Analyzing software repositories to understand software evolution

Marco D'Ambros[1], Harald C. Gall[2], Michele Lanza[1], and Martin Pinzger[2]

[1] Faculty of Informatics, University of Lugano, Switzerland
[2] Institute for Informatics, University of Zurich, Switzerland

1 Introduction

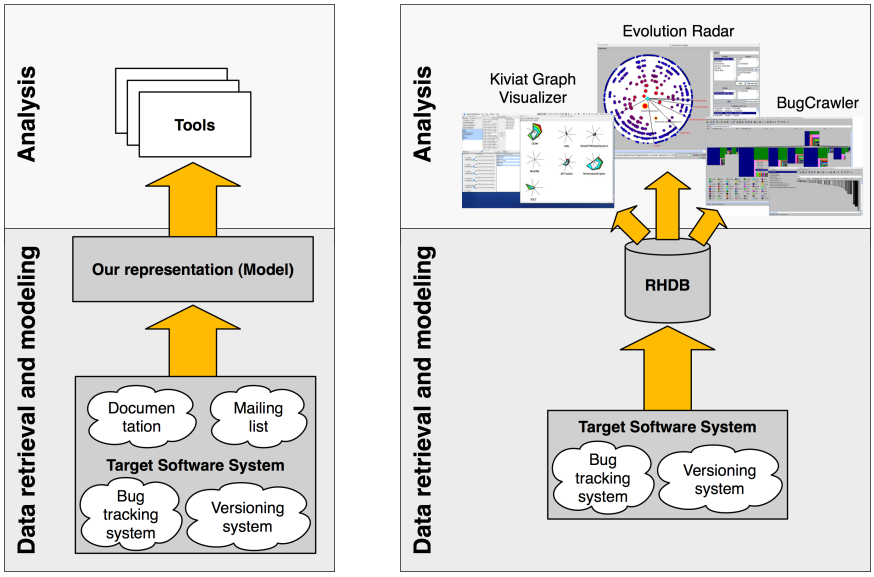
Software evolution analysis is concerned with software changes, their causes, and their effects. It uses all sources of a software system to perform a retrospective analysis. Such data comprises the release history with all the source code and the change information, bug reporting data, and data that can be extracted from the running system. In particular the analysis of release and bug reporting data has gained importance because they store a wealth of information valuable for analyzing the evolution of software systems.

While the recovery of the data residing in versioning systems such as CVS and SubVersion has become a well explored topic, the ultimate challenge lies in the recovered data and how it is possible to make sense of it. A number of techniques are being used, such as classical data mining and machine learning. One promising approach is visualization, because, if well used, it can break down the complexity of the data into pieces that can be better understood.

In this chapter we present different visualization techniques with the focus on visualizing *evolutionary* aspects of software systems. In particular, aspects concern the developer contributions and team structure, the change coupling between source code entities, and evolution of source code entities in terms of software metrics. Each technique allows the user to create different views with the objective to speed up the analysis and understanding of the software system under study. For instance, views show the as-implemented structure of the system enriched with evolutionary metrics information that highlight hot-spots. Such hot-spots refer to unstable implementation units, shortcomings in the implementation of certain features, or in the design and team structure. Pointing out these shortcomings and providing means to improve the current architecture, design, implementation, and team structure is the primary objective.

1.1 Mining Software Repositories

Figure 1(a) shows a generic schema of how to mine software repositories for studying software evolution.



(a) General Schema (b) Our approach

Fig. 1. The general approach and our customization to mining software repositories.

In the schema we identify three fundamental steps necessary for the final analysis of the data:

1. *Data modeling.* The first step of mining consists in creating a model of the evolving software system under analysis. Various aspects of the system and its evolution can be modeled: The last version of the source code, the history of files as recorded by the versioning system, several versions of the source code (*e.g.,* one per release), documentation, bug reports, developers mailing list archives, *etc.*
   While aspects such as source code or file histories have a direct mapping to the system, for others like bug reports or mailing list archives the useful information has to be filtered and linked to software artifacts. When designing the model it is important to consider the tradeoff between the amount of data to deal with (in the analysis phase) and the potential benefit this data can have. In other words, not all the aspects of the system evolution have to be considered, but only the ones which can support the addressing of a specific software evolution problem or set of problems.

2. *Data retrieval and processing.* Once the model is defined, a concrete instance of it has to be created. For this, we need to retrieve and process the information from the various data sources. According to the complexity of the model, the processing can include the parsing of the data (*e.g.,* source code, log files, bug report *etc.*), the application of matching techniques to link different data sources (*e.g.,* versioning system artifacts with bug reports [10, 4]), the reconstruction of information not recorded (*e.g.,* reconstruct commit information from CVS log files [30]) and the application of other techniques such as data mining.
3. *Analysis.* The analysis consists in using the data modeled and retrieved to tackle a software evolution problem or set of problems. According to the particular problem we are dealing with, we apply different techniques and approaches.

Figure 1(b) schematizes how we approach software evolution analysis through mining software repositories. As data sources we consider versioning system log files together with bug report data. We define a model describing an evolving software system based on these two data sources (data modeling). Given a system to analyze, versioning system log files and bug report data are parsed and a concrete instance of the model is created (data retrieval). All the models are then stored in a Release History Database, which works as a starting point for all the following analyses. For the analysis part we use different techniques and tools, aimed at addressing specific software evolution problems. These techniques extensively rely on visualization.

In the remainder of this chapter we describe our approach in detail. We first introduce the Release History Database, the model behind it and the way the model is created, *i.e.,* the database is populated. Then we present different types of software evolution analyses built on top of the Release History Database: Developers effort distribution, change coupling, trend analysis and hot-spot detection.

2 The Release History Database

When we refer to the history of a software artifact, we mean the way it was developed, how it grew or shrank over time, how many developers worked on it and to which extent. These kinds of information are recorded by versioning systems and can be reconstructed by parsing their log files. However, when we analyze the evolution of a software system our goal is to understand its architecture, the dependencies among its components and to detect hot-spots. To support this analysis, besides the history of the software artifacts additional information can be used, like problem (bug) reports. The problem is to link this data to the software artifacts, *e.g.,* which files are affected by a given bug?

In this section we present our approach for integrating versioning system information and bug report data and populating a Release History Database

[10, 4] (RHDB). We first introduce the versioning system and the bug tracking system from which we retrieve the data. Then we describe the model behind the RHDB, *i.e.,* the model of an evolving software system and we finally explain how we populate the database.

CVS and Bugzilla. CVS[3] has been the most used versioning system by the open source community over the last years. Currently it is being substituted by Subversion[4] (SVN). Many open source projects are developed using CVS or SVN as versioning system (examples of such projects are reported in Table 2). Our approach for populating the RHDB,is based on the versioning system log files, thus it can be applied on both CVS and SVN. For each versioned file, the log file contains the information recorded by the versioning system at commit-time: The version number (or revision), the timestamp of the commit, the author who performed the commit, the state (whether the file is still under development or removed), the number of lines added and removed with respect to the previous commit, the branches having the current version as root and the comments written by the author during the commit. Listing 1 shows a chunk of a CVS log file.

```
RCS file: /cvsroot/mozilla/js/src/xpconnect/codelib/Attic/mozJSCodeLib.cpp,v
Working file: codelib/mozJSCodeLib.cpp
head: 1.1
branch:
locks: strict
access list:
symbolic names:
        FORMS_20040722_XTF_MERGE: 1.1.4.1
        XTF_20040312_BRANCH: 1.1.0.2
keyword substitution: kv
total revisions: 6;     selected revisions: 6
description:
----------------------------
revision 1.1
date: 2004/04/19 10:53:08;  author: alex.fritze%crocodile-clips.com;  state: dead;
branches:  1.1.2;  1.1.4;
file mozJSCodeLib.cpp was initially added on branch XTF_20040312_BRANCH.
----------------------------
revision 1.1.4.2
date: 2004/07/28 09:12:21;  author: bryner%brianryner.com;  state: Exp;  lines: +1 -0
Sync with current XTF branch work.
----------------------------
...
----------------------------
revision 1.1.2.1
date: 2004/04/19 10:53:08;  author: alex.fritze%crocodile-clips.com;  state: Exp;  lines: +430 -0
Fixed bug 238324 (XTF javascript utilities).
=============================================================================
```

Listing 1. A CVS log file chunk of `mozJSCodeLib.cpp`.

Bugzilla[5] is a bug tracking system used in the open source community (Table 2 shows examples of open source projects using Bugzilla). Its core is a customizable database with a web interface which allows developers, testers as well as normal users to report and keep track of issues detected in the software

[3] http://www.nongnu.org/cvs/
[4] http://subversion.tigris.org/
[5] http://www.bugzilla.org

system. A typical bug report contains the following pieces of information: An *id* which univocally identifies the bug, the bug status composed of *status* (new, assigned, reopened, resolved, verified, closed) and *resolution* (fixed, invalid, wontfix, notyet, remind, duplicate, worksforme), the location in the system identified by the *product* and the *component*, the *operating system* and the *platform* on which the bug was detected, a *short description* of the problem and a list of comments about it (*long description*). Moreover, each bug refers to several people: The *reporter* who reported the bug, a person who is in charge to fix it (*assigned to*), quality assurance people who are responsible for ensuring that the software meets certain quality standards (*qa*), and a list of people who are interested in being notified of the bug fixing progress (*CC*).

