3

Analysing 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] Department of Informatics, University of Zurich, Switzerland

Summary. Software repositories such as versioning systems, defect tracking systems, and archived communication between project personnel are used to help manage the progress of software projects. Software practitioners and researchers increasingly recognize the potential benefit of mining this information to support the maintenance of software systems, improve software design or reuse, and empirically validate novel ideas and techniques. Research is now proceeding to uncover ways in which mining these repositories can help to understand software development, to support predictions about software development, and to plan various evolutionary aspects of software projects.

This chapter presents several analysis and visualization techniques to understand software evolution by exploiting the rich sources of artifacts that are available. Based on the data models that need to be developed to cover sources such as modification and bug reports we describe how to use a Release History Database for evolution analysis. For that we present approaches to analyse developer effort for particular software entities. Further we present change coupling analyses that can reveal hidden change dependencies among software entities. Finally, we show how to investigate architectural shortcomings over many releases and to identify trends in the evolution. Kiviat graphs can be effectively used to visualize such analysis results.

3.1 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 report 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 valuable information for analysing the evolution of software. While the recovery of the data residing in versioning systems such as CVS or Subversion has become a well explored topic, the ultimate challenge lies in the recovered data and its interpretation.

Some recent topics addressed in the field of analysing software repositories include the following:

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

- *Developer effort and social network analysis.* One of the goals in this topic is to find out the effort that team members are spending on maintaining and evolving software modules and how they communicate with each other. This allows a project manager to plan resources and reason about shortcomings in development processes and the team structure.
- *Change impact and propagation.* The main focus of this topic is to assess the impact of a change, such as the addition of a new or change of an existing feature, on the architecture, design and implementation of a software system. Being able to assess the impact of changes allows one to estimate the effort for maintenance and evolution tasks, to determine the impact of a change on the existing architecture and design of a system. Results are also used to provide guidelines for programmers such as if changing method `a` the programmer should also change method `b` and `c`.
- *Trend and hotspot analysis.* In this topic the trend of software entities is observed to find out shortcomings in the current architecture, design and implementation of software systems. Hotspots are the entities that frequently change and therefore are critical for the evolution of a system. One of the goals is to find heuristics and warning mechanisms that alarm project managers and architects of negative trends of software entities (and in particular of system hotspots) and provide suggestions to return the system into a stable state.
- *Fault and defect prediction.* A wealth of information is provided by software repositories that can be input to data mining and machine learning algorithms to characterize current and predict future properties of software entities. One property of software entities that is addressed by many approaches is the prediction of the location and number of defects in software entities such as source files. The result is a list of entities that will likely to be affected by defects which allows the development team to plan preventive actions such as refactoring.

In this chapter we address the first three topics and present techniques such as our Fractal Figures to analyse development effort, the Evolution Radar to analyse the change impact on source files, and Kiviat diagrams to analyse metric trends and to detect system hotspots. In the next section we present a general approach for analysing software repositories to understand software evolution. Examples of how to model and retrieve the data is presented in Section 3.3. The modeled data is the input for the different software evolution analysis techniques we present in Section 3.4.

