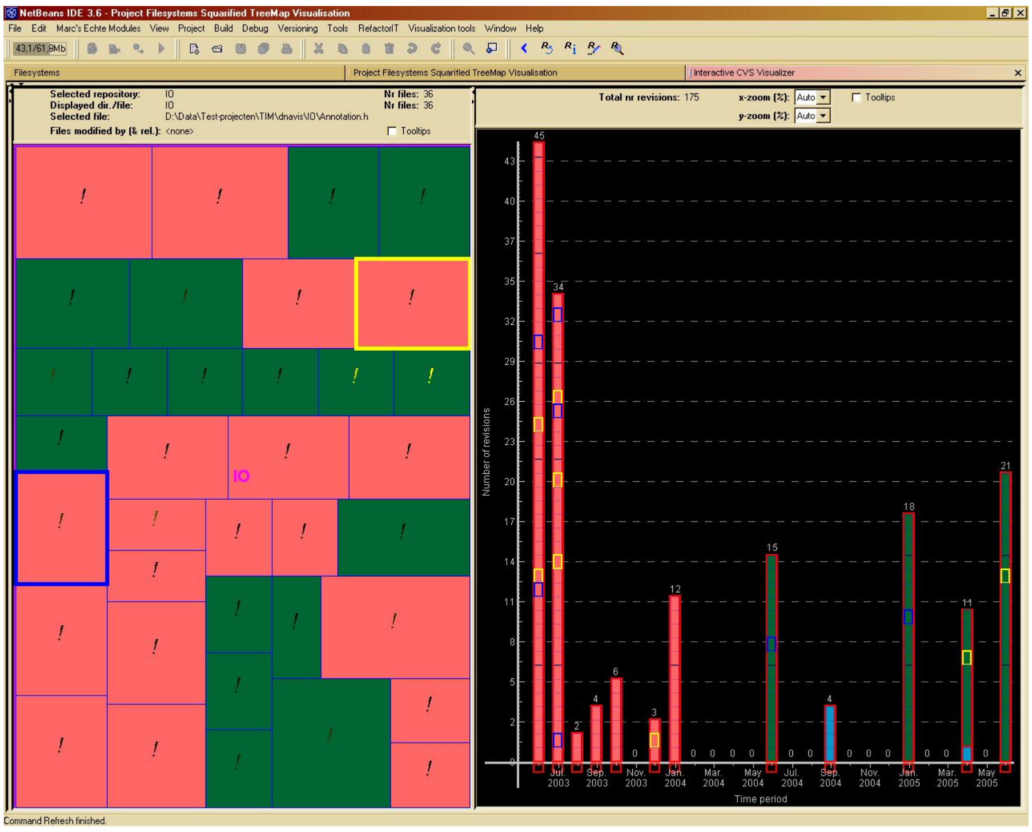
Table 4.1: Table showing the state transitions of nodes with regard to file/revision history selection. Each column represents a state of a node prior to one of the actions “brushing”, “clicking” or “remove cursor” from node. Each row represents one of these actions. Each cell *(c,r)* denotes the state of the node after the action of row *r* has been performed while the node was in the state of column *c*. Note that to click a node or to remove the mouse cursor from a node, the node must first be brushed.

There are two types of file/revision history selection to enable the user to compare the revision histories of two files. A comparison can be made by first applying persistent selection to one file and then applying temporary selection to another file.
Selection of a file and its revision history is displayed in the treemap and history diagram by highlighting the corresponding file node and revision nodes. Here, highlighting of a node means that the border of that node is displayed in bold. The color of the border is blue if temporary selection is applied to the file or revision corresponding to that node, yellow if persistent selection is applied. Table 4.2 describes the primary characteristics of the two types of file/revision history selection. Figure 4.5 shows an example of both temporary selection and

	temporary sel.	**persistent sel.**
initiated by	*brushing* a node	*left clicking* a node
displayed by	bold *blue* node border	bold *yellow* node border

Table 4.2: Table describing characteristics of the two types of file/revision history selection.

persistent selection.
When comparing the revision histories of two files, the different colors allow


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualization tools Window Help
43,1/61,8Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Interactive CVS Visualizer
Tooltips
IO
Number of revisions
Time period
Jul. 2003
Sep. 2003
Nov. 2003
Jan. 2004
Mar. 2004
May 2004
Jul. 2004
Sep. 2004
Nov. 2004
Jan. 2005
Mar. 2005
May 2005
Command Refresh finished.

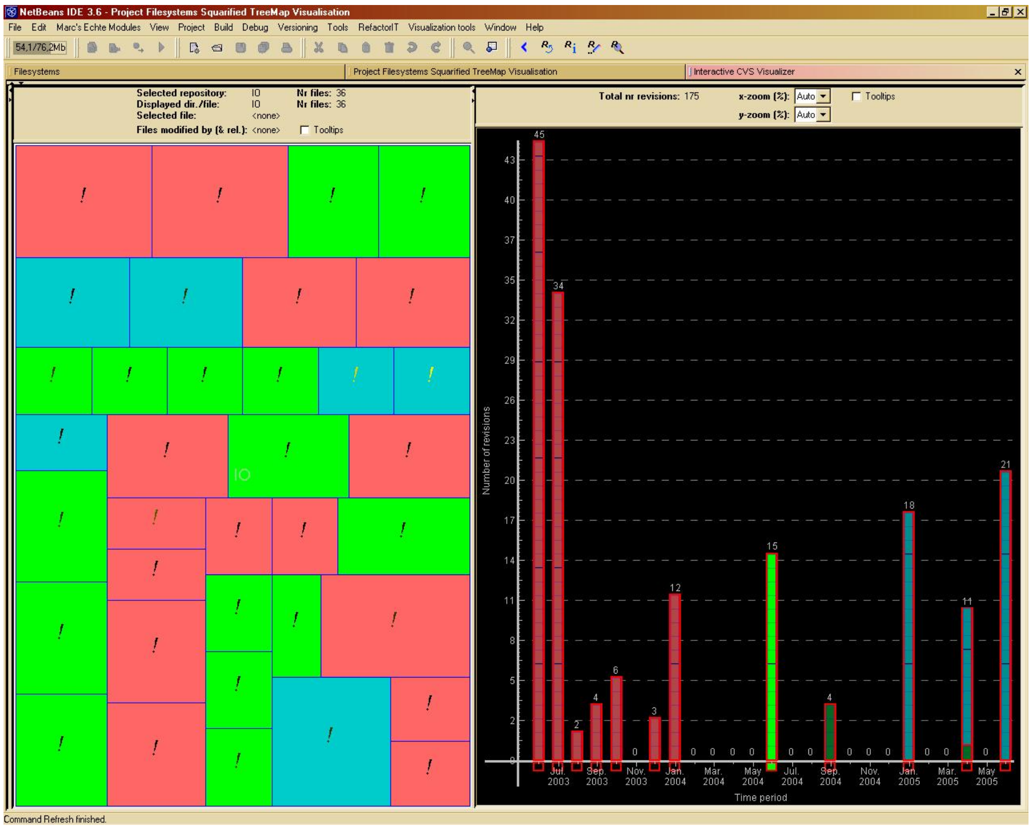
Selected repository: IO
Displayed dir./file: IO
Selected file: D:\Data\Test-projecten\TIM\dnavis\IO\Annotation.h
Files modified by (& rel.): <none>
Nr files: 36
Nr files: 36
Total nr revisions: 175
x-zoom (%): Auto
y-zoom (%): Auto

Figure 4.5: Example of both temporary and persistent file/revision history selection. Temporary selection has been applied to the file and revision nodes highlighted blue, persistent selection to the file and revision nodes highlighted yellow.

the user to discern which revision nodes belong to which file and hence make it easier to compare the revision histories. If both types of selection are applied to one file simultaneously, only persistent selection (yellow highlighting) is displayed (see tables 4.1 and 4.2). This is because there is no need to display both types of selection simultaneously, since comparing the revision history of one file to itself is no use. Furthermore, persistent selection is initiated by clicking a node, temporary selection by brushing a node. Before a node can be clicked, it must first be brushed. Therefore, always both types of selection are applied to one file simultaneously when persistent selection is initiated. After clicking a node, the user must be able to see that he/she has clicked the node. Hence, yellow highlighting (persistent selection) must be displayed after clicking a node. Consequently this is a situation where both types of selection are applied to one file simultaneously and only persistent selection is displayed. It is decided to handle all situations where both types of selection are applied to one file simultaneously consistently. Consequently only persistent selection (yellow highlighting) is displayed in these situations.

Time period file/revision selection

The user can select the files and revisions from a certain time period in the history diagram, i.e. the set of all revisions in the repository committed during that period and all files corresponding to these revisions (the files that were modified during the period). This is done by left clicking the label below the bar corresponding to that time period in the revision history diagram (see figure 4.4). Multiple time periods can be selected at one time. The files and revisions from a time period can be deselected by left clicking the corresponding label again. Note that, unlike as with file/revision history selection, not all revisions of a selected file are selected. Only the revisions committed during the selected time periods are selected.
The files and revisions from a selected time period are displayed in the treemap and history diagram by displaying all nodes corresponding to these files and revisions in bright green, i.e. the node color of these nodes is bright green. Also the area color of the label below the bar corresponding to the time period is bright green. Figure 4.6 shows an example of time period file/revision selection.



Selected repository: IO Nr files: 36
Displayed dir./file: IO Nr files: 36
Selected file: <none>
Files modified by (& rel.): <none> ☐ Tooltips

Total nr revisions: 175 x-zoom (%): Auto ☐ Tooltips
y-zoom (%): Auto

Command Refresh finished.

Figure 4.6: Example of time period file/revision selection. The area color of all nodes in the treemap and history diagram corresponding to selected files and revisions is bright green. The label below the bar corresponding to the selected time period is also bright green.

Note that this form of user interaction, in contrast to the other forms, only applies to the revision history diagram. The reason for this is, that it is impossible for the user to indicate a certain time period in the treemap. To select the files and revisions from a certain time period essentially one must be able to select the set of all revisions committed during that period. There is no way for the user to indicate in the treemap exactly which revisions (of a file) he/she wants to select.

Zooming

Chapter 3 mentions that many of the user questions can be parameterized to be applied to a restricted set of files and that the visualizer must provide ways to answer these questions. It also mentions that the visualizer must enable the user to interactively set the parameters to the desired values. The set of files displayed in the visualizer always consists 1) of all files in a certain root directory X and all files in all of its subdirectories or 2) of a single file X, which is then considered to be the root. Consequently by changing the root (directory) under consideration in the visualizer this set of files can be altered. Here the set of files displayed in the visualizer is denoted as *files(X)*. The set of all revisions corresponding to all files in *files(X)* is denoted as *revisions(X)*. Note that *files(X)* is the set of files displayed in the treemap, *revisions(X)* the set of revisions displayed in the history diagram.

To restrict the set of files displayed in the visualizer (and, hence because of coupling, the set of revisions displayed), the user can change the current root directory X to one of its direct subdirectories Y. The new set of files displayed, *files(Y)*, is a subset of *files(X)*. Therefore the user has effectively zoomed in on directory Y. The user can also change the current root directory X to one of the files Z directly in X. In this case the new set of files displayed, *files(Z)*, is also a subset of *files(X)*. The user has zoomed in on the file Z. After the user zooms in on a single file directly in the current root directory, zooming in is no longer possible.

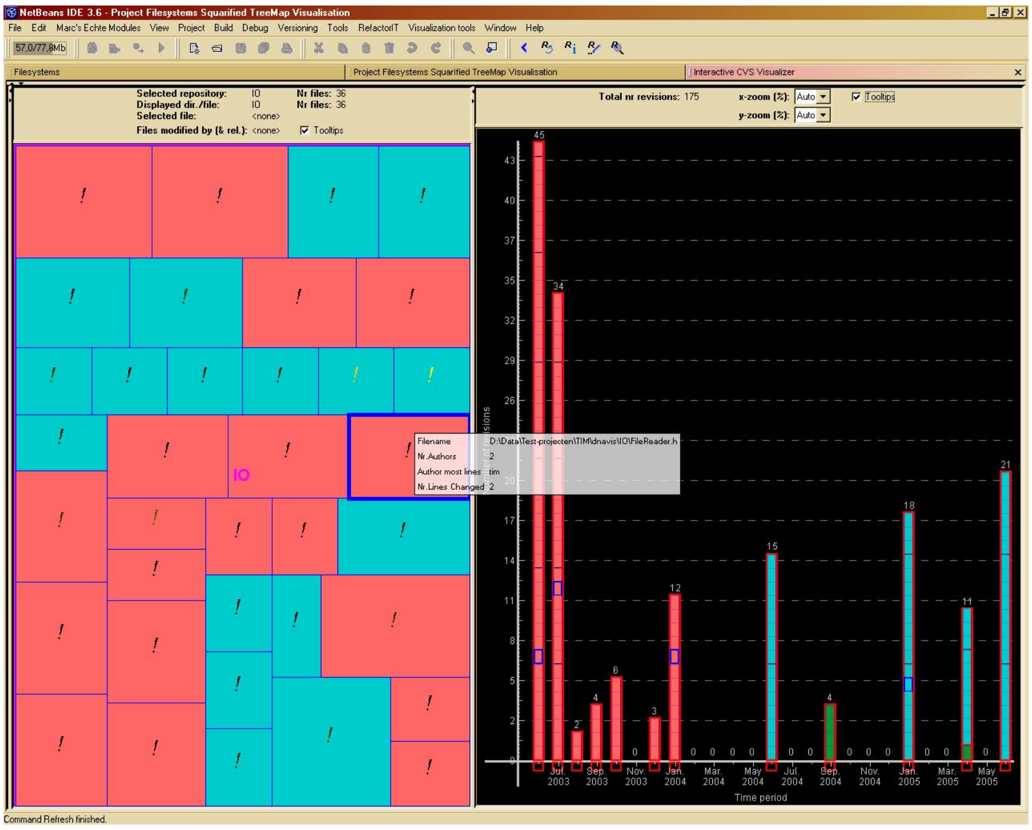
Suppose the current root directory under consideration is X. Then *files(X)* is displayed in the treemap, *revisions(X)* in the history diagram. If the user wants to zoom in on a direct subdirectory Y of X, then he/she can double click on one of the file nodes in the treemap corresponding to a file in *files(Y)* or a revision node in the history diagram corresponding to a revision in *revisions(Y)*. After double clicking the node the treemap displays *files(Y)*, the revision history diagram *revisions(Y)*. Similarly, if the user wants to zoom in on a file Z directly in X, he/she can double click on the file node corresponding to Z in the treemap or a revision node corresponding to a revision of Z in the history diagram. After double clicking the node the treemap displays *files(Z)*, i.e. only Z, the revision history diagram *revisions(Z)*, i.e. only all revisions of Z.