Project	Versioning system	Bug Tracking system	Reference
Mozilla	CVS	Bugzilla	www.mozilla.org
Gnome	CVS and SVN	Bugzilla	www.gnome.org
KDE	SVN	Bugzilla	www.kde.org
Apache	SVN	Bugzilla	www.apache.org
OpenOffice	CVS	Bugzilla	www.openoffice.org
Eclipse	CVS	Bugzilla	www.eclipse.org

Table 1. Examples of open source project using CVS, SVN and Bugzilla.

2.1 The RHDB Model

Figure 2 shows the core of the RHDB model.

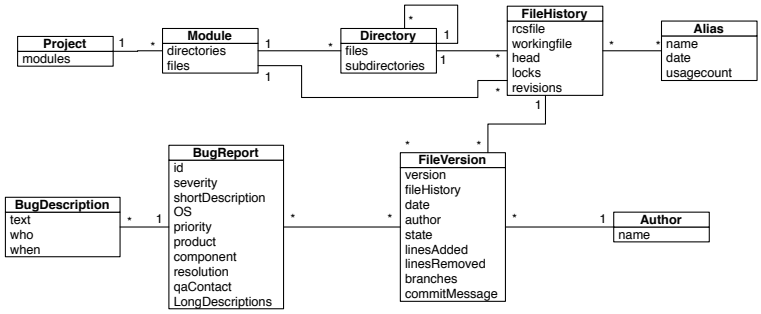


Fig. 2. The RHDB model.

In the model a CVS commit corresponds to a file version, having all the commit-related information: Version associated to the commit, date, author, state (exp or dead), lines added and removed with respect to the previous commit, branches associated with the version and the comment written by the author. A file history, which corresponds to the actual file in the file

system, is composed of a sequence of file versions, one per commit. It has a filename with (rcsfile) and without (workingfile) the entire path name. A file history can be associated to many aliases, used for tagging system releases. A project is composed of modules which contain directories and file histories. A directory can contain sub-directories and file histories. Finally, a file version can be associated to one or more bug reports. The relationship between bug reports and file versions is many to many, meaning that a file version (and therefore a file history) can be affected by many bugs and a bug can affect different file versions and file histories.

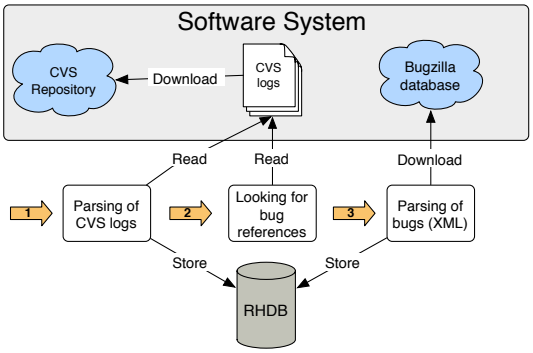


Fig. 3. The RHDB populating process.

2.2 Populating the RHDB

Figure 3 sketches the RHDB populating process. The user needs to enter the url of the CVS repository and of the Bugzilla database, and then the populating task (which according to the size of the system can take several hours) is executed in batch mode. The main steps of the process are:

- The last version of the system is retrieved by means of a cvs checkout command. Then, for each directory, the log file describing the history of the contained files is retrieved and parsed.
- For each file, the information about all its commits (its history) are stored in the database as well as a link to the actual file (to analyze the source code).
- Every time a reference to a bug is found in the comment field (the comment wrote by the author at commit time), the corresponding bug report is retrieved from the Bugzilla database, parsed and stored in the database, together with the link to the affected file. Since the link between CVS artifacts and Bugzilla problem report is not formally defined, we use regular expressions to detect bug references.

3 Software Evolution Analyses

The RHDB contains a concrete instance of our model of an evolving software system. This is the starting point from which, with the support of tools and techniques, we can do several types of analyses. Each technique we designed and each tool we implemented considers a particular perspective on software evolution, and addresses a particular goal or set of goals. In the following we present a number of software evolution and program comprehension problems and describe our techniques to tackle them.

3.1 Analyzing the effort of developers

The first software evolution problem we address concerns development effort. We want to answer questions like: How many developers worked on an entity? How was the effort distributed among them? Is there an owner of the entity, based on the code-ownership principle? Moreover, we also want to be able to categorize entities in terms of the "effort distribution". For an analyst or a project manager, the answers to these questions provide valuable information for a possible restructuring of the development teams.

Versioning systems record the information necessary to answer the previous questions: Each artifact has a list of versions corresponding to commits, and the list of authors who performed the commits[6]. The problem is how to represent and aggregate this large amount of low-level information[7] to get an insight into the team structure and to understand who are the responsible/s of a piece of software, scaling from a module down to the individual file.

Our approach is based on the "*Fractal Figure*" [7, 4] visualization, which encapsulates all the author-related information of a given software artifact. It gives an immediate view of how, in terms of development effort and distribution among authors, an artifact has been developed. We can easily figure out whether the development was done mainly by one author or many people contributed to it and to which extent.

Figure 4 shows the structural principles of a Fractal Figure, applied to a file of Mozilla (`nsTextHelper.cpp`). The figure is composed of a set of rectangles with different sizes and colors. Each rectangle, and thus each color, represents an author who worked on the file. The area of the rectangle is proportional to the percentage of commits performed by the author over the whole set of commits. In the example `warren` did $\frac{5}{14}$ of the commits followed by `dcone` and `gerv` each with $\frac{2}{14}$ of the commits. For more details on the layout algorithm and the expressive power of Fractal Figures see [7].

[6] We can only know who performed the commit, *i.e.,* if a commit includes changes done by several people, those are all mapped to one single developer.

[7] As an example: The Mozilla system, on the first of September 2005, had 4656 source code files with a total number of 326,000 file versions, corresponding to hundred of thousands of commit-related pieces of information to analyze.

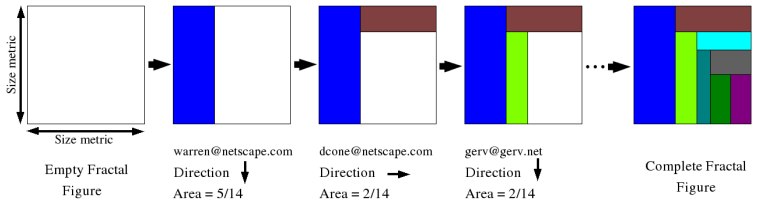
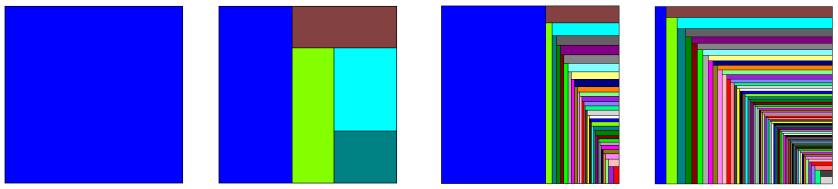


Fig. 4. The structural principles of a Fractal Figure .



(a) One developer (b) Few balanced developers (c) One major developer (d) Many balanced developers

Fig. 5. Development patterns based on the *gestalt* of Fractal Figures .

Fractal Figures allow software entities to be categorized in terms of effort distribution among developers following the *gestalt principle* . We defined four visual patterns representing four development models, depicted in Figure 5: (a) One developer, (b) few balanced developers, (c) one major developers and (d) many balanced developers.

Development patterns allow us to categorize entities according to the way they were developed, from the author perspective. However, the visual nature of both the patterns and the Fractal Figures themselves, is useful to get a qualitative impression only of the development model. To provide also a quantitative measure, we introduced the *Fractal Value* , which for a given software artifact is defined as:

$$\text{Fractal Value } = 1 - \sum_{a_i \in A} \left(\frac{nc(a_i)}{NC} \right)^2, \quad \text{with} \quad NC = \sum_{a_i \in A} nc(a_i) \qquad (1)$$

where $A = \{a_1, a_2, \ldots, a_n\}$ is the set of authors and $nc(a_i)$ is the number of commits performed by the author a_i with respect to the given software artifact. The Fractal Value measures how fragmented a Fractal Figure is, that is how much the work spent on the corresponding entity is distributed among different developers. Equation 1 is defined such that the smaller the quantity $\frac{nc(a_i)}{NC}$ is (always less than 1), the more it is reduced by the square power, since the square equation is sub-linear between 0 and 1. Therefore, the smaller a rectangle is, the less its negative contribution to the Fractal Value

is. The Fractal Value ranges from 0 to 1 (not reachable). It is 0 for entities developed by one author only, while it tends to 1 for entities developed by a large number of authors.

To exploit the expressive power of Fractal Figures we applied them in context of polymetric views [18]. Figures represent RHDB entities, namely files, directories, and modules. To apply them on a directory or a module, we sum up the commit information of all the files belonging to the given directory or module. In the polymetric view context, we also map a metric measurement of the size of the figure[8]. The metric can be structural like LOC or evolutionary like number of commits, number of bugs, number of lines added *etc.*

We use Fractal Figures to answer the questions mentioned at the beginning of this Section: How many developers worked on an entity? How was the effort distributed among them? In the following we present four different example scenarios which show how to use Fractal Figures to address the problem of understanding development effort distribution.