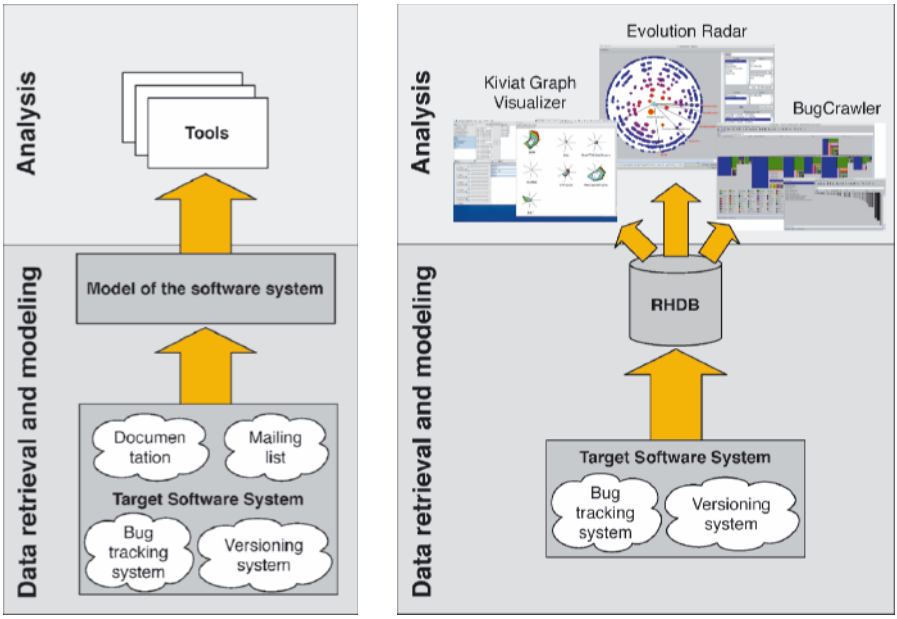
3.2 An Overview of Software Repository Analysis

When mining software repositories one can consider many software artifacts: Source code from versioning systems, bugs from bug tracking systems, communication from e-mail lists or any further software artifacts such as documentation. This diversity of information is the foundation for many kinds of evolution analyses.

3.2.1 A General Approach

Figure 3.1a shows a sketch of how to analyse software repositories for studying software evolution. 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 of creating a data model of an evolving software system. 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, i.e., not all aspects of a system's evolution have to be considered, but only the ones which can address 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. The processing can include the parsing of the data (e.g., source code, log files, bug report etc.), the application of matching



(a) General schema **(b)** Our approach

Fig. 3.1. The general approach and our customization to mining software repositories

techniques to link different data sources (e.g., versioning system artifacts with bug reports [179, 136]), the reconstruction of information not recorded (e.g., reconstruct commit information from CVS log files [566]) and the application of other techniques such as data mining.
3. *Data analysis*. The analysis consists of using the modeled and retrieved data to tackle a software evolution problem or set of problems by means of different techniques and approaches.

3.2.2 Our Approach

Figure 3.1b sketches 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 data model describing an evolving software system based on these two data sources (data modeling). Given a system to analyse, 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 (RHDB), which is the starting point for all the subsequent analyses. For the analysis part we use different techniques and tools, aimed at addressing specific software evolution problems.

In the remainder of this chapter we first introduce the RHDB, the data model behind it and the way the database is populated. Then we present different types of software evolution analyses built on top of the RHDB: Developers effort distribution, change coupling, trend analysis and hot-spot detection. For the RHDB and each evolution analysis technique we also present related work in the field.

3.3 Release History

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 analyse evolution our goal is to understand a system's architecture, the dependencies among its components and to detect evolutionary hot-spots. To support this kind of analysis, additional information such as problem bug reports can be used. The problem is to link this data to the software artifacts to answer specific questions, e.g., which files were affected by a given bug?

In this section we present our approach for integrating versioning system information and bug report data and populating a RHDB [179, 136]. 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 [135] has been the most used version control system by the open source community over the last years. Currently it is being replaced by Subversion [480] (SVN).

Our approach for populating the RHDB is based on the versioning system log files, thus it can be applied to 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 3.1 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 3.1. A CVS log file chunk of `mozJSCodeLib.cpp`

Bugzilla [95] is a bug tracking system that is used heavily in the open source community. 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 system.

A typical bug report contains the following pieces of information: An *id* which unequivocally 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 interested in being notified of the bug fixing progress (*CC*).