Double clicking using the right mouse button on a node in the treemap or the history diagram zooms out one level, i.e. performs the inverse operation.

Zooming acts as a filter on the directory structure of the data. It does not affect which repository is being visualized. It only changes which part of the repository is currently shown by the treemap and the history diagram.

Tooltips display

The user can initiate the display of a tooltip by brushing either a file node in the treemap or a revision node in the history diagram. The tooltip offers detailed information on the file or revision corresponding to the node that is brushed. Figure 4.7 shows an example of a tooltip. The information is displayed in tab-


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualization tools Window Help
57,0/77,8Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Interactive CVS Visualizer
IO
Number of revisions
Time period
Jul. 2003
Sep. 2003
Nov. 2003
Jan. 2004
Mar. 2004
May 2004
Jul. 2004
Sep. 2004
Nov. 2004
Jan. 2005
Mar. 2005
May 2005

Selected repository: IO Nr files: 36
Displayed dir./file: IO Nr files: 36
Selected file: <none>
Files modified by (& rel.): <none> ☑ Tooltips

Total nr revisions: 175 x-zoom (%): Auto ☑ Tooltips
y-zoom (%): Auto

Filename	D:\Data\Test-projecten\TIM\dnavis\IO\FileReader.h
Nr.Authors	2
Author most lines	tim
Nr.Lines Changed	2

Command Refresh finished.

Figure 4.7: Example of a tooltip, the transparent white textbox in the middle of the picture. The tooltip is displayed after the file node below the mouse cursor is brushed (the cursor is above the file node that has a bold blue border). The tooltip offers detailed information on the file corresponding to this node.

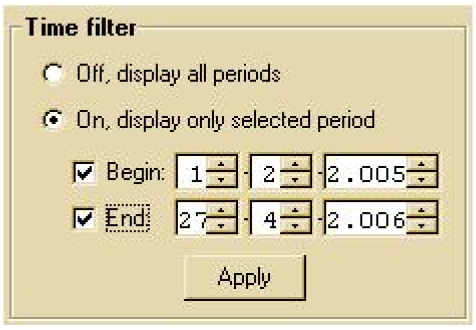
ular form. What information is shown depends on which attributes have been selected to be mapped to the node properties. The tooltip displays the exact (numeric or string) values of these attributes. Also the file name of the corresponding file is shown. A tooltip disappears as soon as the mouse cursor is moved away from the brushed node.
Because the size of the tooltips is quite large, their background is made transparent to reduce the occlusion induced on the treemap and the revision history diagram. Furthermore, the user can choose to disable tooltip display by clicking the checkbox shown above each of the components. After tooltips are disabled, brushing a node no longer causes a tooltip to be displayed, totally preventing occlusion of the treemap and the history diagram.

4.4 Other important features

The visualizer contains two other important features. These are the time filter and the directory/file authors lists. The features are described in the paragraphs below.

Time filter

Chapter 3 mentions that many of the user questions can be parameterized to be applied to a restricted time period and that the visualizer must provide ways to answer these questions. It also mentions that the visualizer must enable the user to interactively set the parameters to the desired values. If the user is interested in data from a specific time interval only, he/she can use the time filter of the visualizer. Figure 4.8 shows a detailed image of the time filter. Figure 4.9 shows the time filter embedded in the visualizer. It allows the user to specify a


Time filter

Off, display all periods [✓ On, display only selected period]

[✓] Begin: 1 - 2 - 2.005

[✓] End: 27 - 4 - 2.006

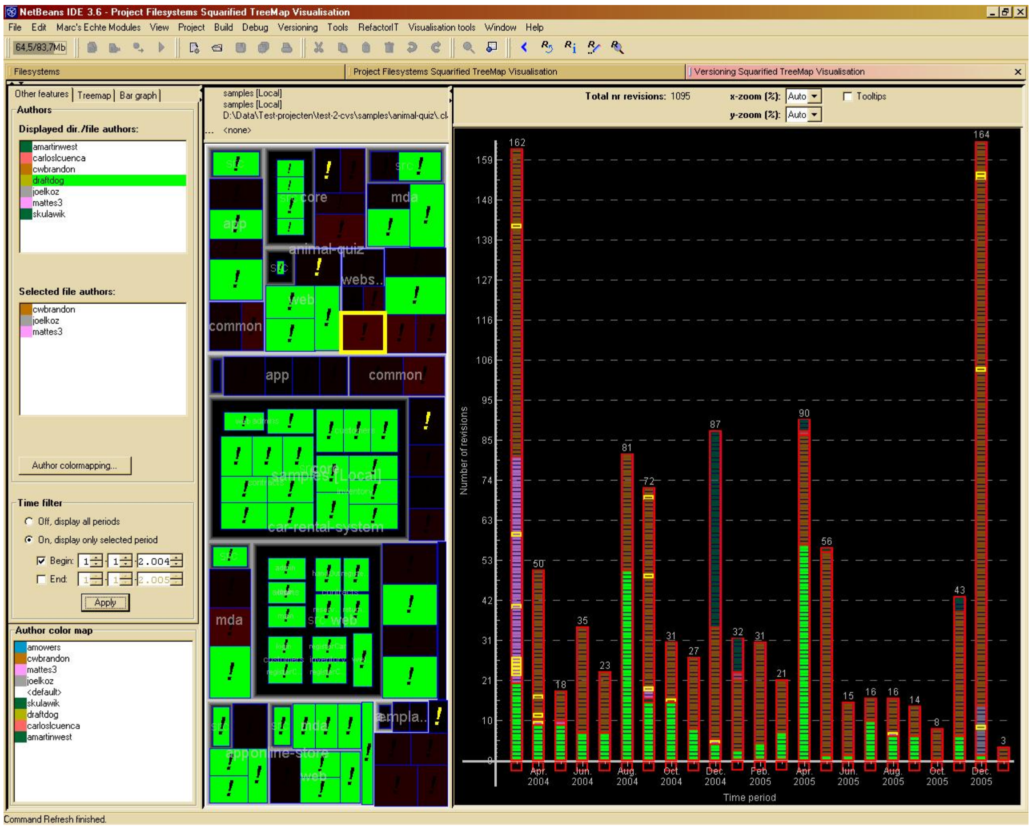
Figure 4.8: Detailed image of the time filter of the visualizer.

specific time interval using a begin and end date. If enabled, only data from the specified interval is shown in the treemap and the history diagram. This means that only revisions committed during the specified interval are used in deriving the attributes mentioned in section 3.2. Also only files modified during the interval are shown in the treemap, only revisions committed during the interval are shown in the history diagram.

Directory/file authors lists

A user might want to know which authors have worked on the files in the repository. He/she might also want to know which of these authors have worked on a specific file. Therefore the visualizer provides two lists: one that displays all authors of all files in *files(X)* for the current root X under consideration (see section 4.3) and one that displays all authors of the file to which persistent selection has been applied. Note that the contents of the first list can change when the user zooms in/out (see section 4.3). In the above, an author has worked on a file if he/she has committed at least one revision of that file.
If the user is interested in what work a specific author has done with regard to all files, he/she can select that author in the first list. After selection, all files in *files(X)* the author has worked on are selected (where X is the current root). Also all revisions in *revisions(X)* (see section 4.3) committed by the selected

author are selected.
The same question with regard to the file to which persistent selection has been applied only can be answered by selecting the author in the second list. Suppose Z is the file to which persistent selection has been applied. If an author in the second list is selected, Z is selected for author selection. Also all revisions of Z committed by the selected author are selected for author selection.
In the treemap a node corresponding to a file selected for author selection is displayed in bright green, i.e. the node color of the node is bright green. Similarly, in the history diagram a node corresponding to a selected revision is displayed in bright green. Figure 4.9 shows an example of author selection. An author has been selected in the author list corresponding to all files in *files(X)*. All files in *files(X)* the author has worked on are displayed in bright green in the treemap. All revisions in *revisions(X)* the author has committed are displayed in bright green in the revision history diagram.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation

samples [Local]
samples [Local]
D:\Data\Test-projecten\test-2-cvs\samples\animal-quiz\.cl
<none>

Total nr revisions: 1095
x-zoom (%): Auto
y-zoom (%): Auto
☐ Tooltips

Authors

Displayed dir./file authors:

🟩	amartinwest
🟥	carloslcuenca
🟧	cwbrandon
🟨	draftdog
⬜	joelkoz
🟪	mattes3
🟩	skulawik

Selected file authors:

🟧	cwbrandon
⬜	joelkoz
🟪	mattes3

Time filter
○ Off, display all periods
● On, display only selected period
☑ Begin: 1 - 1 - 2.004
☐ End: 1 - 1 - 2.005

Author color map

🟦	amowers
🟧	cwbrandon
🟪	mattes3
⬜	joelkoz
	<default>
🟩	skulawik
🟨	draftdog
🟥	carloslcuenca
🟩	amartinwest

Command Refresh finished.

Figure 4.9: Example of author selection. The author in the top author list marked green has been selected. All files this author has worked on are displayed green in the treemap. All revisions committed by the author are displayed green in the history diagram.

Note that author selection is similar to time period file/revision selection (see section 4.3) in the sense that with both forms of selection files and only a subset of the revisions corresponding to these files are selected. Therefore author selection is displayed in the visualizer in the same way as time period file/revision selection. However, this means that the two forms of selection cannot be displayed simultaneously, because otherwise the user would not be able to determine which bright green nodes belong to which form of selection. Hence, selecting an author in either of the two lists will deselect all possibly selected time periods. Similarly selecting a time period will deselect a possibly selected author.

Chapter 5

Implementation

This chapter describes a prototype implementation of the visualizer. As mentioned in the requirements in chapter 2 the visualizer must be implemented as a NetBeans module and hence be implemented in Java.
An important property of NetBeans is that it offers the so called Open APIs [16]. The Open APIs are libraries of classes and methods that allow programmers to access data in the IDE. The developers of the Open APIs take great care to keep the Open APIs consistent in new versions of NetBeans, i.e. the Open APIs are not modified in new versions of the IDE. This means that a module which only accesses data in NetBeans using the Open APIs will also work in future versions of the IDE. In this prototype implementation the Open APIs are used as much as possible to embed the visualizer program in the IDE.
As described in chapter 4 the visualizer consists of two coupled components, a squarified treemap and a revision history diagram. The two components are two views on the same data. They are always shown in a consistent state. To keep the views consistent communication needs to occur between the views. To achieve this, dependencies between the views might be introduced. However, the implementation is intended to be a prototype used for testing. This calls for a flexible design. It must be easy to adapt the code. To achieve separation of concern, communication between the views occurs by generating and processing events.
Rendering of the treemap and history diagram takes a long time due to the amount of data concerned. To keep the user interface (UI) responsive, rendering of the components is done in a separate thread (not in the UI thread). A multithreaded solution is created.
The prototype implementation of the visualizer consists of 19 classes and comprises ± 7500 lines of code. For the interested reader the details of the implementation are described in the following sections.

5.1 Implementation details

Requirement 2 in chapter 2 mentions the latest (stable) version of the NetBeans IDE. At the beginning of this project, February 2005, this was version 4.0. However, module development support [16, "Open APIs Support Module"] was not available for NetBeans 4.0 at that time. Module development support makes developing new NetBeans modules a lot easier. It contains a.o. documentation of the Open APIs and a New Module Wizard that helps in creating new basic modules, which can be extended to offer the desired functionality. There was only limited time available during the project and writing a NetBeans module from scratch requires a lot more work than with module development support. Therefore the version prior to 4.0 (which is 3.6) was chosen as the target platform, since this version did offer module development support.

To create the NetBeans module the Open APIs are used to access data and to properly embed the module in the NetBeans IDE. To create a new window in the IDE, a subclass of *TopComponent* is created. Figure 5.1 shows this relationship. The class *TopComponent* is part of the Open APIs and implements

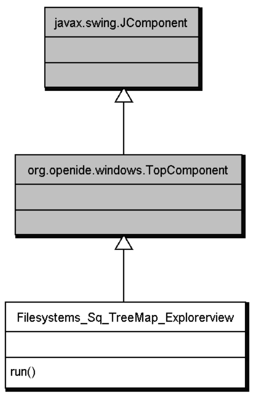

javax.swing.JComponent
org.openide.windows.TopComponent
Filesystems_Sq_TreeMap_Explorerview
run()

Figure 5.1: UML class diagram showing the inheritance in visualizer implementation related to TopComponent class. The class *TopComponent* is part of the Open APIs and implements an embeddable visual component to be displayed in the NetBeans IDE. *Filesystems_Sq_TreeMap_Explorerview* serves as top level window for the visualizer implementation. The classes shown in gray are either part of Swing or the Open APIs and are therefore not part of the implementation.

an embeddable visual component to be displayed in the IDE. It forms the basic unit of display in the NetBeans IDE. *TopComponent* is a subclass of the standard Java Swing [15] class *JComponent*. Therefore, all visual IDE components are Swing components. Hence, the Swing toolkit can be used in implementing the visualizer.
The subclass of *TopComponent* that has been created, *Filesystems_Sq_TreeMap_Explorerview*, serves as top level window for the visualizer implementation.