Scenario 1 - Detecting a major developer in Mozilla webshell

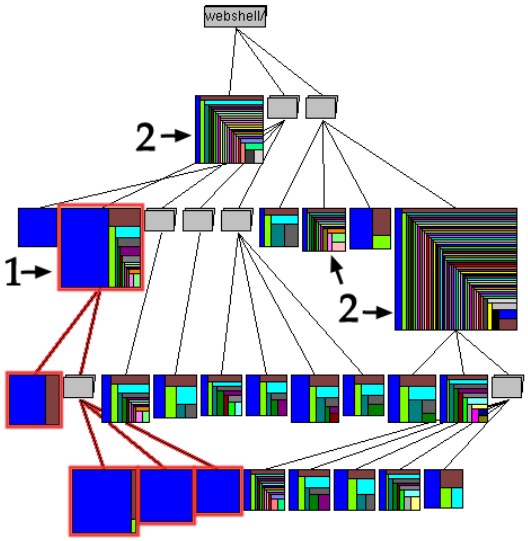


Fig. 6. The webshell hierarchy of Mozilla.

Figure 6 shows the `webshell` directory hierarchy of Mozilla. Fractal Figures represent directories containing at least one file, while grey figures repre-

[8] One metric measurement because Fractal Figures have to be squares for using their layout algorithm.

sent container directories, *i.e.,* directories containing only subdirectories. The size metric maps the directory size in terms of number of contained files. We see that the `webshell` hierarchy of Mozilla includes all the four development patterns. The sub-hierarchy marked as 1 has a major developer pattern (the blue author did most of the commits). The reverse engineer knows who to ask questions about the design and the code contained in this sub-hierarchy. On the contrary, the directory marked as 2 shows that many developers worked on it, and there is no main developer. Modifying code in these directories will be more effort since there is not the right or a single person to ask question about the code. The reverse engineer will need support of other tools like CodeCrawler [19] or BugCrawler [6] . This information is not complex or hard to get, but the value of the Fractal Figure visualization is that it conveys this information (i) in a context (the hierarchy in this case), (ii) easy and fast to read and (iii) with the same visual principle for all the software entities to which it is applied.

Scenario 2 - Re-assessing the development team formation

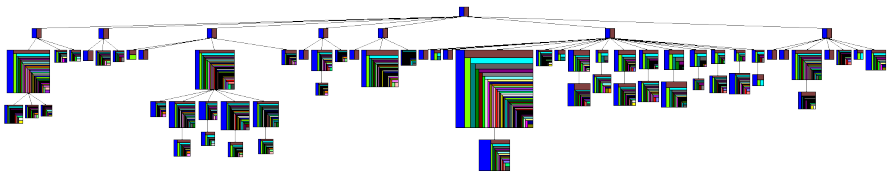


Fig. 7. The intl hierarchy of Mozilla.

Figure 7 shows Fractal Figures applied to the `intl` hierarchy of Mozilla, where the size of the figures map the number of problems (bug reports) generated from files contained in the corresponding directory. In this hierarchy the development effort distribution is clearly defined: The root directories of each sub-hierarchy are developed by two developers only, and they did not introduce bugs. In each sub-hierarchy, all the directories which introduced bugs have the same many balanced developers pattern[9]. There are no outliers, *i.e.,* figures much bigger with respect to their context.

The scenario shown in Figure 8 is different. Most of the directories which introduced bugs have a many balanced developer patterns, but one which

[9] The directories which introduced bugs can be recognized because they are bigger than the default size. To be sure that all of them have the same pattern and introduced bugs, the tool allows the user to query the visualization about the metric (*e.g.,* number of bugs in this case) and Fractal Value .

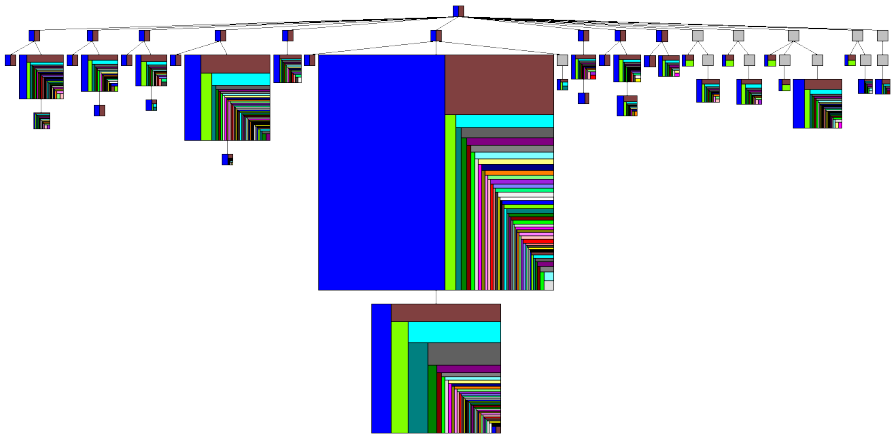


Fig. 8. The network/protocol hierarchy of Mozilla.

has a one major developer pattern: `network/protocol/http/src`. This directory is an outlier; it is responsible for most of the bugs generated in the `network/protocol` hierarchy.

The views shown in Figure 7 and Figure 8 can be valuable for a project manager or an analyst. In the first scenario the information they can get is that the development team is well-formed. In the second scenario a re-assesment of the formation of the development team is needed, given the high number of bugs and one major development pattern of the `network/protocol/http/src` directory.

Scenario 3 - Detecting a "proxy" in Gimp

Figure 9 shows a visualization of all the files belonging to the `app/actions` directory of Gimp[10], where the size of each figure is proportional to the number of bugs the corresponding file is affected by. The view is aimed at understanding how the development effort is distributed among the different authors, according to the code ownership principle.

All the files in the directory have the one developer or one major developer pattern. The author related to the blue color (*mitch*, as shown in the legend) is mainly responsible for their development, but this information should be carefully interpreted, since there is no one-to-one mapping between developers and CVS accounts: A developer can have multiple CVS accounts and a CVS account can "hide" several developers behind. It can either be that mitch developed most of the `app/actions` code or mitch is a "*proxy*" responsible for collecting patches and committing them to the repository. This is a common

[10] Gimp is an open source image manipulation program. See http://www.gimp.org

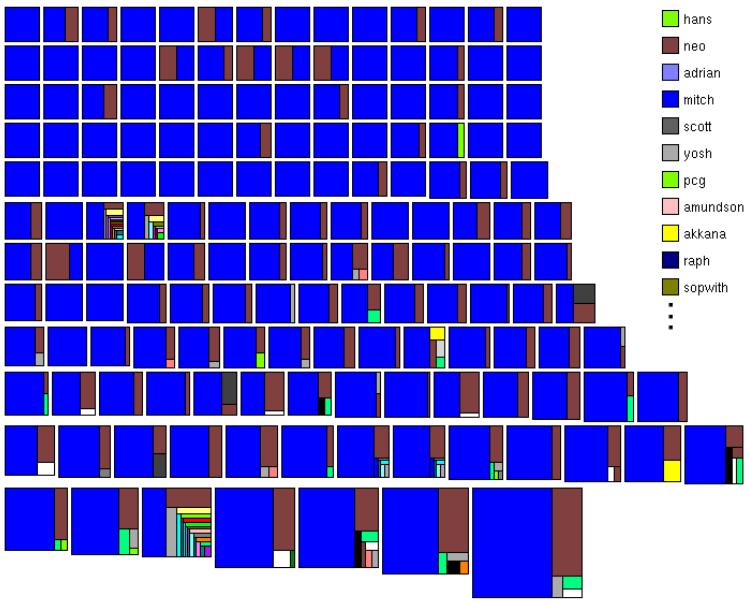


Fig. 9. The files contained in the `app/actions` directory of Gimp.

practice for open source projects, where the write permission to the repository is given to few people and changes and patches are sent to them via e-mail.

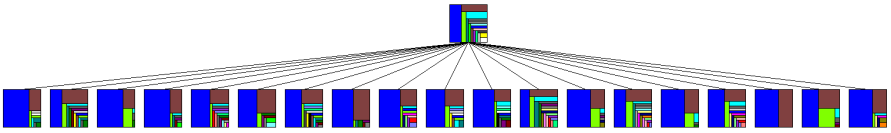


Fig. 10. The `app` hierarchy of Gimp.

To verify if mitch is a proxy, we apply the same visualization to the entire `app` hierarchy (see Figure 10), representing directories as Fractal Figures . The proxy hypothesis is confirmed, since all the directories in the `app` hierarchy have the one major developer pattern, where mitch is the main developer. To get another confirmation we applied the same view on the entire Gimp system and we saw that mitch is the main responsible for most of the directories (and thus files) in the system.

3.2 Change Coupling Analysis

In this section we introduce the EvoLens visualization technique that focuses on highlighting change couplings between source files and software modules. Software modules refer to directories in a project tree. The basic ideas and underlying concepts of the Change Coupling Views have been developed by Ratzinger *et al.* [24].