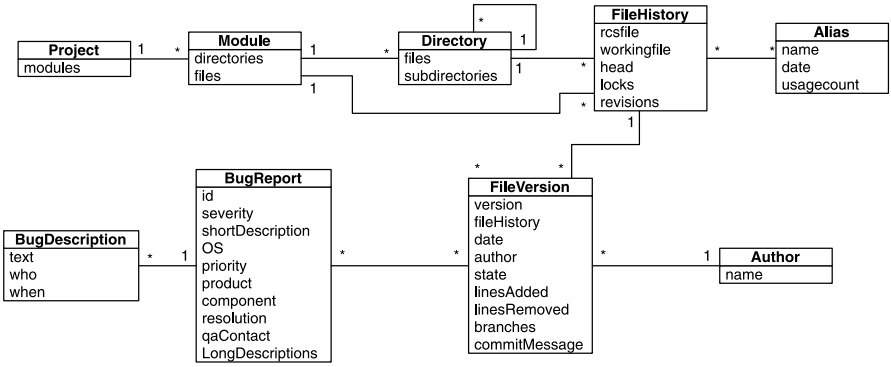


Fig. 3.2. The RHDB data model

3.3.1 The RHDB Model

Figure 3.2 shows the core of 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.

3.3.2 Populating the RHDB

Figure 3.3 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 depending on the size of the system can take several hours) is executed in batch mode. The main steps of the process are:

1. The latest 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.
2. For each file, the data about all its commits (its history) is stored in the database as well as a link to the actual file.
3. Every time a reference to a bug is found in the commit message (the comment written by the author at commit time), the corresponding bug report is retrieved

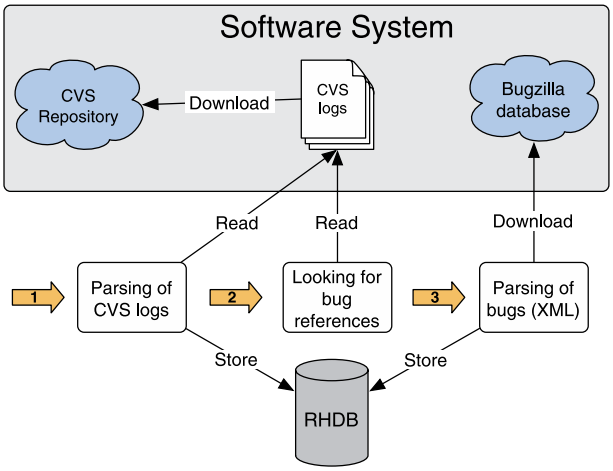


Fig. 3.3. The RHDB populating process

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.

3.3.3 Related Work

Several approaches were proposed to create and populate an underlying model of an evolving software system. These approaches vary according to which information they consider (e.g., only source code repository or also bug tracking system and mail archives), which data sources they support (e.g., only CVS or also SVN, ClearCase etc.) and how these sources are linked to each other.

The previously presented RHDB is based on the CVS versioning system and the Bugzilla bug tracking system, where the links between the two sources are built as presented in Section 3.3.2. The main contribution of the RHDB is that it was the first to link CVS artifacts and Bugzilla problem reports.

Two other approaches similar to the RHDB, also based on CVS and Bugzilla, but which also use other sources of information are Hipikat [133, 511, 510] by D. Čubranić et al. and softChange [196, 195] by D. German. Both techniques use information from mail archives and, in addition, Hipikat also considers data from documentation on the analysed project website (if available).

In both approaches the links between different information sources are inferred based either on conventions (e.g., in some projects there is a convention to include in the commit comment a reference to the bug tracking system entry) or heuristics (e.g., it is likely that the author of a bug fix has committed a source code revision close to the time that the problem report was closed in the bug tracking system).

A common problem encountered while linking mail archives with CVS repository is that people tend to have multiple e-mail addresses, which might not be the same as the ones recorded in the CVS log files [197].

In the Hipikat model (see Figure 3.4), a message is a mail in the mail archive, a file version corresponds to a CVS commit in the repository (a revision), a change task is a Bugzilla problem report and a document is a design document retrieved, for example, from the project web site.