The visualizer implementation contains three main classes:

- *Filesystems_Sq_TreeMap_Explorerview*, the top level window mentioned above
- *MSqTreeMap*, which implements the squarified treemap
- *MBarGraph*, which implements the revision history diagram

A class diagram showing the relationships between these three classes is shown in figure 5.2. *MSqTreeMap* implements the visualization of the directory structure

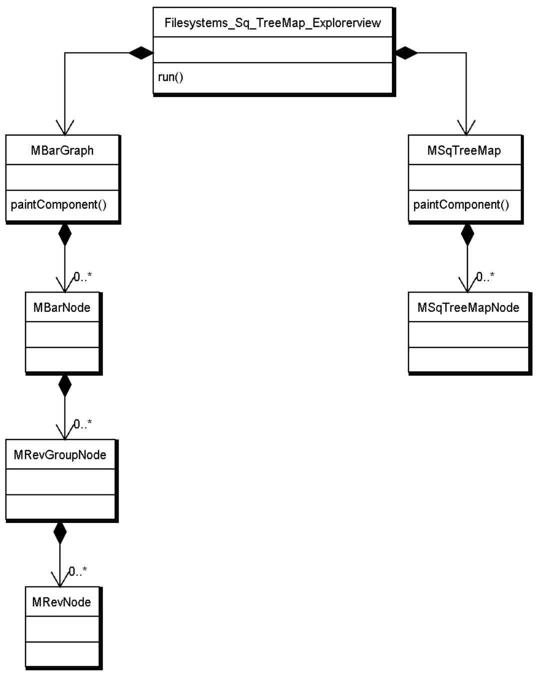


Figure 5.2: UML class diagram showing the three main classes of visualizer implementation: *Filesystems_Sq_TreeMap_Explorerview* (impl. the top level window), *MSqTreeMap* (impl. the treemap) and *MBarGraph* (impl. the history diagram). The treemap and history diagram are embedded in the top level window.

of the repository (see section 4.1), *MBarGraph* the visualization of the evolution of the repository over time (see section 4.2). *Filesystems_Sq_TreeMap_Explorerview* implements a.o. the directory/file authors lists (see section 4.4) and serves as top level window in which the treemap and history diagram are embedded. A description of the remaining classes in figure 5.2 can be found in section 5.1.4.

5.1.1 Rendering

Rendering of the treemap and history diagram components requires quite a long time (up to a few seconds when a repository comprising 500 files/2000 revisions is visualized). Handling this process in the AWT event dispatch thread (the user interface thread) would cause the user interface to freeze which is not desirable. Therefore, rendering of these components is done in a separate thread, here named visualizer thread. The visualizer module has a separate event queue which is needed to transfer control of the rendering from the AWT event dispatch thread to the visualizer thread. Figure 5.3 shows the class implementing this event queue, *MEventQueue*, and the relationships between the treemap, history diagram and event queue.
Repainting of the treemap and revision history diagram is a two step process. This process is shown in figure 5.4. When one of the components needs to be repainted it adds a paint event to the event queue, transfering control of the rendering to the visualizer thread. Handling this event results in updating an

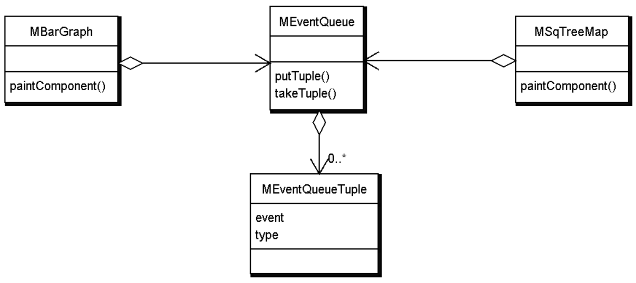


Figure 5.3: UML class diagram showing the relationships between the classes implementing the treemap (*MSqTreeMap*), history diagram (*MBarGraph*) and event queue (*MEventQueue*). When the treemap or history diagram needs to be repainted or to communicate with the other visualization component, it produces an event in the event queue. *MEventQueueTuple* is used to wrap such an event (see section 5.1.4).

offscreen buffer and a repaint call to Swing, transfering control back to the AWT event dispatch thread. When the repaint call is processed the offscreen buffer is written to the screen. The reason why this is a two step process is because Swing requires that all painting is performed by the AWT event dispatch thread. It is not allowed to do this in the visualizer thread of the visualizer module for reasons of efficiency.

5.1.2 Communication between treemap and history diagram

The visualization module created during the project is intended to be a prototype used for testing. To be able to test the treemap and history diagram separate from each other, e.g. to measure memory consumption or processing time, dependencies between these components have to be avoided as much as possible. To achieve separation of concern, the communication between the treemap and history diagram is done by generating and processing events. E.g. after the user zooms in on a directory in the treemap, it produces an event to update the history diagram to keep the two views consistent. Some of the communication events take a long time to be processed. As with the repainting, handling such an event in the AWT event dispatch thread would cause the user interface to freeze which is not desirable. Therefore, the event queue which was introduced above (see figure 5.3) is also used for communication between the treemap and the history diagram. In this way the event queue acts as a Mediator [8] between these two components, promoting loose coupling between them. The communication events are processed by the visualizer thread which also handles the paint events. The problem could also have been solved by introducing another event queue and thread specifically to handle the communication events. However, this would unnecessarily complicate synchronization between threads.

5.1.3 CVS data acquisition

As mentioned in the requirements in chapter 2 the visualizer implementation must obtain the CVS data directly from the NetBeans IDE. Figure 5.5 shows all

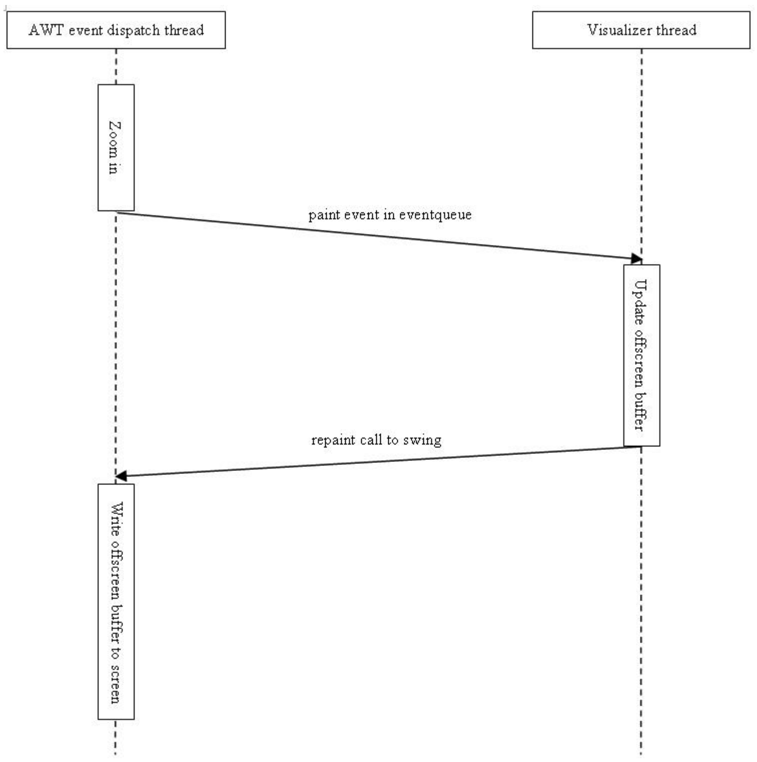

AWT event dispatch thread
Visualizer thread
Zoom in
paint event in eventqueue
Update offscreen buffer
repaint call to swing
Write offscreen buffer to screen

Figure 5.4: Diagram showing an example of the rendering process of the treemap and history diagram after the user zooms in on a directory in the treemap. Each of the two vertical lines shows the activities of the corresponding thread with respect to the rendering process. The vertical order of the activities in the diagram represents their respective order in time. The arrows between the vertical lines represent transfer of control of the process between the threads.

classes of the visualizer implementation involved in the retrieval of CVS data. Within NetBeans 3.6 this data is available as a *NodeTreeModel*, a subclass of the Swing class *DefaultTreeModel* (see figure 5.5). However, the root node of this model, which is needed to construct the model, cannot be obtained using the Open APIs. It can only be retrieved directly from the versioning module of the IDE. This means that the visualizer module might not work in newer (> 3.6) versions of NetBeans. However, testing the module in a newer version of NetBeans without changing the Open APIs might be desirable. Because the CVS data might only be available in a different form than a *NodeTreeModel* in this newer version, it must be easy to adapt the visualizer module to retrieve CVS data from another data source. Therefore, the treemap and history diagram should not have direct dependencies on the *NodeTreeModel*. Hence a separate class, *GetDataFromModel* (see figure 5.5), is created to retrieve the CVS data from the *NodeTreeModel*. The treemap and history diagram use this class to obtain their data. To retrieve the CVS data from another source, only the implementation of *GetDataFromModel* needs to be changed. *GetDataFromModel* acts as a Facade [8] to the CVS data, promoting weak coupling between the two visualization components and the CVS data source and making the data source

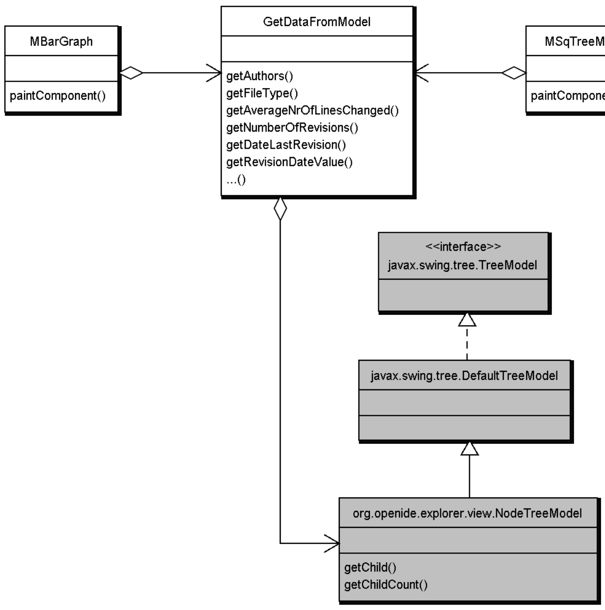


Figure 5.5: UML class diagram showing all classes in the visualizer implementation involved in the retrieval of CVS data. When the treemap (impl. by *MSqTreeMap*) or history diagram (impl. by *MBarGraph*) needs to access CVS data, it sends a request to *GetDataFromModel*. *GetDataFromModel* retrieves the data from the *NodeTreeModel* and aggregates the data to deliver the requested information. All classes shown in gray are either part of Swing or the Open APIs and are therefore not part of the implementation.

easier to use.
The *NodeTreeModel* used by *GetDataFromModel* contains three types of nodes: nodes which represent directories, nodes which represent working copies and nodes which represent revisions. Each node has a number of properties. Which properties it has depends on the type of the node. The directory nodes have only two properties: “name” and “sort mode”. The data available for the working copy and revision nodes is mentioned in section 3.1. *GetDataFromModel* reads the values of these properties and processes (aggregates) the data to deliver the attributes mentioned in section 3.2.
However, retrieval of revision data from the server using the *NodeTreeModel* is slow. For example reading a repository of 250 files over a local network takes about 3 minutes. NetBeans sets up a new connection with the CVS server for each file to retrieve its revision data. This causes a lot of overhead, because each time 1) the connection needs to be set up, 2) the data must be retrieved, and finally 3) the connection must be disconnected again. Setting up one connection to retrieve the revision data of all files in the repository would be more efficient, since steps 1 and 3 need to be executed only once. To achieve this either the versioning module of the IDE must be modified, increasing the efficiency of the *NodeTreeModel*, or a separate data backend must be written to obtain the data directly from the CVS server, avoiding the use of the *NodeTreeModel*. The latter option, however, violates the requirement that the visualizer module must obtain the CVS data directly from the NetBeans IDE. The former option can only

be achieved by sending a request for change to the appropriate developer mailing list (see [16]). Because the visualization module created during this project is intended only for testing purposes, performance is not crucially important. Hence, no actions were undertaken to solve this inconvenience. Furthermore, once the revision data has been retrieved from the server it is stored in a local cache. The cache is only flushed when the NetBeans IDE is closed. Hence, reading the data is only slow the first time during a session with the NetBeans IDE, i.e. during the time period after the NetBeans IDE program has been started and before it is closed.

5.1.4 Description of classes

An overview of all classes used in implementing the visualizer is shown in figure 5.6. Below a short description is given of each of these classes.

- **FilesystemsSqTreeMapAction**
  FilesystemsSqTreeMapAction is a subclass of *CallableSystemAction* which is part of the Open APIs. *CallableSystemAction* implements an action which may be called programmatically (see [16]). Such an action is used to handle Action events [15], which are produced e.g. when a button is pressed in the user interface. The code associated with the action, i.e. its *performAction()* method, is executed when an event is handled.
  In the menu bar of the NetBeans IDE, selecting the menu item corresponding to the visualizer module causes the *performAction()* method of *FilesystemsSqTreeMapAction* to be executed. The *performAction()* method creates an instance of *Filesystems_Sq_TreeMap_Explorerview* and embeds this window in the IDE. After that the method ends its execution.

- **Filesystems_Sq_TreeMap_Explorerview**
  Filesystems_Sq_TreeMap_Explorerview is a subclass of *TopComponent* which is part of the Open APIs. This subclass implements the top level window of the visualizer implementation. The treemap and history diagram, as well as a number of user interface components, are embedded in this window.
  It also contains an innerclass that implements the visualizer thread which processes all events from the event queue of the visualizer module (see section above).

- **GetDataFromModel**
  GetDataFromModel retrieves the CVS data from the *NodeTreeModel* and acts a Facade to the data (see section above). It is used to provide all data to the treemap and revision history diagram. It reads and processes (aggregates) all data from the *NodeTreeModel* to deliver the attributes mentioned in section 3.2.
  The name of this class, *GetDataFromModel*, implies that it retrieves data from a model. However, another implementation of the class might retrieve CVS data from a different source (see section above). For historic reasons the name *GetDataFromModel* was introduced and unfortunately never changed. A better name for the class would have been e.g. *CVSFacade*.

- **MAuthorColorMapper**
  MAuthorColorMapper is a subclass of the standard Swing class *JFrame*. It implements a separate window in which the user can alter the colormapping for all authors that have worked on the files in the repository.

- **MAuthorsListCellRenderer**
  MAuthorsListCellRenderer implements the Swing *ListCellRenderer* interface. This auxiliary class is used to render two values (a color and a name) in one cell of a list.

- **MAuthorsListTuple**
  MAuthorsListTuple is an auxiliary class that implements a data container storing two values: a color and a name.

- **MBarGraph**
  The *MBarGraph* class implements the revision history diagram. It is a subclass of the standard Swing class *JComponent*. An instance of the class is embedded in the *Filesystems_Sq_TreeMap_Explorerview*. *MBarGraph* uses *GetDataFromModel* to retrieve its data. It communicates with the treemap and explorerview by adding events to the event queue of the visualizer module (see section above). Finally, it uses one instance of *MBarNode* for each bar that is added to the history diagram.
  The name of this class, *MBarGraph*, refers to the fact that the design of the revision history diagram is inspired by a bar graph. This name was introduced for historic reasons and never changed. A better name would have been e.g. *MRevHistDiagram*.