EvoLens Views

EvoLens is a graph-based visualization technique that represents the directory structure and source files as nested graphs. Source files are represented by ellipses and surrounding directories as rectangles. Change coupling dependencies between source files are visualized as straight lines between nodes. EvoLens also follows the measurement mapping principle described before. For source files the growth-rate in lines of code added or deleted is mapped to the color of nodes. Basically, a light color denotes minimum and an intensive color denotes maximum growth. Concerning the change coupling the number of common commits between two files is mapped to the width of arcs — the more common commits two source files share the thicker the arc between the two corresponding nodes. We explain the different visualization concepts with examples taken from a case-study with a picture archiving system.

When analyzing a large software system a top-down approach starting with a coarse grained picture is recommended. EvoLens follows this top-down approach by using nested graphs that on the top-level visualize modules, on the next level submodules and then the source files. A *focal point* is used to focus the analysis on the change coupling relationships of a specific entity. Entities not change coupled with the entity in focus are left out leading to simpler and better understandable graphs. The focal point is set by the user to always focus on the entity of interest.

Figure 11 depicts an example of change couplings between module and submodules with the focal point set on the module `jvision`. Nodes are drawn as rectangles and represent the modules and sub-modules. The nesting of modules and submodules is expressed by the nesting of nodes. For instance, the node in the center shows that the module `jvision` consists of 10 submodules.

EvoLens draws the intra- as well as the inter-module coupling relationships. The width of the line denotes the strength of the coupling in terms of number of common commits. On the module and submodule level this number is the sum of underlying change couplings between source files of module pairs. The graph of Figure 11 depicts the change couplings between the sub-modules of `jvision`. Within this module the submodules `image`, `main`, `renderer`, `overlay`, and `vis` are frequently changed together as indicated by thick lines drawn between corresponding graph nodes. Regarding the inter-module coupling with the `jfolder` module the submodule `main` is pointed

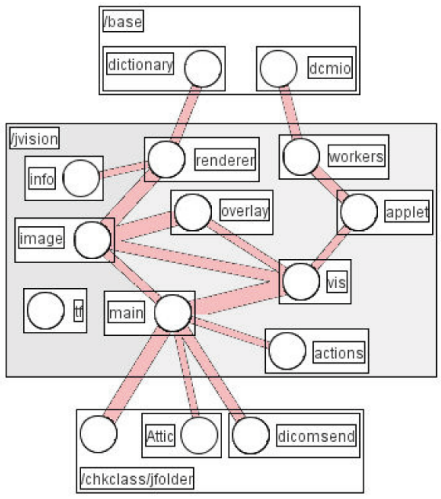


Fig. 11. Nested graph visualization of module jvision [24].

out. Consequently, this submodule is a primary candidate to re-factor for decoupling the two modules.

Zooming in and out in the graph can be done by expanding and collapsing the module and submodule nodes. EvoLens also provides a filter mechanism with configurable thresholds. It allows the user to focus on the entities with strong coupling and filter out the other entities with weak couplings. In addition to the focal point this further reduces the amount of information to be visualized in graphs.

Figure 12 describes the zoom-in into submodule `jvision/main`. The focal point is moved to the submodule `main` and its contents is unfolded and drawn in a rectangle. In this example the contained nodes are source files as indicated by the ellipses. The ellipses show different colors denoting the growth of the file. The two files `MainFrame2` and `SeqPanel2` increased most as highlighted by the dense red color of the corresponding ellipses.

The modules and submodules changed coupled with the submodule `main` are drawn as extra rectangles in the graph. In the example these modules are `jvision` with the source files `VisDisplay2` and `Vis2` and the module `chkclass` with the two source files `SendPanel` and `CHKFolderPanel`. Files with lower change coupling are filtered. Discussing this view with the developers we found out that the main class is implemented in `JVision2`. This fact partially justifies the change coupling because during start-up initializations of other parts are usually made. However, a detailed inspection of the source code yielded that `JVision2` exposes access to many parts of the system through static member variables. This should be improved.

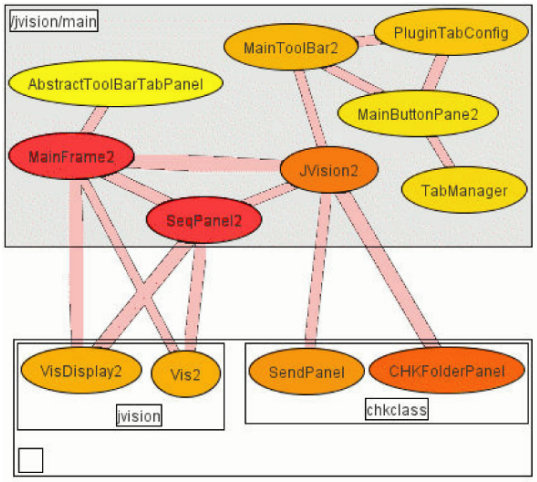


Fig. 12. Zoom of the module hierarchy of jvision [24].

Selective Change Coupling. Since module boundaries are sometimes too restrictive for in depth inspections, EvoLens incorporates the visualization of individually selected sets of files. In the graph depicted in Figure 13 we selected the four files: `MainFrame2`, `VisDisplay2`, `ImgView2`, and `Localizer`. These classes are the ones that are responsible for the strong change couplings between the submodules `main` and `vis`, and `image` and `overlay` of module `jvision`. The selected files are grouped in a rectangle and all non selected modules are folded. The change coupling within the group of files as well as with folded modules is shown. Figure 13 shows that all four selected files are change coupled with files of `jvision`. Furthermore, `MainFrame2` has weak change coupling with classes of module `chkclass`. With the help of this feature the user can select and group different sets of modules, submodules, and files and analyze their change coupling with the rest of the system.

Change couplings are measured on a time-window basis: the number of common commits of two files during a given observation period. Such an observation period, for instance can be from the very beginning of the project or the last six month. The observation period is user-configureable by setting the begin and end time. When the period is changed EvoLens recomputes the change coupling relationships and redraws the graph.

Figure 14 shows the change coupled submodules and source files of module `jvision` within the entire 18 months of the inspection period (a); and the first 9 months (b). The graph on the left-hand side shows, for instance a change coupling between the two source files `MainFrame2` and `VisDisplay2` that did not occur in the first 9 month as shown in the graph on the the right-hand side. Consequently, the coupling between the two files was introduced through later development activities.

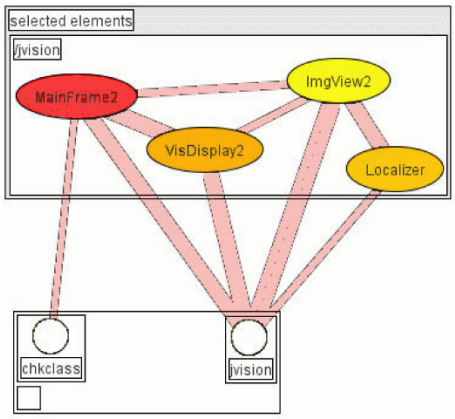
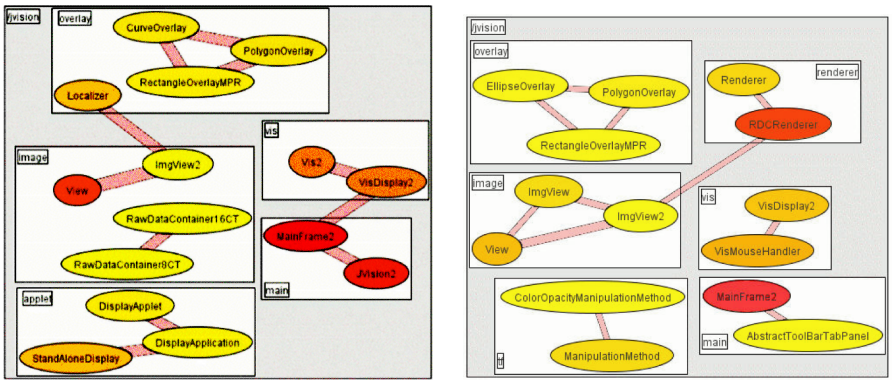


Fig. 13. Selective Coupling of files MainFrame2, VisDisplay2, ImgView2, and Localizer [24].



(a) Change coupling of jvision within entire history.

(b) Change coupling of jvision within first nine months.

Fig. 14. Sliding Time Window in EvoLens [24].

The coloring of the ellipses provides additional hints about the evolution of source files. For instance, the change coupling and coloring of `View` and `ImgView2` is striking. Both modules have a strong change coupling but the file `View` grew by more than 300 lines of code whereas `ImgView2` remained almost constant at 150 lines of code but is continuously modified. Apparently the class implemented in file `ImgView` desires a more clean and stable interface to reduce the change impact when modifying related files.

3.3 The Evolution Radar

The Evolution Radar is an interactive visualization technique for analyzing change couplings to detect architecture decay and coupled components in a given software system. We want to detect strong coupling at different levels of abstraction: Which are the components, *e.g.,* modules, with the strongest coupling? Which low level entities, *e.g.,* files, are responsible for these couplings?