In the softChange architecture (see Figure 3.5), we see two main components: The Trail Extractor and the Fact Enhancer. The Trail Extractor retrieves the following software trails: CVS logs, Bugzilla problem report, ChangeLogs and releases of the system (tar files distributed by the software team). The Fact Enhancer uses the retrieved software trails to generate/infer new facts. For example it reconstructs the commit-set, since the commit operation in CVS is not atomic, it links CVS artifacts with Bugzilla problem report or messages from the mail archives, etc.

The information stored by Hipikat forms an "implicit group memory" (where group stands for group of developers) which is then used to facilitate the insertion of newcomers in the group, by recommending relevant artifacts for specific tasks. The data retrieved and processed by softChange is used for two types of software evolution analysis and visualization: (i) Statistics of the overall evolution of the project, using histograms where the x axis usually represents the time dimension and (ii) analysis of the relationships among files and authors, using graphs where authors and/or files are represented as nodes and their relationships as edges.

Another approach similar to the RHDB is the Kenyon framework [58] by J. Bevan et al. Kenyon provides an extensible infrastructure to retrieve the history of a software project from a SCM repository or a set of releases and to process the retrieved information. It also provides a common interface based on ORM (Object-Relational Mapping) to access to processed data stored in a database.

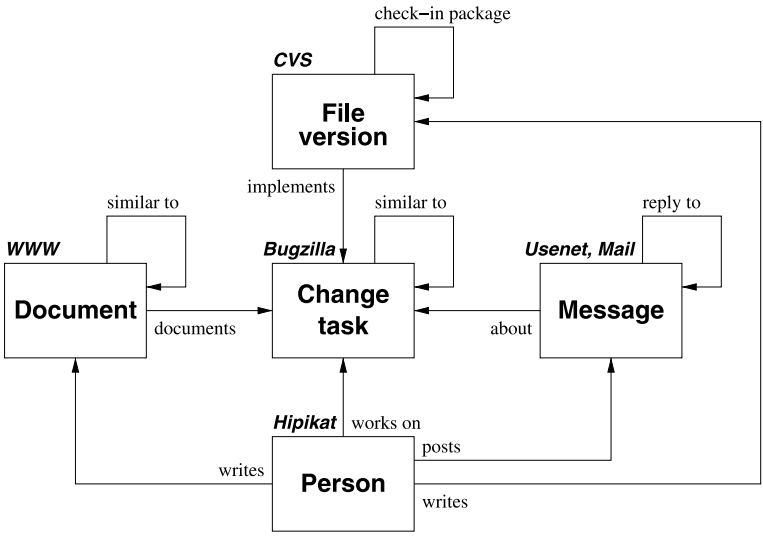


Fig. 3.4. The Hipikat model [511] ©[2005] IEEE

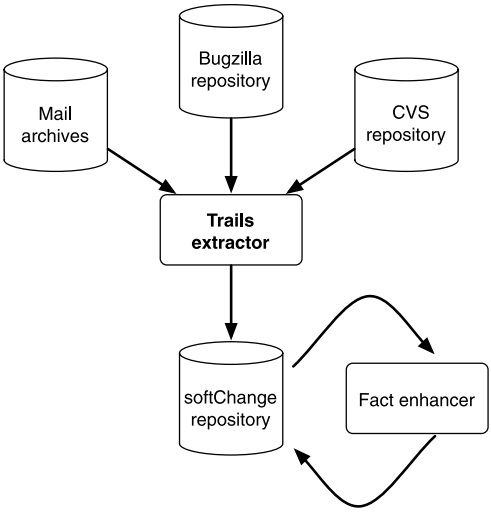


Fig. 3.5. The softChange process [195]