- **MBarNode**
  The *MBarNode* class implements one bar to be displayed in the revision history diagram. It stores one instance of *MRevGroupNode* for each group of revisions the bar contains. The most important tasks of *MBarNode* are assigning each revision to the correct revision group and managing the sorting of revision groups within a bar.

- **MColorIndicator**
  MColorIndicator is a subclass of the Swing class *JComponent*. This auxiliary class implements a user interface component that shows a bar with a particular color fading from bright to dark. It is used in legends to show which brightness of the color indicates which value.

- **MEventQueue**
  The *MEventQueue* class implements the event queue of the visualizer module (see section above). It is used to store the paint events generated by the treemap and the history diagram. It is also used by the treemap and history diagram to communicate with each other and the explorerview by adding events to the queue. A separate thread processes all events from the queue.
  Some optimization is done by the queue. E.g. to prevent unnecessary repainting of the treemap and history diagram, the queue always contains at most one paint event, i.e. if the queue contains a paint event an attempt to add another paint event will be ignored. Furthermore, processing of a paint event is postponed as long as possible. If the queue also contains

other types of events, then the paint event will be placed at the end of the queue.

- **MEventQueueTuple**
  MEventQueueTuple is an auxiliary class that implements a data container storing two values: an event object and an event type. All events that are put in the event queue of the visualizer module are wrapped in an instance of this class.

- **MFileTypeColorMapper**
  MFileTypeColorMapper is a subclass of the Swing class *JFrame*. It implements a separate window in which the user can alter the colormapping for all file types that occur in the repository.

- **MNameColorCellRenderer**
  MNameColorCellRenderer implements the Swing *ListCellRenderer* interface. This auxiliary class is used to render two values (a color and a name) in one cell of a list. It differs from *MAuthorsListCellRenderer* in that another background color is used when rendering a selected item.

- **MNameColorTuple**
  MNameColorTuple is an auxiliary class that implements a data container storing two values: a color and a name. The definition of this class equals the definition of the *MAuthorsListTuple* class and could be replaced by it. However, for historic reasons this class was introduced and never replaced.

- **MRevGroupNode**
  MRevGroupNode implements one group of revisions to be displayed within a bar of the revision history diagram. It stores an instance of *MRevNode* for each revision the group contains.

- **MRevNode**
  The *MRevNode* class implements one revision node to be displayed in the history diagram. It stores data such as file name, author and revision date for one revision.

- **MSqTreeMap**
  The *MSqTreeMap* class implements the treemap. It is a subclass of the Swing class *JComponent*. An instance of the class is embedded in the *Filesystems_Sq_TreeMap_Explorerview*. *MSqTreeMap* uses *GetDataFromModel* to retrieve its data. It communicates with the history diagram and explorerview by adding events to the event queue of the visualizer module (see section above). Finally, it uses one instance of *MSqTreeMapNode* for each node that is added to the treemap.

- **MSqTreeMapNode**
  MSqTreeMapNode implements a node to be displayed in the treemap. Such a node can represent either a directory or a file. Directory nodes can have children and therefore contain a list of child nodes. File nodes are the leafs of the treemap. Hence, the list of child nodes is always empty in these nodes.

- **MStatusColorMapper**
  MStatusColorMapper is a subclass of the Swing class *JFrame*. It implements a separate window in which the user can alter the colormapping for all possible values of CVS file status.

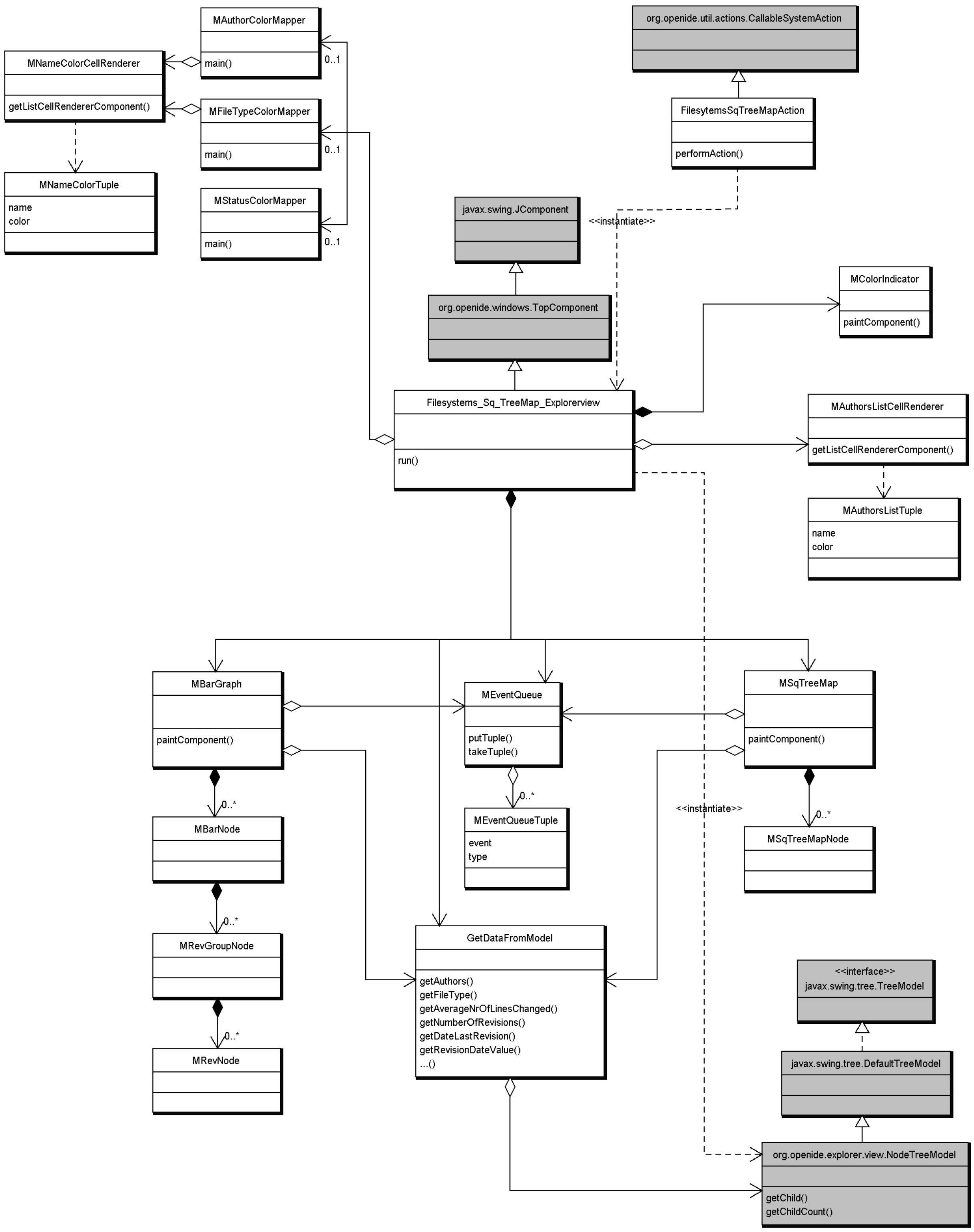


Figure 5.6: UML class diagram showing an overview of all classes used in the visualizer implementation. A description of each of the classes can be found in section 5.1.4. The classes shown in gray are either part of Swing or the Open APIs and are therefore not part of the implementation.

Chapter 6

Evaluation

To evaluate the visualizer created during this project, it is first investigated if and how the tool answers the questions mentioned in chapter 3. After that, it is compared to other similar tools. The following sections describe if and how the tool answers the user questions, what other tools were found for comparison, which of these were considered useful to compare to and finally an actual comparison of the tools.

6.1 Answering the user questions

In this section it is shown how the visualizer answers the questions mentioned in chapter 3. A CVS repository containing the source code of the Compaq Extended Static Checker for Java (ESC/Java) [10, 3] is used as example data. ESC/Java is a programming tool that attempts to find common run-time errors in Java programs by static analysis of the program text. The repository contains ± 200 files/ ± 1900 revisions. A number of authors (± 15) has worked on the project during a time period of about four years. These characteristics make this repository ideally suited for the visualizer. It contains enough files/revisions to be interesting to visualize, though not too many to display. It also contains a large amount of author data which makes it interesting to visualize.

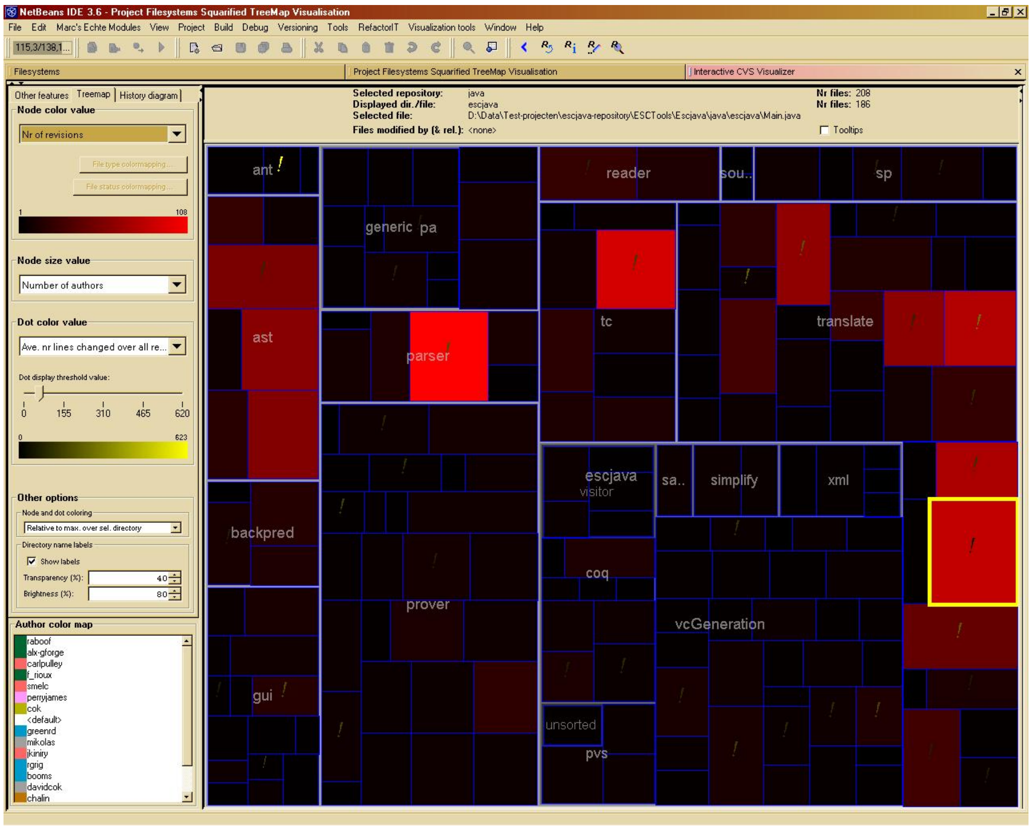
The user question "*How important is this file with respect to the evolution of the repository?*" can be answered by considering how often the file has been changed, i.e. the number of revisions it has, a large number indicating an important file. Another aspect involved in answering this question is how large each change to the file was. An indication thereof can be obtained by considering the minimum, average, maximum or total number of lines changed aggregated over all revisions, a large value indicating large changes and hence an important file. A third aspect which is important with respect to this question is the number of authors that have worked on the file, again a large number indicating an important file. Considering the above this question can be answered using the visualizer by selecting:

- "Number of revisions" to be mapped to the treemap node color
- "Minimum ..", "Average ..", "Maximum .." or "Total number or lines

changed aggregated over all revisions" to be mapped to the exclamation mark color

- "Number of authors" to be mapped to the treemap node size

Figure 6.1 shows an example of this.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation

Node color value: Nr of revisions

Node size value: Number of authors

Dot color value: Ave. nr lines changed over all re...

Dot display threshold value: 0 155 310 465 620

Other options

Node and dot coloring: Relative to max. over sel. directory

Directory name labels: ☑ Show labels

Transparency (%): 40

Brightness (%): 80

Selected repository: java

Displayed dir./file: escjava

Selected file: D:\Data\Test-projecten\escjava-repository\ESCTools\Escjava\java\escjava\Main.java

Files modified by (& rel.): <none>

Nr files: 208

Nr files: 186

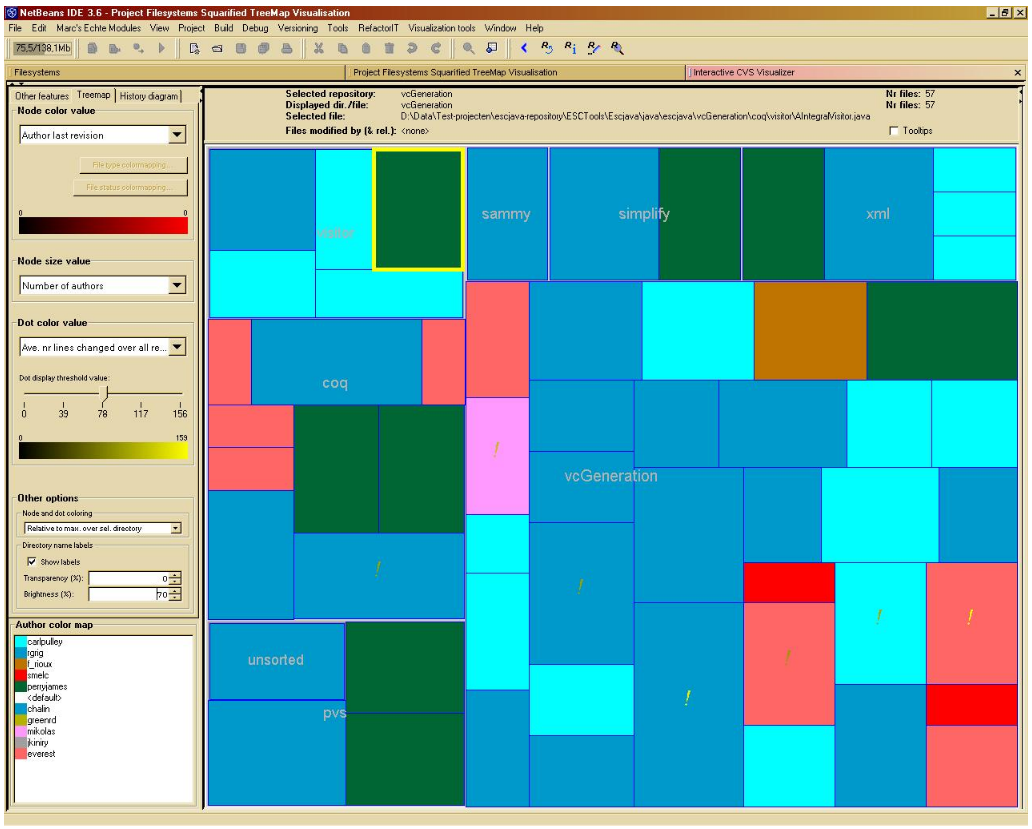
☐ Tooltips

Author color map

🟩	raboof
🟩	alx-gforge
🟥	carlpulley
🟩	f_rioux
🟥	smelc
🟪	perryjames
🟨	cok
⬜	<default>
🟦	greenrd
⬜	mikolas
🟥	jkiniry
🟦	rgrig
🟦	booms
⬜	davidcok
🟧	chalin

Figure 6.1: Example of how the visualizer helps to answer the question "*How important is this file with respect to the evolution of the repository?*". The number of revisions of a file is mapped to the treemap node color, the number of authors that have worked on the file to the node size and the average number of lines changed aggregated over all revisions to the exclamation mark color. The selected file (yellow border), Main.java, is an example of an important file. Its node size is large indicating that a large number of authors has worked on the file. Its node color is bright red indicating that the file has been changed quite often. Finally, an exclamation mark is shown in the node while the threshold value for displaying an exclamation mark is set to 65 lines changed. The exclamation mark in the node therefore indicates that the average number of lines changed aggregated over all revisions is at least 65. Thus on average the changes to the file were quite large.