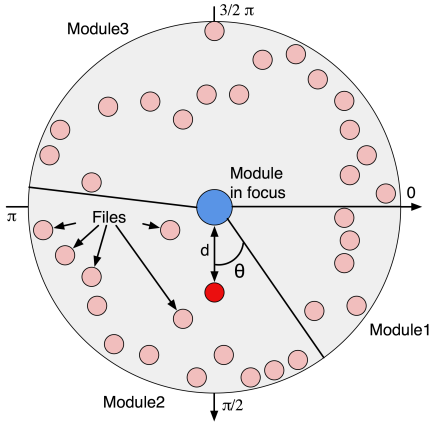


Fig. 15. The structural principles of the Evolution Radar.

Figure 15 shows the structural principles of the Evolution Radar. It visualizes dependencies between groups of entities, in this specific case dependencies between modules (groups) as group of files (entities). The module in focus is visualized as a circle and placed in the center of a pie chart. All the other system modules are represented as sectors. The size of the sectors is proportional to the number of files contained in the corresponding module. The sectors are sorted according to this size metric, *i.e.,* the smallest is placed at 0 radian and then all the others clockwise (see Figure 15). Within each sector files are represented as colored circles and positioned using polar coordinates where the angle and the radius are computed according to the following rules:

- *Radius d* (or distance from the center). It is inversely proportional to the change coupling the file has with the module in focus, *i.e.,* the more they are coupled, the closer the circle (representing the file) is to the center circle (representing the module in focus).
- *Angle θ*. The files of each module are alphabetically sorted considering the entire directory path, and the circles representing them are then uniformly distributed in the sectors with respect to the angle coordinates.

Moreover, arbitrary metrics can be mapped on the color and the size of the circle figures.

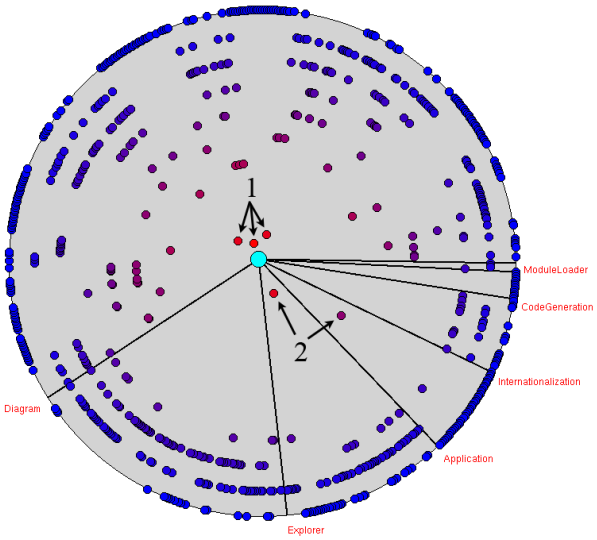


Fig. 16. An example Evolution Radar applied on the *Model* module of ArgoUML .

Figure 16 shows an example Evolution Radar visualizing the coupling between the *Model* module (represented as the cyan circle in the center) and all the other modules of ArgoUML[11] (represented as the sectors). The size of the figures is fixed and the color metric is the same as the distance, *i.e.,* the change coupling. We see that the *Diagram* module is the largest and most coupled module. The three files marked as 1 in the figure are the ones with the strongest coupling. They should be further analyzed to understand which is the most appropriate module to contain them: *Model* or *Diagram*. For the remaining modules the coupling is not as strong as for *Diagram* but we see the presence of some outliers (files for which the coupling is much higher with respect to their context). The two files marked as 2, belonging to the *Application* and *Internationalization* modules, have a very strong coupling with respect to the other files belonging to the same modules. They should also be analyzed and moved in case they belong to the wrong module.

In the Evolution Radar files are placed according to the change coupling they have with the module in focus. To compute this metric value we use the following formula:

$$CC(M, f) = \max_{f_i \in M} CC(f_i, f) \tag{2}$$

[11] ArgoUML is a UML modeling tool written in Java (see argouml.tigris.org).

$CC(M, f)$ is the change coupling between the module in focus M and a given file f and $CC(f_i, f)$ is the coupling between the files f_i and f. It is also possible to use other group operators instead of the maximum like the average or the median. We use the maximum because it points us to the files with the strongest coupling, *i.e.,* the main responsible for the module dependencies.

The value of the coupling between two files is equal to the number of transactions which include both files. Since transactions are not recorded by CVS we reconstruct them using the sliding time window approach proposed by Zimmermann and Weißgerber in [30], which is an improvement of the simpler fixed time window approach. For further details about the sliding and the fixed time window approach we refer the readers to [5, 30].

The Evolution Radar is implemented as an interactive visualization. It is possible to inspect all the entities visualized, *i.e.,* files and modules, to see commit-related information like author, timestamp lines added and removed *etc.* Moreover, it is also possible to see the source code of selected files. Three important features for performing analyses with the Evolution Radar are (1) moving through time, (2) tracking and (3) spawning.

(1) Moving through Time. The change coupling measure is time dependent. If we compute it considering the whole history of the system we can obtain misleading results. Figure 17 shows an example of such a situation.

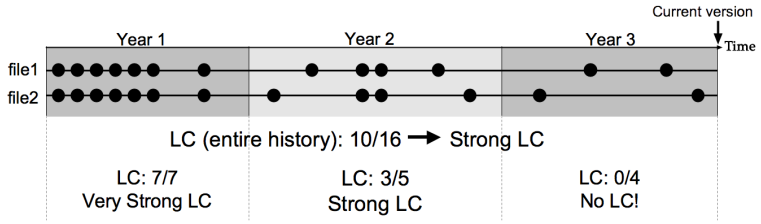


Fig. 17. An example of misleading results obtained by considering the entire history of artifacts to compute the change coupling value: We obtain a strong change coupling, while file1 and file2 are not coupled at all during the last year.

Figure 17 shows the history, in terms of commit, of two files, where the time is on the horizontal axis from left to right and commits are represented as circles. If we compute the change coupling measure according to the entire history we obtain 10 shared commits on a total of 16, which is an high value because it means that the files changed together more than fifty percent of the time. Although this result is correct, it is misleading because it brings us to the conclusion that file1 and file2 are strongly coupled, but they were so only in the past and they are not coupled at all during the last year of the system. Since we analyze change coupling information for detecting architecture decay and design issues in the current version of the system, recent change couplings

are more important than old ones. In other words, if two files were strongly coupled at the beginning of a system, but they are not coupled in recent times (perhaps because the coupling was removed during a reengineering phase), we do not consider them as a potential problem.

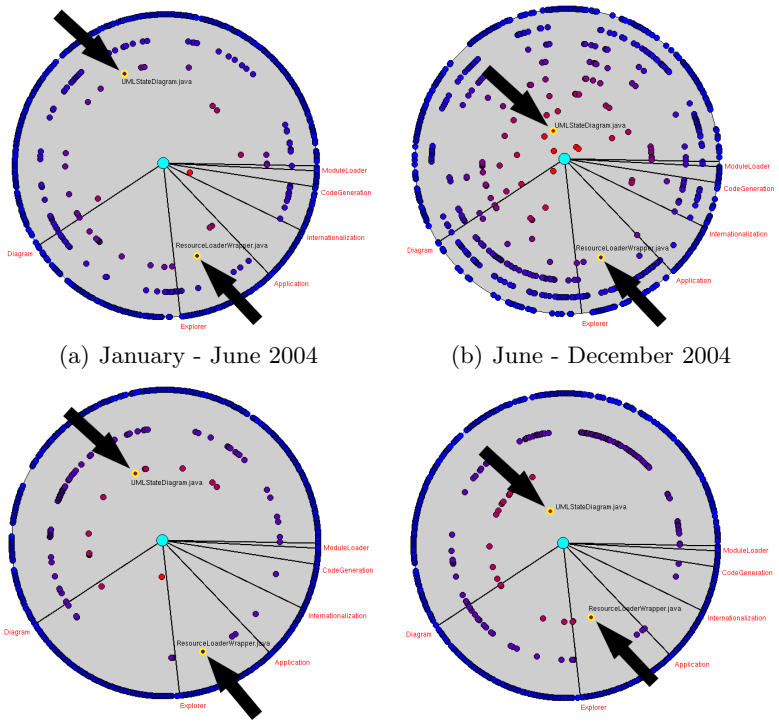
For these reasons the Evolution Radar is time dependent, *i.e.,* it can be computed either considering the entire history of files or with respect to a given time window. When creating the radar the user can divide the lifetime of the system into time intervals. For each interval a different radar is created, and the change coupling is computed with respect to the given time interval. The radius coordinate has the same scale in all the radars, *i.e.,* the same distance in different radars represents the same value of the coupling. This makes it possible to compare radars and to analyze the evolution of the coupling over time. In our tool implementation the user "moves through time" by using a slider, which causes the corresponding radar to be displayed. This feature introduced also a problem: How do we keep track of the same entity over time, *i.e.,* on different radars? To answer this question we introduced a second feature called tracking.