Figure 3.6 shows the high-level data flow architecture of Kenyon. The DataManager class is the execution entry point: It reads a configuration file and invokes the other components, i.e., the Configuration Retrieval, the Fact Extractor and the Object Data Storage. The SCMInterface class isolates Kenyon from the concrete implementation of different SCM systems. The FactExtractor and MetricLoader abstract classes are the API points for specific tool invocation extensions. This means that users of Kenyon are free to attach their own external Fact Extractor and Metric Loader tools (typically analysis-specific). Besides this extension, Kenyon offers predefined fact extractor and metric loader tools. Kenyon saves the results from each

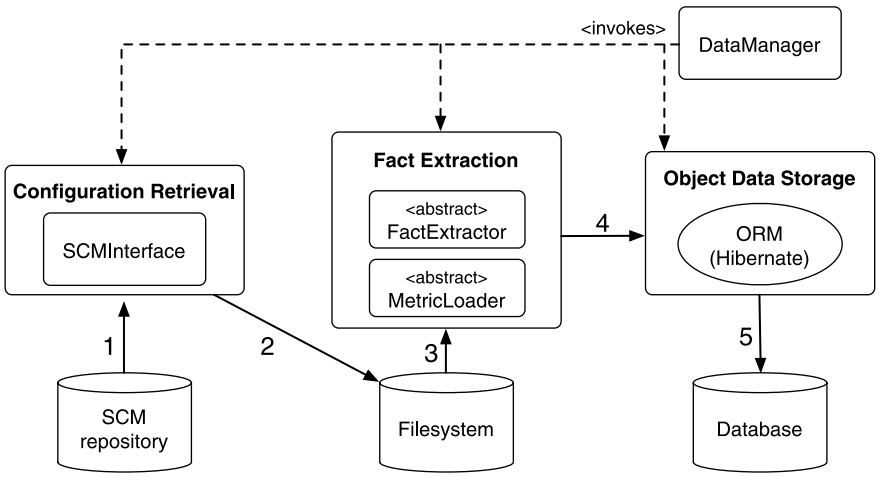


Fig. 3.6. The high-level data flow architecture of Kenyon [58], ©ACM, 2005

processed configuration to a database. An ORM mechanism is provided to help automate the storage to and retrieval of Java object from the database.

As depicted in Figure 3.6 Kenyon retrieves information from SCM only (or filesystem, i.e., set of releases), without considering other sources, such as bug tracking system or mail archives. On the other hand Kenyon supports several SCMs, namely: CVS, SVN, ClearCase and sets of releases in the filesystem.

A common aspect of Kenyon and the RDHB is that both store the data for later evolution analyses, while for softChange and Hipikat the task for using the data is already defined.

3.4 Software Evolution Analysis

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. In the following, we present some software evolution analysis problems and describe our techniques to tackle them.

3.4.1 Analysing Developer Effort

The first software evolution problem we address concerns development effort. We want to answer questions such as: 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 "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.

Version control systems record the information necessary to answer these questions, as each ach artifact has a list of versions corresponding to commits, and the list of authors who performed the commits[3]. The problem is how to represent and aggregate this large amount of low-level information[4] to get an insight into the team structure and to understand who are the responsible/s of a software entity, scaling from a module down to the individual file.

Our approach is based on the "*Fractal Figure*" [139, 136] 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

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

[4] 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 hundreds of thousands of commit-related data to analyse.

which extent. A fractal 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. For more details on the layout algorithm and the expressive power of Fractal Figures see [139].

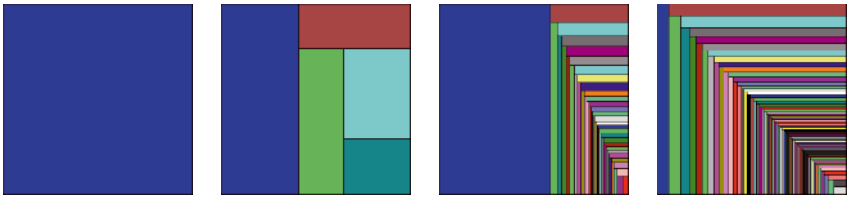
Fractal Figures allow software entities to be categorized in terms of effort distribution among developers following the *gestalt principle*. We defined four visual patterns representing four development models, depicted in Figure 3.7: (a) One developer, (b) few balanced developers, (c) one major developer and (d) many balanced developers.