During this project it is assumed the question "*Who knows most about this file/directory?*" can be answered in three ways (see section 3.2): by finding the author of the last revision of the file, the author who committed most revisions of the file or the author who changed most lines of the file. Consequently, this question can be answered using the visualizer by selecting either "Author of last revision", "Author who committed most revisions" or "Author who changed most lines" for the treemap node color. An example of this is shown in figure 6.2.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation

Selected repository: vcGeneration
Displayed dir./file: vcGeneration
Selected file: D:\Data\Test-projecten\escjava-repository\ESCTools\Escjava\java\escjava\vcGeneration\coq\visitor\AIntegralVisitor.java
Files modified by (& rel.): <none>
Nr files: 57
Nr files: 57

Node color value: Author last revision
Node size value: Number of authors
Dot color value: Ave. nr lines changed over all re...
Node and dot coloring: Relative to max. over sel. directory
Show labels: ✓
Transparency (%): 0
Brightness (%): 70

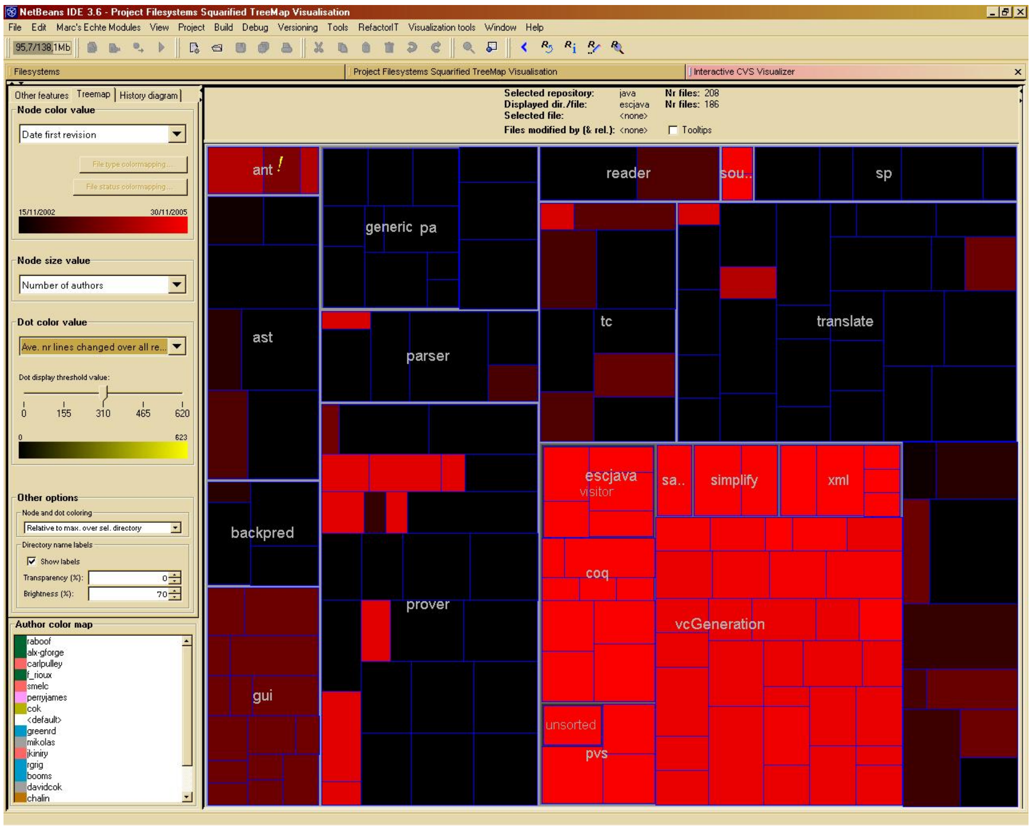
Author color map

🟦	carlpulley
🟦	rgrig
🟫	f_rioux
🟥	smelc
🟩	perryjames
⬜	<default>
🟦	chalin
🟨	greenrd
🟪	mikolas
⬜	jkiniry
🟥	everest

File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualization tools Window Help
75,5/138,1Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Interactive CVS Visualizer
Other features
Treemap
History diagram
File type colormapping...
File status colormapping...
0
0
Dot display threshold value:
0
39
78
117
156
0
159
Other options
Directory name labels
Tooltips
visitor
sammy
simplify
xml
coq
vcGeneration
unsorted
pvs

Figure 6.2: Example of how the visualizer helps to answer the questions "*Who knows most about this file/directory?*" and "*Who introduced this bug in this file?*". The author of the last revision is mapped to the treemap node color. In answering the former question, it is assumed the author of the last revision knows most about a file. In answering the latter question it is assumed the bug has been present in the file only recently and hence was introduced in the last revision. Consequently, the answer to both questions for the selected file (yellow border) is "perryjames".

The question "*Who introduced this bug in this file?*" can be answered in the same way, as it is assumed that either the author of the last revision of the file, the author who committed most revisions of the file or the author who changed most lines of the file introduced the bug in the file (see section 3.2). Hence figure 6.2 also shows an example of how the visualizer helps to answer this question.
"*Which are the old parts of the project, which the new parts?*" can be answered in two ways, either by considering the commit date of the first revision of the file or the commit date of the last revision of the file. In the former case the user is interested in the date a file was added to the project. In the latter case he/she is interested in the date that the file was last modified. The visualizer can be used to answer the question in both ways by selecting either "Date of first revision" or "Date of last revision" to be mapped to the treemap node color. Figure 6.3 shows an example of this.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation

Filesystems | Project Filesystems Squarified TreeMap Visualisation | Interactive CVS Visualizer

Other features | Treemap | History diagram

Node color value: Date first revision

15/11/2002 – 30/11/2005

Node size value: Number of authors

Dot color value: Ave. nr lines changed over all re...

Dot display threshold value: 310 (0 – 620)

0 – 623

Other options

Node and dot coloring: Relative to max. over sel. directory

Directory name labels: ☑ Show labels

Transparency (%): 0

Brightness (%): 70

Author color map

🟩 raboof
🟩 alx-gforge
🟥 carlpulley
🟩 f_rioux
🟥 smelc
🟪 perryjames
🟨 cok
⬜ <default>
🟦 greenrd
⬜ mikolas
🟥 jkiniry
🟦 rgrig
🟦 booms
⬜ davidcok
🟧 chalin

Selected repository: java — Nr files: 208

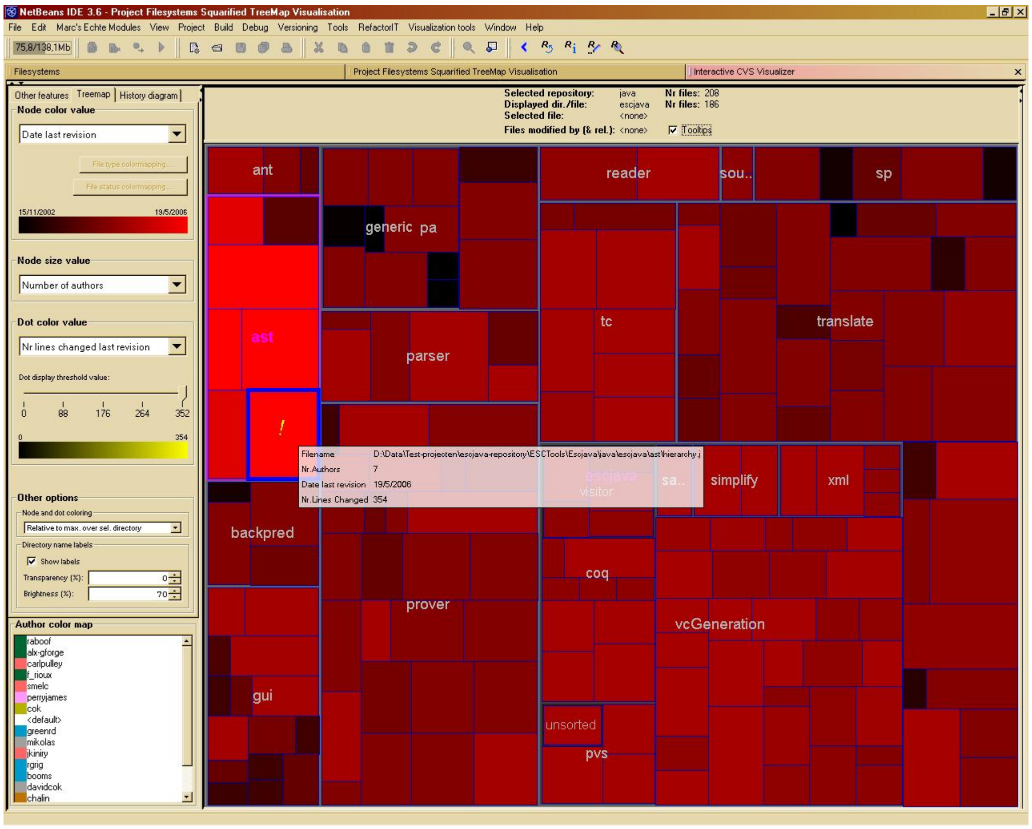
Displayed dir./file: escjava — Nr files: 186

Selected file: <none>

Files modified by (& rel.): <none> — ☐ Tooltips

Figure 6.3: Example of how the visualizer helps to answer the question "*Which are the old parts of the project, which the new parts?*". The commit date of the first revision of a file is mapped to the treemap node color. All nodes in the directory "vcGeneration" are bright red in contrast to (most of) the nodes in the other directories. This indicates that the files in "vcGeneration" have been added to the project much more recently than the rest of the files.

The question "*What is the probability that there will be a new revision of this file soon?*" can be answered by visualizing the commit date of the last revision of the file and the number of lines changed in that revision (see section 3.2). An example of this is shown in figure 6.4.
The user question "*Do I have the latest version of this file?*" can be answered by considering the CVS file status attribute of the local working copy of the file. If the value of this attribute is "Up-to-date" the working copy is a check out of the last revision of the file (see section 3.1) and the answer to the question is "yes". The visualizer can be used to answer this question by selecting "CVS file status" to be mapped to the treemap node color. An example is shown in figure 6.5.
An answer to the question "*What part of the total work did this person do and when did he perform it?*" can be found by considering which files in the repository have been modified by the person, which revisions he/she has committed and when these revisions have been committed. The visualizer can be used to answer the question by selecting the author in the list of authors of all files (see


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualization tools Window Help
75,8/138,1Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Interactive CVS Visualizer
Other features
Treemap
History diagram
File type colormapping...
File status colormapping...
ant
generic pa
reader
sou..
sp
ast
tc
translate
parser
!
escjava
visitor
sa..
simplify
xml
backpred
coq
prover
vcGeneration
gui
unsorted
pvs

Node color value: Date last revision

15/11/2002 19/5/2006

Node size value: Number of authors

Dot color value: Nr lines changed last revision

Dot display threshold value: 0 88 176 264 352

0 354

Other options

Node and dot coloring: Relative to max. over sel. directory

Directory name labels

☑ Show labels

Transparency (%): 0

Brightness (%): 70

Author color map

🟩 raboof
🟩 alx-gforge
🟥 carlpulley
🟩 f_rioux
🟥 smelc
🟪 perryjames
🟨 cok
<default>
🟦 greenrd
⬜ mikolas
🟥 jkiniry
🟦 rgrig
🟦 booms
⬜ davidcok
🟧 chalin

Selected repository: java Nr files: 208
Displayed dir./file: escjava Nr files: 186
Selected file: <none>
Files modified by (& rel.): <none> ☑ Tooltips

Filename	D:\Data\Test-projecten\escjava-repository\ESCTools\Escjava\java\escjava\ast\hierarchy.j
Nr.Authors	7
Date last revision	19/5/2006
Nr.Lines Changed	354