(2) Tracking. It allows the user to keep track of files over time. When a file is selected for tracking in a visualization related to a particular time interval, it is highlighted in all the radars (with respect to all the other time intervals) in which the file exists.

Figure 18 shows an example of tracking through four radars, related to four consecutive time intervals, from January 2004 to December 2005. The highlighting consists in using a yellow border for the tracked files and in showing a text label with the name of the file (indicated with arrows in Figure 18). Like this it is possible to detect files with a strong change coupling with respect to the last period of time and then move the time and analyze the coupling in the past. This allows the distinction between persistent change coupling, *i.e.,* always present, and recent change coupling, *i.e.,* present during the last time intervals only.

(3) Spawning. The spawn feature is aimed at inspecting the change coupling details. Outliers indicate that the corresponding files have a strong coupling with certain files of the module in focus, but we ignore which ones. To uncover this dependency between files we spawn a secondary Evolution Radar as follows: The outliers are grouped to form a temporary module M_t represented by a circle figure. The module in focus (M) is then expanded, *i.e.,* a circle figure is created for each file composing it. Finally, a new Evolution Radar is created. The temporary module M_t is placed in the center of the new radar. The files belonging to the module previously in focus (M) are placed around the center. The radius coordinate, *i.e.,* the distance from the center, is inversely proportional to the change coupling they have with the module in the center M_t. For the angle coordinate alphabetical sorting is used. Since all the files belong to the same module there is only one sector.

We use The Evolution Radar to answer the questions mentioned at the beginning of this Section: Which are the modules with the strongest coupling



(a) January - June 2004 (b) June - December 2004

(c) January - June 2005 (d) June - December 2005

Fig. 18. The change coupling evolution of the *Model* module of ArgoUML. Moving through time, the Evolution Radar can keep track of certain files (yellow border).

in a given software system? Which files are responsible for these evolutionary dependencies? In the following we apply the radar on ArgoUML[12], a large and long-lived open source software system. We first present example scenarios of how to study change coupling at different levels of abstraction, detecting architecture decay and design problems and performing impact analysis. We finally use the radar to analyze the evolution of the couplings, identifying phases in the history of the system.

Detecting design issues and architecture decay

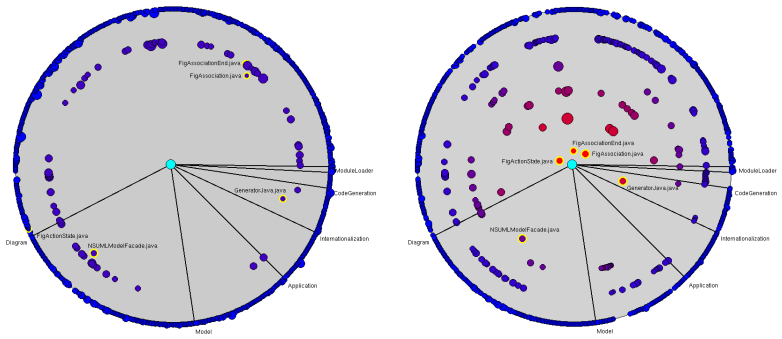
From the documentation of ArgoUML we know the system decomposition in modules[13]. We focused our analysis on the three largest modules: *Model*,

[12] ArgoUML is an open-source UML modeling tool written in Java, consisting of more than 200,000 lines of code. It is available at http://argouml.tigris.org.

[13] We did not consider some modules for which the documentation says "They are all insignificant enough not to be mentioned when listing dependencies".

Explorer and *Diagram*. From the documentation we know that *Model* is the central module that all the others rely and depend on. *Explorer* and *Diagram* do not depend on each other.

We created a radar for every six months of the system's history. We started the study from the most recent one, since we are interested in problems in the current version of the system. Using a relatively short time interval (six months) ensures that the coupling is due to recent changes and is not "polluted" by commits far in the past. As metrics we used the change coupling for both the position and the color of the figures. The size (the area) is proportional to the total number of lines modified in all the commits performed during the considered time interval.



(a) From January to June 2005. (b) From June to December 2005.

Fig. 19. Evolution Radars applied to the *Explorer* module of ArgoUML for the year 2005.

Figure 19(b) shows the Evolution Radar for the last six months of history of the *Explorer* module. From the visualization we see that the coupling with *Diagram* is much stronger than the one with *Model*, although the documentation states that the dependency is with *Model* and not with *Diagram*. The most coupled files in *Diagram* are *FigActionState.java*, *FigAssociationEnd.java*, *FigAssociation.java*. Using the tracking feature, we found out that these files have only been recently coupled with the *Explorer* module. In the other radar (Figure 19(a), showing the previous six months) they are not close to the center. This implies that the dependency is due to recent changes only.

To inspect the change coupling details, we used the spawning feature: We grouped the three files and we generated another radar, shown in Figure 20 having this group as the center. We now see that the dependency is mainly due to *ExplorerTree.java*. The high-level dependency between two modules is thus reduced to a dependency between four files. These four files represent a problem in the system, because modifying one of them may break the others.

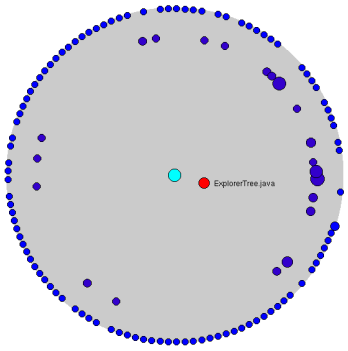


Fig. 20. Details of the change coupling between the *Explorer* module and the files *FigActionState.java*, *FigAssociationEnd.java* and *FigAssociation.java*.

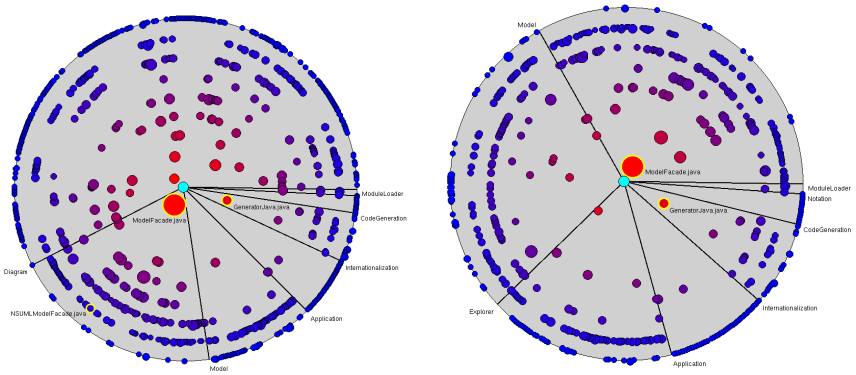
The fact that they belong to different modules makes it easier to forget this hidden dependency.

The visualization in Figure 19(b) shows that the file *GeneratorJava.java* is an outlier, since its coupling is much stronger with respect to all the other files in the same module (*CodeGeneration*). By spawning the group composed of *GeneratorJava.java* we obtained a visualization very similar to Figure 20, in which the main responsible for the dependency is again *ExplorerTree.java*. Reading the code revealed that the *ExplorerTree* class is responsible for managing mouse listeners and generating names for figures. This explains the dependencies with *FigActionState*, *FigAssociationEnd* and *FigAssociation* in the *Diagram* module, but not the dependency with *GeneratorJava*.

The past (see Figure 19(a) and Figure 21(a)) reveals that *GeneratorJava.java* is an outlier since January 2003. This long-lasting dependency indicates design problems.

A further inspection is required for the *ExplorerTree.java* file in the *Explorer* module, since it is the main responsible for the coupling with the modules *Diagram* and *CodeGeneration*.

The radars in Figure 19(b) and Figure 19(a) show that during 2005 the file *NSUMLModelFacade.java* in the *Model* module had the strongest coupling with *Explorer* (module in the center). Going six months back in time, from June to December 2004 (see Figure 21(a)), we see that the coupling with *NSUMLModelFacade.java* was weak, while there was a very strong dependency with *ModelFacade.java*. This file was also heavily modified during that time interval, given its dimension with respect to the other figures (the area is proportional to the total number of lines modified). *ModelFacade.java* was also strongly coupled with the *Diagram* module (see Figure 21(b)). By looking at its source code we found out that this was a God class[25] with thousands of lines of codes, 444 public and 9 private methods, all static. The *ModelFacade* class is not present in the other radars (Figure 19(b) and Figure 19(a))



(a) Explorer module. (b) Diagram module.

Fig. 21. Evolution Radars of the *Explorer* and *Diagram* modules of ArgoUML from June to December 2004.

because it was removed from the system the 30th of January 2005. By reading the source code of the most coupled file in these two radars, *i.e., NSUMLModelFacade.java*, we discovered that it is also a very large class with 317 public methods. Moreover, we found out that 292 of these methods have the same signature of methods in the *ModelFacade* class[14], with more that 75% of the code duplicated. *ModelFacade* represented a problem in the system and thus was removed. Since many methods were copied to *NSUMLModelFacade*, the problem has just been relocated!

This example shows how historical information can reveal problems, which are difficult to detect with only one version of the system. Knowing the evolution of *ModelFacade* helped us in understanding the role of *NSUMLModelFacade* in the current version of the system.