Development patterns allow us to categorize entities according to the way they were developed from an authors' 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:

$$\text{Fractal Value} = 1 - \sum_{a_i \in A} \left(\frac{nc(a_i)}{NC} \right)^2, \quad \text{with} \quad NC = \sum_{a_i \in A} nc(a_i) \tag{3.1}$$

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. (3.1) 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 [309]. 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. We map a metric measurement of the size of the figure. The metric can be structural such as LOC



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

Fig. 3.7. Development patterns based on the *gestalt* of Fractal Figures [139] ©[2005] IEEE

or evolutionary such as number of commits, number of bugs, number of lines added etc.

In the following we present different example scenarios which show how to use Fractal Figures to address the problem of understanding development effort distribution.

Detecting a Major Developer

Figure 3.8 shows the `webshell` directory hierarchy of Mozilla. Fractal Figures represent directories containing at least one file, while grey figures represent 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 whom 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 a single person to ask questions about the code. The reverse engineer will need support of other tools such as CodeCrawler [310] or BugCrawler [138]. This information is not complex or hard to get, but the value of the Fractal Figure visualization is that it conveys this information in a context (the hierarchy in this case), easy and fast to read, and with the same visual principle for all the software entities to which it is applied.

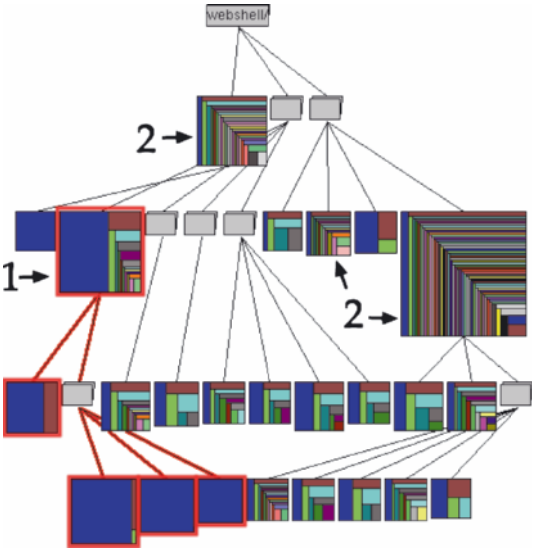


Fig. 3.8. Fractal Figures applied to the `webshell` hierarchy of Mozilla [139] ©[2005] IEEE. The size metric maps the directory size in terms of number of contained files

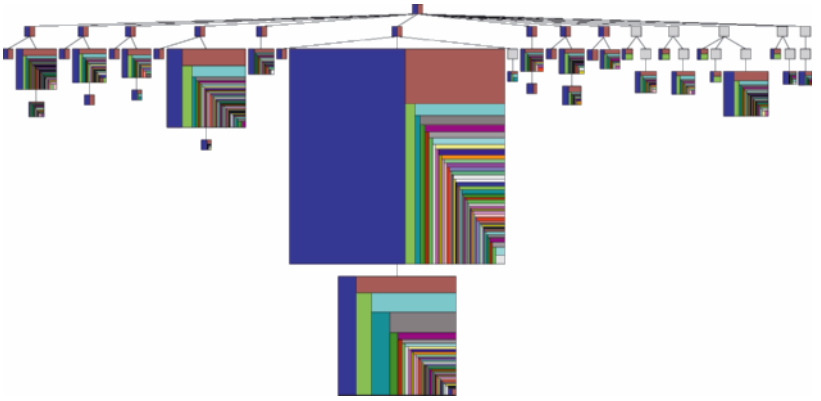


Fig. 3.9. Fractal Figures applied to the `network/protocol` hierarchy of Mozilla. The size metric maps the number of bug reports