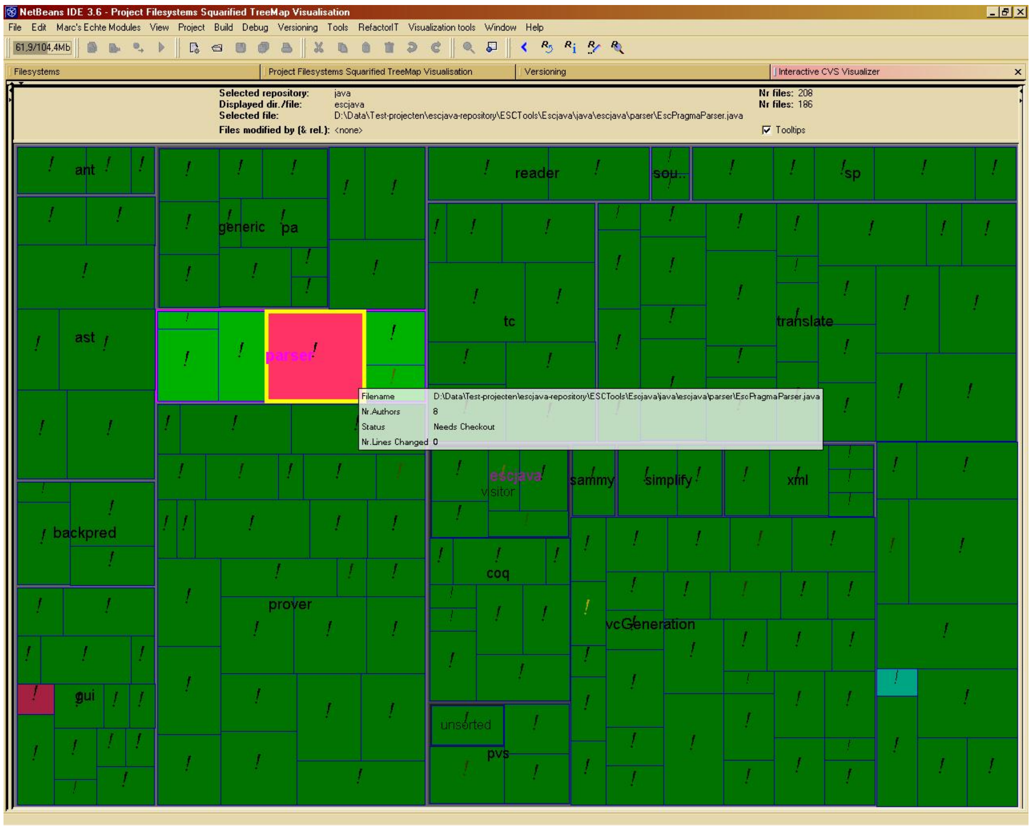
Figure 6.4: Example of how the visualizer helps to answer the question "*What is the probability that there will be a new revision of this file soon?*". The commit date of the last revision of a file is mapped to the treemap node color, the number of lines changed in that revision to the exclamation mark color. The selected file node (blue border) is bright red indicating that the file has been changed very recently. It also contains a bright yellow exclamation mark indicating that these recent changes were quite large. A large recent change to a file is assumed to indicate a high probability for a new change since bugs might have been introduced in the last change (see section 3.2). Consequently, there is a high probability that there will be a new revision of the selected file soon.

section 4.4). Figure 6.6 shows an example of how this question is answered.

"*What is the productivity of each author over time?*" can be answered in two ways, depending on which definition of "amount of work done during a certain time period" (see section 3.2) is used:

- by determining how many revisions each person has committed during each time period
- by determining how many lines each person has changed during each time period

In the visualizer the answer to this question can be found by grouping the revisions in the history diagram by author. The productivity of an author over time can be determined by inspecting the height of the revision group corresponding to that author in each of the bars. A large group indicates great productivity


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation
File Edit Marc's Echte Modules View Project Build Debug Versioning Tools RefactorIT Visualization tools Window Help
61,9/104,4Mb
Filesystems
Project Filesystems Squarified TreeMap Visualisation
Versioning
Interactive CVS Visualizer
Tooltips
ant
generic
pa
reader
sou...
sp
tc
translate
ast
parser
escjava
visitor
sammy
simplify
xml
backpred
coq
prover
vcGeneration
gui
unsorted
pvs

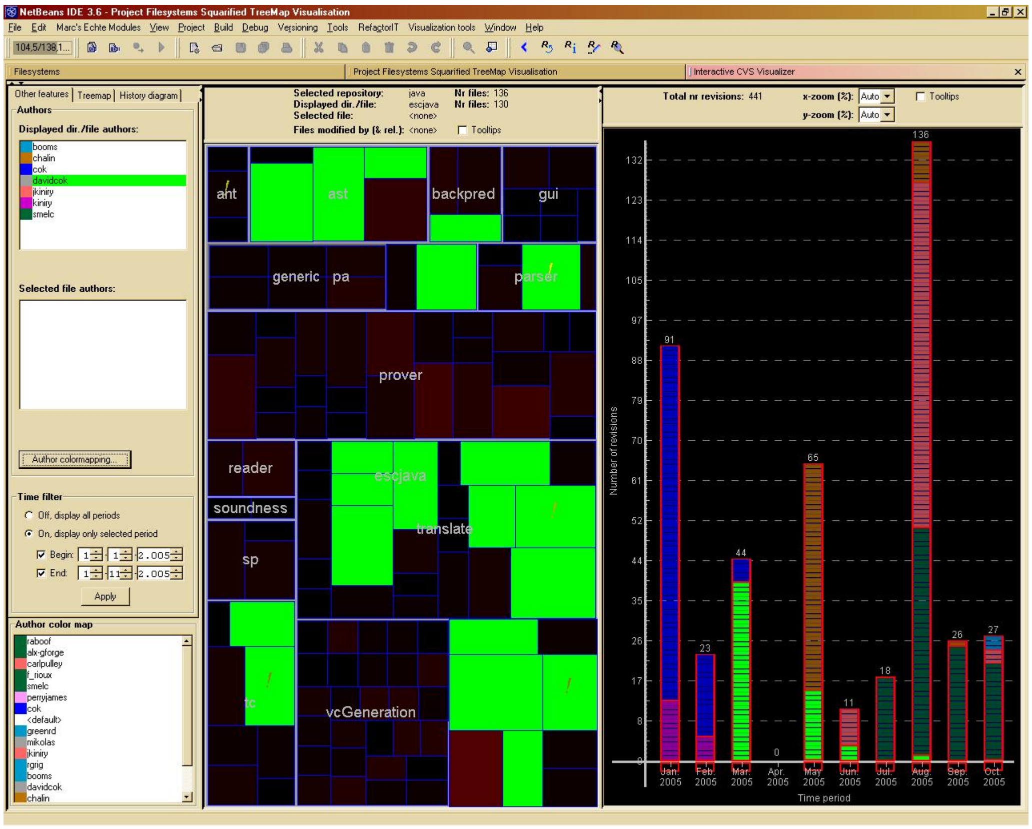
Selected repository: java
Displayed dir./file: escjava
Selected file: D:\Data\Test-projecten\escjava-repository\ESCTools\Escjava\java\escjava\parser\EscPragmaParser.java
Files modified by (& rel.): <none>

Nr files: 208
Nr files: 186

Filename	D:\Data\Test-projecten\escjava-repository\ESCTools\Escjava\java\escjava\parser\EscPragmaParser.java
Nr.Authors	8
Status	Needs Checkout
Nr.Lines Changed	0

Figure 6.5: Example of how the visualizer helps to answer the question "*Do I have the latest version of this file?*". The CVS file status of the local working copy of each file is mapped to the treemap node color. The answer to the question for the selected file (yellow border) is "no". The color pink corresponds to the status "Needs Checkout" which means someone else has committed a newer revision of this file to the repository (see [9]).

during the corresponding time period. The visualizer supports both methods mentioned above: the history diagram node height is either constant or represents the number of lines changed in the corresponding revision, depending on which value the user has selected. An example is shown in figure 6.7.
How the answer to the user question "*During which time periods was most of the work done on the project?*" can be determined depends on which definition of "amount of work done during a certain time period" (see section 3.2) is used. It can either be determined by considering the total number of revisions committed during each time period, or by considering the total number of lines changed during each time period. The visualizer supports both methods depending on which attribute has been selected to be mapped to the history diagram node height: constant in the former case, the number of lines changed in the latter case. The answer to the question can be found by determining the time periods corresponding to the bars which are highest. An example is shown in figure 6.7.
Finally, to answer the question "*Who worked hardest during a certain time period?*" either the author who committed most revisions during that period or



Displayed dir./file authors:

🟦 booms
🟧 chalin
🟦 cok
⬜ davidcok
🟥 jkiniry
🟪 kiniry
🟩 smelc

Selected file authors:

Time filter

( ) Off, display all periods
(•) On, display only selected period

☑ Begin: 1 1 2.005
☑ End: 1 11 2.005

raboof
alx-gforge
carlpulley
f_rioux
smelc
perryjames
cok
<default>
greenrd
mikolas
jkiniry
rgrig
booms
davidcok
chalin

Selected repository: java
Displayed dir./file: escjava
Selected file: <none>
Files modified by (& rel.): <none>
Nr files: 136
Nr files: 130
☐ Tooltips

Total nr revisions: 441
x-zoom (%): Auto
y-zoom (%): Auto
☐ Tooltips

Figure 6.6: Example of how the visualizer helps to answer the question "*What part of the total work did this person do and when did he perform it?*". The author "davidcok" has been selected in the list of authors of all files in the displayed directory. In the treemap, the node color of all files this author has worked on is bright green. In the history diagram, the node color of all revisions committed by the author is also bright green. Consequently, the part of the total work done by "davidcok" is displayed in bright green. In the history diagram the user can see when the author committed the revisions and hence when he performed the work.

the author who changed most lines during that period needs to determined, depending on which definition of "amount of work done during a certain time period" is used (see section 3.2). To answer the question using the visualizer, the revisions in the history diagram must be grouped by author. The user must choose which attribute to map to the history diagram node height: either a constant value or the number of lines changed in the corresponding revision. The answer to the question can be found by determining the author corresponding to the largest revision group in the time period under consideration. To make this process easier the user can also choose to sort the revision groups in each bar by size.

Chapter 3 mentions that the user questions can be parameterized in two ways: 1) the questions can be applied to a restricted set of files and 2) they can be applied to a restricted time period. It is also mentioned that the visualizer must provide ways to answer these parameterized questions. In the visualizer this can be done by zooming (see section 4.3) and the use of the time filter (see section

4.4), respectively.
In summary, all user questions mentioned in chapter 3 can be answered by the visualizer. Consequently it is concluded that the project goal has been accomplished.

6.2 Comparison to other tools

To further evaluate the visualizer it must be compared to other similar tools. To find other version control visualization tools a search was conducted on the Internet. The following tools were found:

• Seesoft	(AT&T Bell Lab, 1991)
• VRCS	(Univ. of Electro-Communications, 1997)
• Beagle	(SWAG, Univ. of Waterloo, 2002)
• Palantír	(Univ. of California, Irvine, 2002)
• Version Tree	(sourceforge.net, 2003)
• Xia	(CHISEL, Univ. of Victoria, 2003)
• CVSscan	(TU/e, 2005)

Below a short description is given of each of these tools. Each tool is described using the properties as mentioned by the author of the tool.
SeeSoft [7] visualizes various textual aspects of evolving large and complex software systems. Such aspects involve software complexity metrics, number and scope of modifications, number and types of bugs and dynamic program slices. SeeSoft is implemented using the information from version control systems.
VRCS [12] is a visualization system for the Unix Revision Control System (RCS). It visualizes RCS data and enables users to perform RCS commands using a graphical user interface (GUI).
Beagle [17] is a research tool for exploring software evolution. It incorporates techniques from reverse engineering, visualization and database and builds a platform where the users can navigate through and annotate software releases so that a better understanding can be achieved.
Palantír [5] is a system that aims to bring increased levels of awareness to configuration management (CM) systems. It supports developers by informing them continuously of which artifacts are being changed and the impact and severity of those changes. To do so, Palantír provides advanced graphical views, which are based on check-in / check-out information and show the changes that are being made to the artifacts that are of interest to the developer. Changes to an artifact are tracked at real time, and the severity and impact of each change are calculated and visualized.
Version Tree [4] displays a graphical tree of the revision history of a file.
Xia [2] is a version control visualization tool, used to browse and explore the software version history and associated human activities. It obtains its data from CVS. Using SHriMP views (see [2]) it visualizes a number of revision attributes.
CVSscan [19] is a visual tool which supports program and process understanding for the maintenance of large software projects. It retrieves its data from CVS. It focuses on the text content of revisions and changes in these texts.
A mutual comparison of the first six version control visualization tools can be

found at [1].

A number of criteria are defined in determining whether the tools are suitable for comparison to the visualizer module created during this project. These are the following:

1. The tool must visualize CVS data
2. The tool must visualize parts of CVS data that are also visualized by the visualizer, e.g. the data available on the CVS server
3. Executable code of the tool must be available

The first two criteria are considered to be important because to properly compare two tools, those tools should use the same data. The last criterion has been defined because without executable code of a tool it is difficult to assess in detail how the tool works. Inferring from a document describing the tool how it actually works can be difficult, if not impossible.
Based on these criteria the following is concluded about the tools:
Seesoft and VRCS fail to meet the last criterion. For both tools only a document describing it is available. These tools are therefore not suitable for comparison.
Beagle does not use CVS or another traditional Version Control System, such as Subversioning, to retrieve its data (it builds its own database of versions). Beagle fails to meet the first criterion and is consequently not used for comparison.
Palantír focuses on changes in multiple working copies of a file on different client computers, not e.g. on the data on the CVS server. The visualizer does not consider CVS data on different client computers. Hence, Palantír fails to meet the second criterion and is discarded for comparison.
Version Tree does meet all criteria. However, it is a very minimal visualization offering little functionality. It only shows a tree view of the revisions of a single file. Therefore it is not considered useful to compare the visualizer and the Version Tree.
Xia meets all criteria and offers a great amount of functionality. It is considered very suitable for comparison.
Finally, CVSscan focuses on the text content of revisions and differences between these texts. Since the visualizer ignores this data, CVSscan fails to meet the second criterion and is not used for comparison.
In summary, only one of the tools found is considered suitable for comparison: Xia. Hence the visualizer is only compared to this tool.

6.2.1 Comparison

In this section a comparison is made between the visualizer and Xia. The comparison is based on the user questions mentioned in chapter 3. Section 6.1 describes how the visualizer answers these questions. In this section it is described if and how Xia answers them. Advantages and disadvantages of the tools with respect to the user questions are given.

First a short description of Xia is given. Xia visualizes data from a CVS repository using SHriMP (Simple Hierarchical Multi-Perspective) views (see [2]).