We showed examples of how to use the Evolution Radar to detect problematic parts of the ArgoUML system, which represent good candidates for reengineering. The main findings of the discussed example scenario are:

- The *Diagram* and *Explorer* modules are the most coupled. Since this dependency is not mentioned in the module relationships page in the documentation, either the modules should be restructured to decrease the coupling or the documentation should be updated. We identified the four files mainly responsible for this hidden dependency.
- The files *GeneratorJava.java* in the *CodeGeneration* module and *ExplorerTree.java* in the *Explorer* module should be further analyzed and, in case, refactored. *GeneratorJava.jav*a has a persistent coupling with the

[14] With the difference that in *NSUMLModelFacade* the methods are not static and that it contains only two attributes, while *ModelFacade* has 114 attributes.

Explorer module, while *ExplorerTree.java* is coupled with both *CodeGeneration* and *Diagram*.
- Two problematic classes were detected: *ModelFacade* and *NSUMLModelFacade*. Most of the methods of the first class were copied to the second one, and then *ModelFacade* was removed from the system.

Identifying phases in the history

In this last scenario we study the evolution of change coupling at the module level to (i) detect phases in the history of the systems and (ii) to understand whether the change coupling relationship of a module is "ameliorating" or "degrading". It is degrading if the module is more and more logically coupled to the others, leading to maintenance problems and suggesting refactoring.

We analyze the evolution of the coupling between *Explorer* and all the other modules of ArgoUML. From Figure 21(a), we see that from June to December 2004 the couplings were very strong. Then, from January 2005 to June 2005 (Figure 19(a)), they decreased a lot. This suggests that in the previous period the module was restructured and its quality was improved, since in the next time interval the couplings with the other modules were weak. The effort spent for the restructuring can be seen from the size of the figures, representing the total number of changed lines. In the radar relative to June - December 2004 (Figure 21(a)) the figures are bigger than in the radar relative to January - June 2005 (Figure 19(a)). At the end of the restructuring phase, the class *ModelFacade* was removed. From June to December 2005 (see Figure 19(b)) the coupling increased again. This can be related to a new restructuring phase.

3.4 Trend analysis and hot-spot detection

In this section we present the ArchView approach used to create different higher-level views on the source code. Views visualize the software modules and their dependency relationships. Software modules stem from the decomposition of a system into manageable implementation units. Such units, for instance are packages, source code directories, classes, or source files. Dependency relationships refer to *uses* or *inheritance* dependencies. The uses-dependency is further detailed into dependency relationships on the source code level namely file includes, method calls, variable accesses, and type dependencies.

The objective of the Multiple Evolution Metrics Views is to point out implementation specific aspects of *one* and *multiple* source code releases. For instance, highlighting modules that are exceptionally large, complex, and exhibit strong dependency relationships to other modules. They are the so called *hot-spots* in the system. Furthermore, modules with a strong increase in size and complexity, or modules that have become unstable are highlighted. Such views can be used by software engineers, for instance to 1) get a clue of the

implemented design and its evolution; 2) to spot the important modules implementing the key-functionality of a software system; 3) to spot the heavily coupled modules; 4) to identify critical evolution trends. The basic ideas and underlying concepts of Multiple Evolution Metrics Views have been developed in the work of Pinzger et al. [23].

Source Code Data. For the Multiple Evolution Metrics Views the input data comprises structural source code information and metrics data extracted from a number of source code releases. Source code metrics quantify the size, program complexity, and coupling of modules and the strength of dependency relationships. Typical module size metrics are lines of code (LOC), number of methods, number of attributes, etc. Program complexity metrics, for instance, are McCabe Cyclomatic Complexity [21], Halstead Intelligent Content, Halstead Mental Effort, and Halstead Program Difficulty [13]. The strength of dependency relationships, for instance, is given in number of static method calls or attribute accesses between two modules.

The extraction and computation of the dependency relationships and the metric values is done using parsing and metrics tools. In our case-studies with the Mozilla open source web-browser we used the tool Imagix-4D [15] for C/C++ parsing and metric computation for a selected set of source code releases. The selection of the source code releases depends on the time period for the analysis. For instance, all major releases may be selected when analyzing the whole life cycle of a software system.

Obtained metric values of each module and dependency relationship are assigned to a feature vector. Feature vectors are tracked over the selected n releases and composed to the evolution matrix E. The values in the matrix quantify the evolution of a module or dependency relationship:

$$E_{i \times n} = \begin{pmatrix} m'_1 & m''_1 & .. & m^n_1 \\ m'_2 & m''_2 & .. & m^n_2 \\ . & . & ... & . \\ . & . & ... & . \\ m'_i & m''_i & .. & m^n_i \end{pmatrix}$$

It contains n feature vectors with measures of i metrics. Evolution matrices are computed for each module and dependency relationship. They form the basic input to our ArchView visualization approach.

Visualizing Multiple Metric Values of One Release

The ArchView approach is an extension of the Polymetric Views technique presented by Lanza et al. [18]. Instead of using graphical shapes limited in the number of representable metrics ArchView uses *Kiviat* diagrams also known as *Radar* diagrams. These diagrams are suited to present multiple metric values available for a module as described next.

[15] http://www.imagix.com

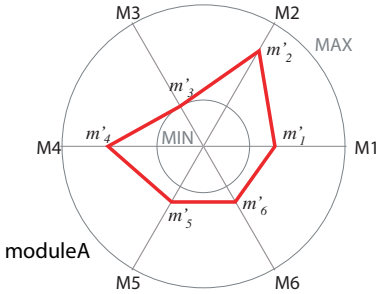


Fig. 22. Kiviat diagram of moduleA representing measures of six source code metrics $M_1, M_2, ..., M_6$ of one release.

Figure 22 shows an example of a Kiviat diagram representing measures of six metrics $M_1, M_2, ..., M_6$ of one release of the module `moduleA`. The underlying data is from the following evolution matrix E:

$$E_{6\times 1} = \begin{pmatrix} m'_1 \\ . \\ . \\ m'_6 \end{pmatrix}$$

In a Kiviat diagram the metric values are arranged in a circle. For each metric there is a straight line originating in the center of the diagram. The length of this line is fixed for all metrics and each metric value is normalized according to it. In the examples presented in this section we use the following normalization:

$$l(m_i^{'}) = \frac{m_i^{'} * cl}{max(m_i^{'})} \tag{3}$$

where cl denotes the constant length of the straight line, and $max(m_i^{'})$ the maximum value for a metric $m_i^{'}$ across all modules to be visualized. With the normalized value and the angle of the straight line denoting the metric the drawing position of the point on the line is computed. To make the metric values visible in the diagram adjacent metric values are connected forming a polygon such as shown in Figure 22.

Kiviat diagrams are the nodes in a graph representing modules, connected by edges denoting, for instance uses-dependency relationships between modules. The Polymetric Views principle is also applied to edges by mapping the number of underlying dependency relationships such as the number of static method calls between two modules to the width of an arc.

The set of metrics and their arrangement in the diagram can be configured. The same holds for the types of dependency relationships and the metric that is mapped to the width of arcs. This allows the user to create *different* views on the implementation highlighting particular aspects. For instance, Figure 23

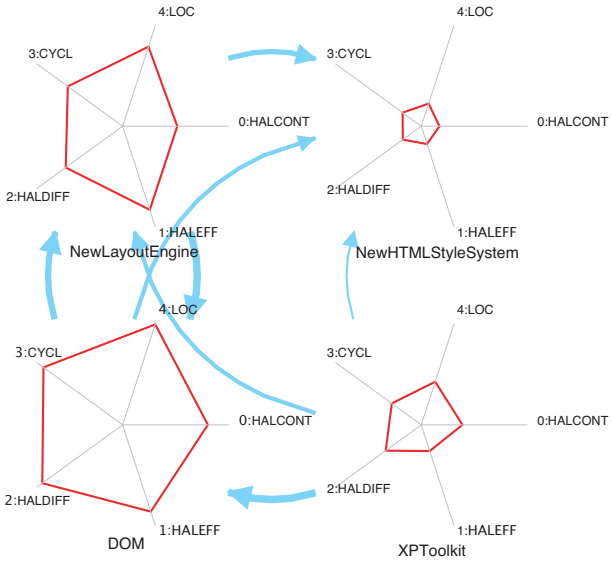


Fig. 23. Kiviat graph of four Mozilla content and layout modules showing program complexity, lines of code, and strong include dependencies of release 1.7.

shows a view on four content and layout modules of the Mozilla source base of release 1.7.

The aspects visualized in this graph concern the highlighting of the large and complex modules as well as the strong include dependency relationships between them. Kiviat diagrams represent the four modules and blue arcs between them the include dependency relationships. In the Kiviat diagrams the size of a module is represented with lines of code (LOC), program complexity with Halstead (HALCONT, HALEFF, HALDIFF) and McCabe Cyclomatic Complexity (CCMPLX) metrics. The width of arcs represents the strength of the include dependency relationships whereas the number of include relationships crossing module boundaries are counted. Large and complex modules are pointed out by large, red polygons. Strong include dependencies are represented by thick blue arcs.