Re-Assessing Development Team Formation

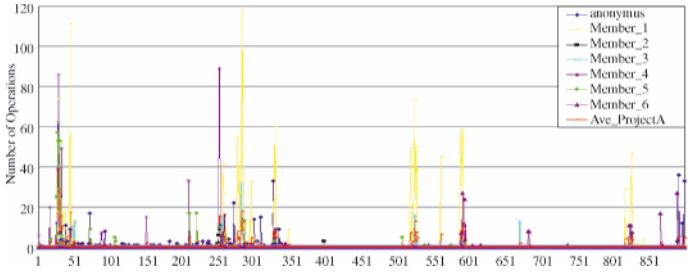
Figure 3.9 shows an example with the network protocol implementation of Mozilla. Most of the directories which introduced bugs have a many balanced developer patterns, but one which has a one major developer pattern: `network/protocol/http/src`. This directory is responsible for most of the bugs generated in the `network/protocol` hierarchy. Such a view can be valuable for a project manager or an analyst. It shows that a re-assessment of the formation of the development team is needed, given the high number of bugs and one major developer pattern of the `network/protocol/http/src` directory.

Related Work

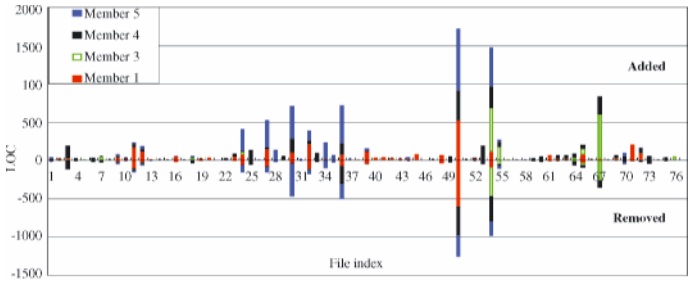
Many software evolution analysis techniques focus more on the developers and their interaction with software artifacts than on software artifacts themselves. Liu et al [330] applied the CVSChecker tool to analyse CVS log files with the aim of understanding author contributions and identifying patterns. They wanted to study the open source development process and to understand what activities are carried out in open source project and by whom. The CVSChecker tool supports the analysis of the performance of an individual developer and the effort distribution patterns of teams.

CVSChecker has a set of parsers which extract information from the CVS source code repository and store them in a relational database. The tool then uses this information to produce four kinds of visualizations:

1. Temporal distribution of CVS activity, for each developer (see Figure 3.10a);
2. Distribution of CVS operation types, for each developer;
3. Distribution of CVS operation types, for each file;
4. Added and removed lines of code (LOC) by each developer, on each file (see an example in Figure 3.10b).



(a) Temporal distribution of CVS activity for each developer



(b) Added and removed LOC for each developer, for each file

Fig. 3.10. CVSChecker example visualizations [330]

The visualizations are used in [330] to extract development patterns to characterize the open source development process.

Gîrba et al. [203] defined a measurement for the notion of code ownership in CVS repositories. They defined that a developer is the owner of a file if he/she owns the major part of it in terms of lines. He/she owns a line of code if he/she was the last one that committed a change to that line in the repository. Based on that principle, they introduced the Ownership Map visualization, which shows the evolution of a software project, according to the following rules (summarized in Figure 3.11):

- Each file is represented as a colored line;
- The x axis represents the time dimension, from left to right;
- Each commit of a file in the repository is represented as a colored circle on the corresponding line;
- Each developer is represented by a color;
- Commits (circles) are colored according to the developer who did them, while pieces of histories of files (corresponding to pieces of lines) are colored according to the owner of the file, during the considered time interval.