SHriMP views are used to display the hierarchy of the repository. Each revision in the repository is mapped to a single node in SHriMP. Figure 6.8 shows an example of a repository visualized by Xia. Each node is displayed as a colored rectangle. Rectangles can be nested. The views allow the data to be displayed using a number of layouts for the nodes, a.o. a treemap layout, a spring layout and a radial layout. In Xia the SHriMP views are extended with an Attribute Panel (see [2]). The attribute panel is shown at the top of figure 6.8. This panel allows the user to map CVS data attributes to the color of the nodes and to the tooltip. Attributes the user can choose from are a.o. the number of revisions of a file and the author of most revisions of a file. A detailed description of how Xia visualizes the CVS data can be found in [2].

In the following a CVS repository containing the source code and documentation of DNAVis [14] is used to show examples of how Xia answers the user questions mentioned in chapter 3. DNAVis is a flexible tool that allows both browsing and comparing annotated DNA sequences interactively and in real-time. It has been developed in a joined effort by Wageningen University and Research Center and the University of Technology Eindhoven (TU/e). The repository contains ± 250 files / ± 1200 revisions. Three authors have worked on the project during a time period of about two years. Unfortunately the ESC/Java repository described in section 6.1 cannot be used in the prototype of Xia that is available for testing. The prototype contains errors which make it impossible to retrieve and visualize all data from the repository.
In answering the question "*How important is this file with respect to the evolution of the repository?*" the following data is considered relevant: how often the file has been changed, how large each change was and how many authors have made these changes. In Xia only the first data attribute mentioned here is available. Xia allows the "Number of changes", i.e. the number of revisions of a file, to be mapped to the node color and to the tooltip. Figure 6.8 shows an example of this.
Exact values can only be obtained when the attribute is mapped to the tooltip. No information concerning the other attributes mentioned here is available in Xia. The visualizer, however, visualizes all three attributes simultaneously (see section 6.1) answering the question in more detail.
The question "*Who knows most about this file/directory?*" is answered in Xia in a way similar to how the visualizer answers this question (see section 6.1). Either the attribute "Author with most changes" or the attribute "Author with last change" is mapped to the node color. Figure 6.9 shows an example. "Author with most changes" corresponds to the author who committed most revisions of the file, "Author with last change" to the author of the last revision. The visualizer, however, offers a third way to answer this question: by mapping the attribute "author who changed most lines" to the treemap node color (see section 6.1). Hence, the visualizer answers the question in more detail.
As the question "*Who introduced this bug in this file?*" is assumed to be answered in the same way as the previous question (see section 6.1), the visualizer also answers this question in more detail than Xia.
In Xia the question "*Which are the old parts of the project, which the new parts?*" is answered in a way similar to how the visualizer answers the question (see section 6.1). The attribute "Last commit date", corresponding to the commit date of the last revision, is mapped to the node color. In figure 6.10 an

example is shown of how Xia answers this question. The visualizer, however, offers a second way to answer the question: the commit date of the first revision can also be mapped to the treemap node color (see section 6.1). Hence, again the visualizer answers the question in more detail than Xia.
The user question "*What is the probability that there will be a new revision of this file soon?*" cannot be answered by Xia. The number of lines changed in the last revision, used to determine this probability in the visualizer (see section 6.1), is not available in Xia. Furthermore, no other criteria can be defined according to which Xia can answer the question.
The answer to the question "*Do I have the latest version of this file?*" depends on the CVS file status of the local working copy of the file. This data is not available in Xia. Therefore Xia cannot answer this question.
In answering the question "*What part of the total work did this person do and when did he perform it?*" one is interested in which files a specific author has modified, which revisions he/she has committed and when those revisions were committed. However, in Xia it is not possible to determine this information. Consequently the question cannot be answered by Xia.
As with the previous question, answering "*What is the productivity of each author over time?*" involves determining what work an author has done and when he/she has performed it. Hence, also this question cannot be answered by Xia.
In Xia it is impossible to determine how many revisions were committed during a certain time period. Hence, it is not possible to determine the answer to the question "*During which time periods was most of the work done on the project?*".
Finally, because of the same reason the question "*Who worked hardest during a certain time period?*" cannot be answered by Xia. It is impossible to determine which revisions were committed during a certain time period. Consequently, it is not possible to determine which author committed most of those revisions.
In summary, it is concluded that Xia can answer some of the user questions. However, the visualizer always gives a more complete answer to those questions. Some questions cannot be answered at all by Xia. On the other hand there are questions, not mentioned in chapter 3, Xia can answer but which cannot be answered by the visualizer. These questions are described in the next paragraph.

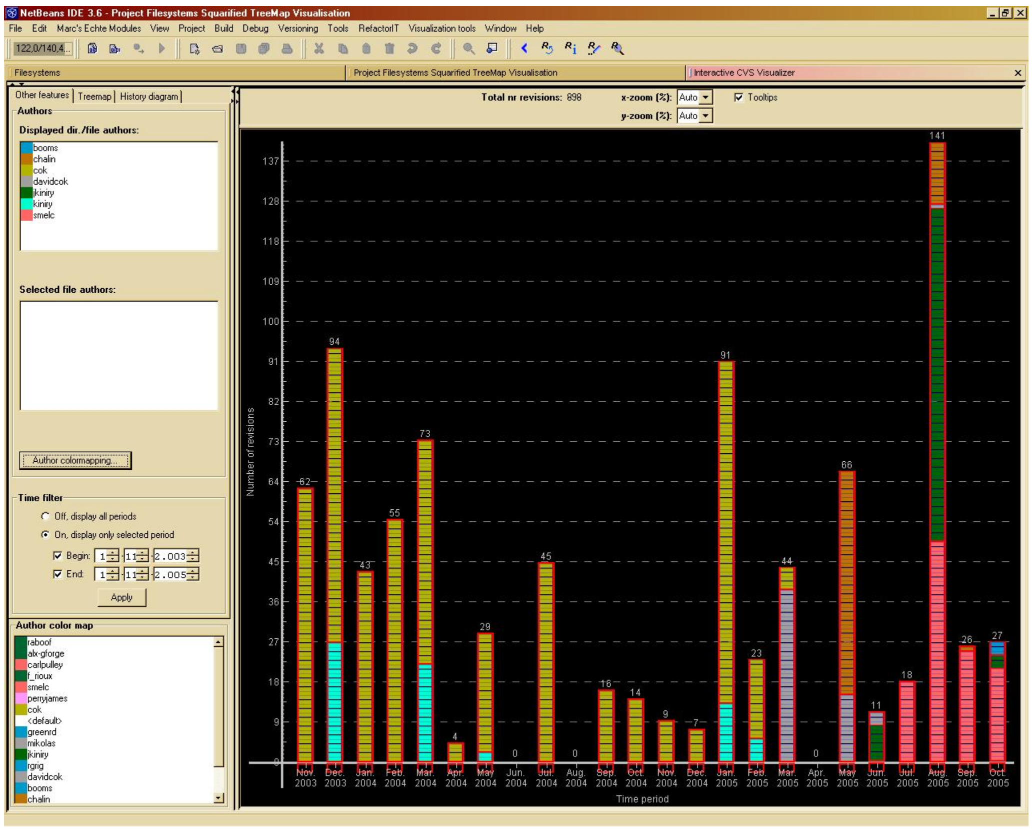
Additional questions answered by Xia

Xia was developed to answer the following six questions regarding a CVS repository (see [2]):

1. What happened since I last worked on the project (types of events, such as new file added, file modified, etc.)?
2. Who made this happen?
3. Where did this take place (location of the new file, change, deletion, etc.)?
4. When did this happen?
5. Why were these changes made (what is the rationale of the designer(s) who made the change)?
6. How has a file changed (exact details of the change, as well as relationship to other files)?

How Xia answers these questions is described in [2]. The visualizer can help to answer the first question by mapping the CVS file status of the local working copy of each file to the treemap node color. If e.g. another user of the repository has committed a newer revision of a file since the local user last worked on the project, the status of the local working copy of that file has changed. The visualizer displays this change using another color for the node corresponding to the file.
The second question can be answered by the visualizer by mapping the author of the last revision of the file to the treemap node color, since this is the person who last changed the file.
The third question is also answered by the visualizer: the user can determine the location of the changed file by looking at the position of the corresponding node in the treemap.
The answer to the question "*When did this happen?*" can be found using the visualizer by mapping the date of the last revision of each file to the treemap node color, since this is the date when the file was last changed.
The fifth and sixth questions cannot be answered by the visualizer. In Xia they are answered primarily by looking at the "log message" attribute or the text content of the revisions. This data, however, is ignored in the visualizer (see sections 3.1 and 3.2).

It can be concluded that there is some overlap in functionality between Xia and the visualizer. However, there are questions which can be answered by the visualizer but not by Xia, and vice versa. This makes the tools difficult to compare. Hence, no conclusion can be made about which tool is best.


NetBeans IDE 3.6 - Project Filesystems Squarified TreeMap Visualisation

Total nr revisions: 898

x-zoom (%): Auto

y-zoom (%): Auto

☑ Tooltips

Time filter

○ Off, display all periods

● On, display only selected period

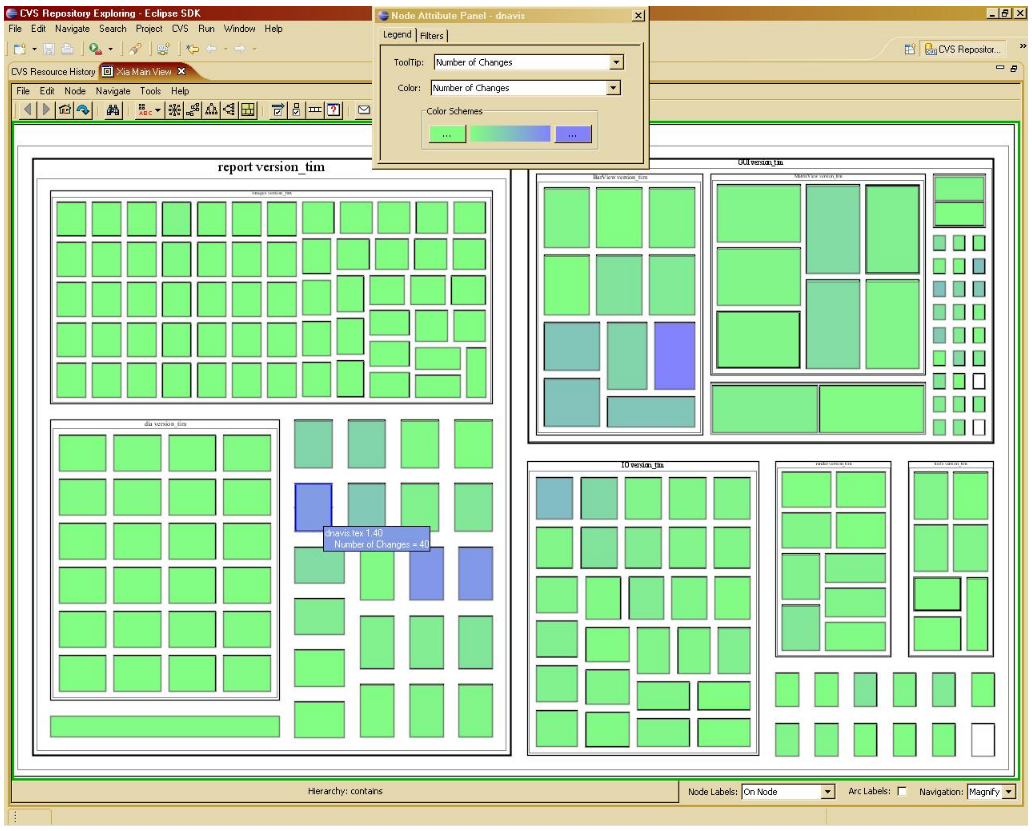
☑ Begin: 1 11 2.003

☑ End: 1 11 2.005

Displayed dir./file authors: booms, chalin, cok, davidcok, jkiniry, kiniry, smelc

Author color map: raboof, alx-gforge, carlpulley, f_rioux, smelc, perryjames, cok, <default>, greenrd, mikolas, jkiniry, rgrig, davidcok, booms, chalin

Figure 6.7: Example of how the visualizer helps to answer the questions "*What is the productivity of each author over time?*" and "*During which time periods was most of the work done on the project?*". In the history diagram, the revisions are grouped by author. The revision node height is constant. The productivity of an author over time can be determined by inspecting the height of the revision group corresponding to that author in each of the bars. Here it is assumed the productivity of an author during a time period is equal to the number of revisions he/she has committed during that period. E.g. author "cok" has been active in the period from Nov. 2003 till Mar. 2005 during which he was most productive from Nov. 2003 to Mar. 2004 and in Jan. 2005. The second question is answered by determining the time periods corresponding to the highest bars in the history diagram. Most of the work was done on the project during the months Dec. 2003, Jan. 2005 and Aug. 2005.