Using this mapping the view clearly shows that `NewLayoutEngine` and `DOM` (Document Object Model) are large and complex. Compared to them, `NewHTMLStyleSystem` is a rather small module. The module view also shows the strong include dependencies between the four modules. Interesting is that `DOM`, which implements content functionality includes a high number of files from the `NewLayoutEngine` and `NewHTMLStyleSystem` module who provide functionality for the layout of web-pages but not vice versa. Furthermore, there is a strong bidirectional include dependency between the `NewLayoutEngine` and the `DOM` modules. Both potential shortcomings should be discussed with

the developers because they hinder separate maintenance and evolution of these three modules.

Visualizing Multiple Metric Values of Multiple Releases

When visualizing the metric values for a number of subsequent releases our main focus is on highlighting the change between metric values. Typically, increases in metric values indicate the addition and decreases the removal of functionality. The addition of functionality is a usual sign of evolving software systems so represents no problem. In contrast, the removal of functionality often indicates changes in the design. For instance, methods are moved to a different class to resolve a bidirectional dependency relationship and improve separation of concerns or methods are deleted because of removal of dead code (*i.e.*, code that is not used anymore).

To highlight the changes in metric values we use the Kiviat diagrams as described before. The n values of each metric obtained from the multiple releases are drawn on the same line. Again the adjacent metric values of the same release are connected by a line forming a polygon for each release. Then the emerging area between two polygons of two subsequent releases are filled with different colors. Each color indicates and highlights the change between the metric values of two releases. The larger the change the larger the polygon.

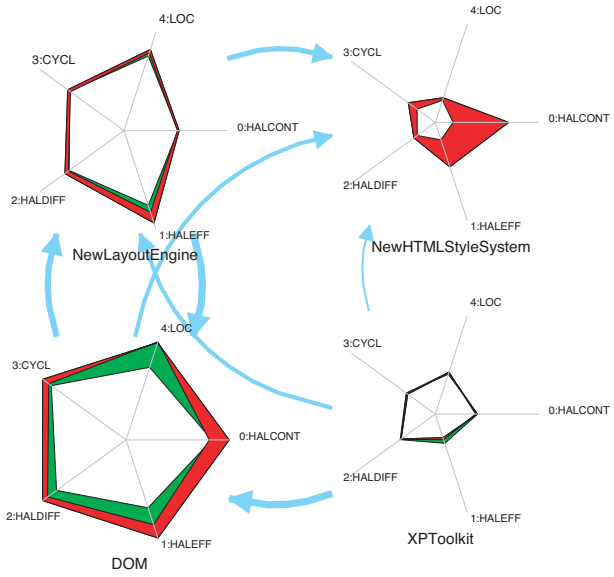


Fig. 24. Kiviat graph of four Mozilla content and layout modules showing program complexity, lines of code metrics, and strong include dependencies of three subsequent releases 0.92, 1.3a, and 1.7.

To distinguish the changes between different source code releases we use the color gradient of the rainbow. Figure 24 depicts the same four Mozilla modules as before but this time with metrics data of three subsequent releases 0.92, 1.3a, and 1.7. The view shows strong changes in the two modules `DOM` and `NewHTMLStyleSystem`. In the latter module the two Halstead metric values HALCONT and HALDIFF decreased between the previous and the last release remarkably though the size (LOC) did not change a lot. Apparently refactorings on the code were performed to reduce the difficult content making the module's source code easier to understand. The metric values of the `DOM` module first increased and then in the last release decreased again. First, functionality was added to the module which during the implementation of the last release was refactored. In comparison to these two modules the metric values of the other modules indicate only minor changes in size and program complexity hence they are stable. Based on the assumption that modules that changed in past release will be likely to change in future releases the two modules `DOM` and `NewHTMLStyleSystem` are the candidates that should be taken care of.

4 Related Work

A number of approaches have been developed that concentrate on visualizing the revision and change history of software systems. Riva *et al.* analyzed the stability of the architecture [11, 14] by using colors to depict the changes over a period of releases. Similar to Riva, Wu *et al.* describe an Evolution Spectrograph [28] that visualizes a historical sequence of software releases. Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [26]. Voinea *et al.* [27] presented the CVSscan approach, which allows the user to interactively investigate the version and change information from CVS repositories with a line-oriented display. The visualization techniques presented in this chapter complements these approaches by taking into account change coupling data, bug data, and multiple metrics of multiple releases.

Our techniques work at the level of source files and directories. Other approaches work at finer granularity levels such as methods. For instance, Zimmermann *et al.* [31] used the information about changes that are occurring together to predict entities (classes, methods, fields *etc.*) that are likely to be modified when one is being modified. In [2] Breu and Zimmermann applied data mining techniques on co-changed entities to identify and rank cross-cutting concerns in software systems. Ying *et al.* applied data mining techniques to the change history of the code base to identify change patterns to recommend potentially relevant source code for a particular modification task [29]. Cubranic and Murphy introduced the Hipikat [3] approach. Hipikat uses project information to provide recommendations for a modification task. Project information comprises a number of different sources, including the

source code versions, modification task reports, news-group messages, email messages, and documentation. The focus of these approaches is on providing recommendations for relevant project artifacts to developers who are evolving a system whereas our focus is on software evolution analysis.

A number of visualization techniques and tools have been developed in the area of reverse engineering and program understanding. Tools such as the Bookshelf of Finnigan *et al.* [9], Dali of Kazman *et al.* [15], Bauhaus[16] of Koschke *et al.*, Rigi of Müller *et al.* [22], and Creole[17] of Storey *et al.* use graph-like visualization techniques to create views of *one* particular source code release. Nodes typically represent source code entities and edges represent the dependency relationships between them. Similar, commercial reverse engineering and program understanding tools such as Imagix4D, Sotograph, and Cast exist. Our visualization techniques are based on these techniques but also include data about the evolution and multiple source code releases. Another extension to these approaches is the use of Polymetric Views.

Polymetric Views as used by our visualization techniques have been introduced by Lanza and Ducasse [18] for visualizing source code with graphs enriched with metric values. They integrated a number of pre-defined views In the CodeCrawler tool [17] facilitating coarse and fine-grained software visualization. Views follow the principle of measurement mapping in that larger metric values lead to larger glyphs in a graph as described in [8]. Using the concepts of Polymetric Views and taking into account different revisions of classes Lanza *et al.* then presented the Evolution Matrix [16]. Similarly, Girba *et al.* described an approach that based on summarizing source code metric values of several releases facilitates identifies change prone classes [12]. Source Viewer 3D (sv3D) is a tool that uses a 3D metaphor to represent software systems and analysis data [20]. The representation is based on the SeeSoft pixel metaphor [1] and extends it to 3D.

References

1. T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
2. S. Breu and T. Zimmermann. Mining aspects from version history. In *Proceedings of ASE 2006 (21st IEEE International Conference on Automated Software Engineering)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.
3. D. Cubranić and G. C. Murph. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Portland, Oregon, 2003. IEEE Computer Society Press.
4. M. D'Ambros. Software archaeology - reconstructing the evolution of software systems. Master thesis, Politecnico di Milano, Apr. 2005.

[16] http://www.iste.uni-stuttgart.de/ps/bauhaus
[17] http://www.thechiselgroup.org/creole

5. M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189 – 198, 2006.
6. M. D'Ambros and M. Lanza. Bugcrawler: Visualizing evolving software systems. In *Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering)*, page to be published, 2007.
7. M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of Vissoft 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*, pages 46–51, 2005.
8. N. E. Fenton and S. L. Pfleeger, editors. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 2nd edition, 1996.
9. P. Finnigan, R. C. Holt, I. Kallas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.
10. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Los Alamitos CA, Sept. 2003. IEEE Computer Society Press.
11. H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, UK, 1999. IEEE Computer Society Press.
12. T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the International Conference on Software Maintenance*, pages 40–49, Chicago, Illinois, 2004. IEEE Computer Society Press.
13. M. H. Halstead. Elements of software science, operating, and programming systems series. *Elsevier*, 7, 1977.
14. M. Jazayeri. On architectural stability and evolution. In *Proceedings of the Reliable Software Technlogies-Ada-Europe*, pages 13–23, Vienna, Austria, 2002. Springer Verlag.
15. R. Kazman and S. J. Carriére. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.
16. M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 37–42, Vienna, Austria, September 2001. ACM Press.
17. M. Lanza. Codecrawler - polymetric views in action. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 394–395, Linz, Austria, 2004. IEEE Computer Society Press.
18. M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.
19. M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.
20. J. I. Maletic, A. Marcus, and L. Feng. Source viewer 3d (sv3d): a framework for software visualization. In *Proceedings of the 25th International Conference on Software Engineering*, pages 812–813, Portland, Oregon, 2003. IEEE Computer Society Press.
21. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

22. H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, April 1988. IEEE Computer Society Press.
23. M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75, St. Louis, Missouri, 2005. ACM Press.
24. J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005. IEEE Computer Society Press.
25. A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.
26. F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 328–337, Chicago, Illinois, USA, September 2004. IEEE Computer Society Press.
27. L. Voinea, A. Telea, and J. 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.
28. J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 57–66, Kyoto, Japan, September 2004. IEEE Computer Society Press.
29. A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.
30. T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press.
31. T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.