ToolTip: Number of Changes

Color: Number of Changes

dnavis.tex 1.40
Number of Changes = 40

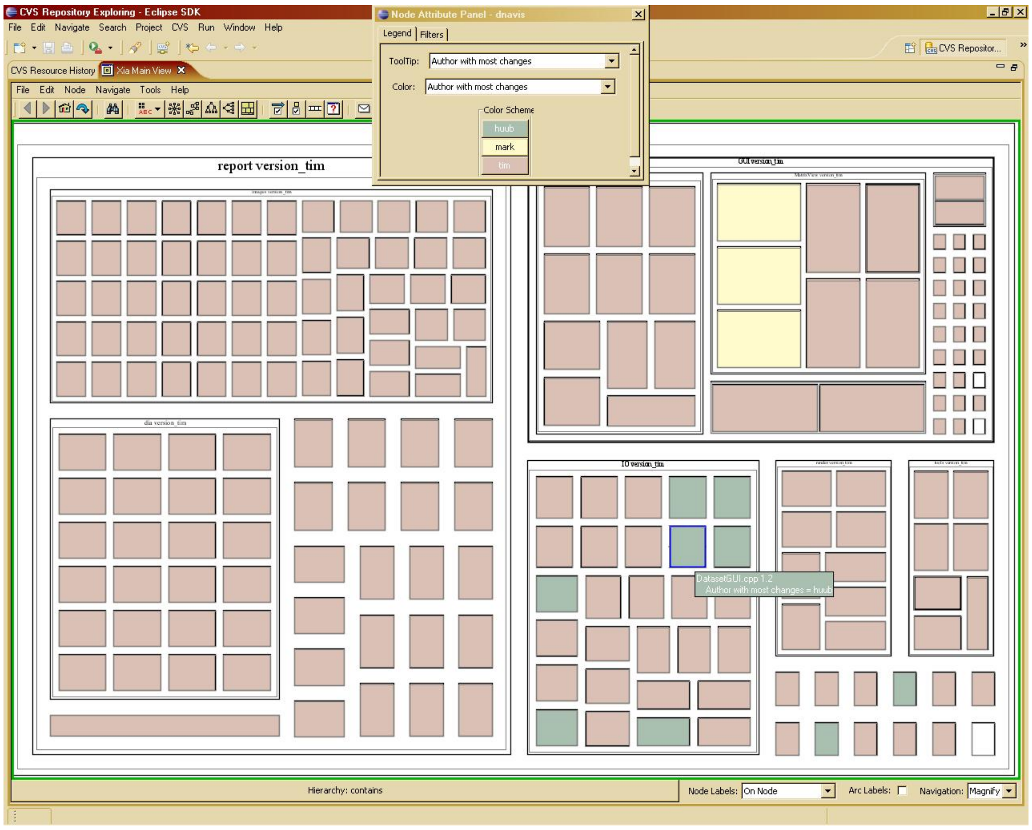
Hierarchy: contains

Node Labels: On Node

Arc Labels: ☐

Navigation: Magnify

Figure 6.8: Example of how Xia helps to answer the question "*How important is this file with respect to the evolution of the repository?*". A squarified treemap layout is used for all nodes that are displayed. The attribute "Number of changes", i.e. the number of revisions of a file, is mapped to both the node color and the tooltip (see node attribute panel at the top of the figure). The color green indicates a small number of changes, blue a high number of changes. In answering the question it is assumed that a file which has been changed very often is an important file. The selected file (blue border) is therefore important. The tooltip indicates it has 40 revisions.


CVS Repository Exploring - Eclipse SDK
Node Attribute Panel - dnavis
Legend
Filters
Color Scheme
huub
mark
tim
report version_tim
DatasetGUI.cpp 1.2
Author with most changes = huub
Hierarchy: contains

ToolTip: Author with most changes

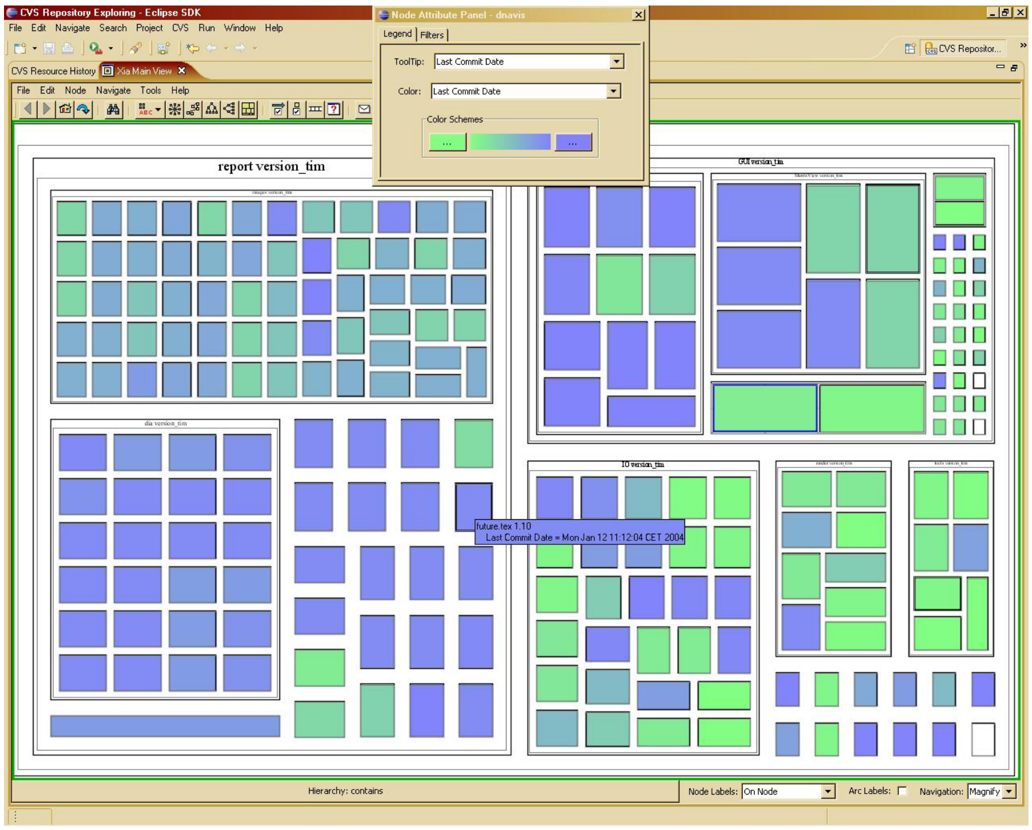
Color: Author with most changes

Node Labels: On Node

Arc Labels:

Navigation: Magnify

Figure 6.9: Example of how Xia helps to answer the questions "*Who knows most about this file/directory?*" and "*Who introduced this bug in this file?*". A squarified treemap layout is used for all nodes that are displayed. The attribute "Author with most changes", i.e. the author who committed most revisions of a file, is mapped to both the node color and the tooltip. The colormapping used for the author names is shown in the node attribute panel at the top of the figure. In answering the former question it is assumed the author who changed a file most often knows most about the file. In answering the latter question it is assumed the bug has been present in the file for a long time and therefore the author who changed the file most often introduced this bug. Consequently, the answer to both questions for the selected file (blue border) is "huub". This value is also shown by the tooltip.


CVS Repository Exploring - Eclipse SDK
File Edit Navigate Search Project CVS Run Window Help
CVS Resource History
Xia Main View
CVS Repositor...
File Edit Node Navigate Tools Help
Node Attribute Panel - dnavis
Legend
Filters
Color Schemes
report version_tim
GUI version_tim
IO version_tim
dia version_tim

ToolTip: Last Commit Date

Color: Last Commit Date

future.tex 1.10
Last Commit Date = Mon Jan 12 11:12:04 CET 2004

Hierarchy: contains

Node Labels: On Node

Arc Labels:

Navigation: Magnify

Figure 6.10: Example of how Xia helps to answer the question "*Which are the old parts of the project, which the new parts?*". A squarified treemap layout is used for all nodes that are displayed. The attribute "Last commit date", i.e. the commit date of the last revision, is mapped to both the node color and the tooltip. The color green indicates a date a long time ago, blue a more recent date. All nodes in the directory "dia" (bottom left corner of the figure) are blue. This indicates these files have been modified more recently than most of the other files.

Chapter 7

Conclusions

This chapter describes a number of conclusions on the project. First there is some reflection on the goals set out at the beginning of the project. Next, there is a brief recap of the conclusions reached in the evaluation. Finally, recommendations for further research are given.

7.1 Project goals

There were two goals in this project (see chapter 2):

- To design an interactive visualization of CVS data available in NetBeans.
- To implement this visualization as a NetBeans module and consequently to integrate the implementation in the IDE.

The visualization designed during the project is called "the visualizer". The user can interact with the visualizer e.g. by selecting files or zooming in on a directory (see section 4.3). Answering the user questions mentioned in chapter 3 was chosen to be the concrete goal of the visualizer. From the evaluation in chapter 6 it can be concluded that the visualizer answers all these questions. The prototype of the visualizer created during this project retrieves its data directly from the NetBeans IDE. Hence, from the above it can be concluded that the first project goal has been reached. Furthermore, the prototype was implemented as a NetBeans module and henceforth integrated in the IDE. Therefore the second goal has also been accomplished.

7.2 Evaluation of the visualizer

As mentioned in the previous section, from the evaluation it is concluded that the visualizer answers all user questions given in chapter 3. It can also be concluded that the functionality of the tool is to a great extent unique. Only one other similar tool could be found on the internet. Although this tool, Xia, can answer some of the user questions, it does so in less detail.

7.3 Recommendations for further research

The visualizer is designed to answer the user questions mentioned in chapter 3. From the evaluation it is concluded that the tool answers all these questions. However, some of the questions are answered by the visualizer in only one way though other methods can be defined to answer them. E.g. in answering the user question "*What is the probability that there will be a new revision of this file soon?*" only one criterion is used: a large recent change to a file is assumed to indicate a high probability for a new change (see section 3.2). Other criteria can be defined to determine the answer to this question. Another way to answer this question might e.g. be by looking at the frequency of changes to the file in the past. Incorporating other criteria in the tool enables it to answer the user questions in more detail.

In section 6.2.1 it is concluded that some questions which can be answered by Xia cannot be answered by the visualizer. These are the following:

- Why were these changes made (what is the rationale of the designer(s) who made the change)?
- How has a file changed (exact details of the change, as well as relationship to other files)?

Research showed that these are questions commonly asked by programmers (see [2]) and that they are therefore important to answer. The visualizer can be adapted to also answer these questions. This can be done by considering the text content of the revisions in the visualizer. Furthermore, the answer to the first question can be given by visualizing the "log message" attribute for each revision in the tooltip for the corresponding node in the history diagram. The second question can be answered e.g. by providing direct access to a diff between the text content of a revision A and the previous revision B of a file. Access could be provided in the history diagram by clicking the node corresponding to A while pressing the ALT key on the keyboard. This would then result in opening a new window showing the diff. The graphical diff viewer contained in NetBeans 3.6 could be used to create the contents of the new window.

The evaluation in chapter 6 does not contain a user test of the visualizer. To gain insight in the usefulness of the tool in practice it should be tested by a large number of users. The users can provide feedback on how they find the visualizer performs, i.e. whether they find the visualizer answers the user questions mentioned in chapter 3 in a satisfactory way. They may provide directions how to adapt the tool to answer the questions in another way, they find more suitable. Incorporating these changes in the visualizer improves it usefulness. Users may also have other important questions about CVS data they want the visualizer to answer. Modifying the tool to also answer those questions again improves it usefulness in practice.

The scalability of the visualizer is only limited. It can only be used to visualize small (software) projects. This is primarily caused by the design of the revision history diagram. Using the entire screen for the history diagram, only about 5000 revision nodes can be displayed. Consequently, the application of the visualizer is limited to repositories containing less than 5000 revisions in total. The scalability can be improved by aggregating nodes into a single node in the history diagram, e.g. by displaying only one node per revision group. However, if this change is made to the tool problems concerning file/revision history selection have to be solved.

The colors mentioned in sections 4.3 and 4.4 which are used to indicate temporary and persistent file/revision history selection (blue and yellow respectively), time period file/revision selection and author selection (both bright green) cannot be altered. This means the colormappings used to map attribute values to node colors (see sections 4.1 and 4.2) should not contain these colors, otherwise it becomes difficult to discern which nodes have been selected. Modifying the visualizer to allow the user to change the selection colors would make the tool more flexible and hence easier to use.

As mentioned in section 5.1.3 the CVS data cannot be obtained from NetBeans using the Open APIs. This means that the visualizer implementation created during the project might not work in newer (> 3.6) versions of NetBeans (testing showed it does not work in NetBeans 4.0). The visualizer module created during this project is intended to be a prototype. The code is only used for testing and is presumably discarded after the project has finished. Therefore, (NetBeans) version independence is not very important in the prototype. No actions were undertaken to adapt the Open APIs to retrieve the CVS data. If a production version of the visualizer module is created one should strive to obtain the CVS data using the Open APIs, possibly by sending a request to change the Open APIs to the proper channel of the NetBeans community, see e.g. [16, "Mailing lists"].
Finally, section 5.1.3 also mentions that data retrieval in the prototype implementation is quite slow. The section also gives some recommendations how to solve this problem. In a production version of the visualizer module this problem should be solved to increase its usability.

Bibliography

[1] *CHISEL – Version Visualization Tools Comparison.*
http://www.csc.uvic.ca/˜jiez/VVizComparison.htm.
last accessed on 15-05-2006.

[2] *Designing and Evaluating an Interactive Visualization Tool to Support Version Control of Software.*
Xiaomin Wu, Adam Murray, Margaret-Anne Storey and Rob Lintern
http://www.csc.uvic.ca/˜jiez/Xia.doc.
last accessed on 15-05-2006.

[3] *KindSoftware – ESC/Java2 home page.*
http://secure.ucd.ie/products/opensource/ESCJava2/.
last accessed on 29-05-2006.

[4] *Version Tree for CVS.*
http://versiontree.sourceforge.net/.
last accessed on 15-05-2006.

[5] Zahra Noroozi Anita Sarma and André van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *Proceedings of Twenty-Fifth International Conference on Software Engineering*, pages 444–454, May 2003.

[6] Moshe Bar and Karl Fogel. *Open Source Development with CVS.* Paraglyph Press, third edition, 2003.

[7] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[8] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.

[9] Per Cederqvist et al. *Version Management with CVS for cvs 1.12.9.* Free Software Foundation, Inc., 2004.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[11] Brian Scott Johnson. *Treemaps: visualizing hierarchical and categorical data.* PhD thesis, University of Maryland at College Park, 1993.

[12] Hideki Koike and Hui-Chu Chu. Vrcs: Integrating version control and module management using 3d graphics. In *Proceedings of 1997 IEEE/CS Symposium on Visual Languages (VL'97)*, 1997.

[13] Kees Huizing Mark Bruls and Jarke J. van Wijk. Squarified treemaps. In *Proceedings of Joint Eurographics and IEEE TCVG Symp. on Visualization (TCVG 2000)*, pages 33–42. IEEE Press, 2000.

[14] Tim H. J. M. Peeters Jarke J. van Wijk Mark W. E. J. Fiers, Huub van de Wetering and Jan-Peter Nap. Dnavis: interactive visualization of comparative genome annotations. *bioinformatics*, 22(3):354–355, 2006.

[15] Kathy Walrath Sharon Zakhour Mary Campione, Alison Huml. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* The Java Series. Addison Wesley Professional, second edition, February 2004.

[16] Simeon Greene Vaughn Spurlin Jack J. Woehr Tim Boudreau, Jesse Glick. *NetBeans: The Definitive Guide.* O'Reilly, first edition, October 2002.

[17] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.

[18] Peter van der Linden. *Just Java 2.* Prentice Hall, sixth edition, June 2004.

[19] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM Press.

[20] Colin Ware. Designing with a 2 1/2d attitude. *Information Design Journal*, 10(3):255–262, 2001.