In [203] the authors used the Ownership Map visualization to define development patterns such as *monologue* (a period where all the changes and most files belong to the same author), *takeover* (a developer takes over a large amount of code in a short amount of time), *team work* (two or more developers commit a quick succession of

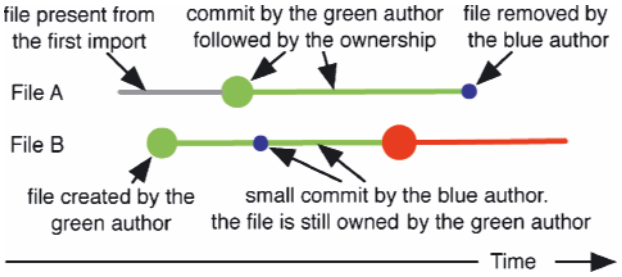


Fig. 3.11. The principles of the Ownership Map visualization [203] ©[2005] IEEE

changes to multiple files) etc. The patterns were defined with the aim of characterizing different developer behaviors.

In [528], Voinea and Telea presented a similar visualization, in which CVS files are represented as colored lines and the color represents the developer. The visualization is implemented in CVSgrab, a tool which also supports the visualization and analysis of activities in the repository. In [528] the authors applied a cluster algorithm on the visualizations to put files (lines) with similar development (with respect to either the authors or the activity) close to each other. The aim of their work was to allow developers and project managers to visually explore the evolution of a software project in a way that facilitates the system and process understanding.

Voinea et al. also presented the CVSscan tool [529], based on CVSgrab for extracting the data from the CVS repository. The tool can visualize the evolution of CVS files by visualizing the evolution of individual lines. CVSscan provides three types of color encoding to associate the color of each code line to its author. This visualization is used in [529] to understand who performed modifications on the code and where, thus facilitating the development process understanding.

Author information stored in CVS repositories was also used in the context of social networks. In [68] Bird et al. created social networks or email correspondents from OSS email archives. They linked emails with CVS accounts to analyse the relationship of email activity and commit activity and the relationship of social status with commit activity. The case study they conducted on the Apache HTTP server project indicated a strong relationship between the level of activity in the source code, and a less strong relationship with document change activity. They also found out that developers (people with email *and* CVS accounts) play a much more significant social role than other participants in the mail archives.

3.4.2 Change Coupling Analysis

Change coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system. This co-change information can either be present in the versioning system, or must be inferred by analysis. For example SVN marks co-changing files at commit time as belonging to the same *change set* while in CVS the files which are change (i.e., logically) coupled must be inferred from the modification time of each individual file.

The Evolution Radar [137, 140] is an interactive visualization technique for analysing change couplings to detect architecture decay and coupled components in a given software system. It addresses the following questions: What are the components (e.g., modules) with the strongest coupling? Which low level entities (e.g., files) are responsible for these couplings?

Figure 3.12 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 all others clockwise (see Figure 3.12). 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.

Arbitrary metrics can be mapped on the color and the size of the circle figures. 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:

$$CC(M, f) = \max_{f_i \in M} CC(f_i, f) \tag{3.2}$$

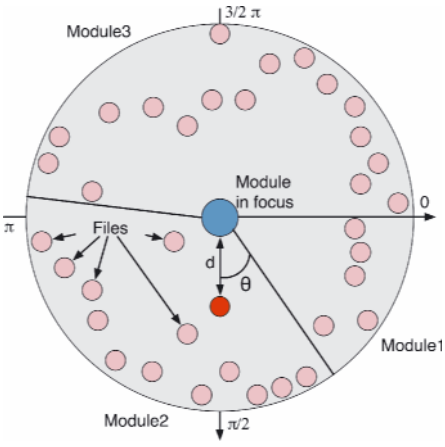


Fig. 3.12. The structural principles of the Evolution Radar [137] ©[2006] IEEE

$CC(M, f)$ is the change coupling between the module M in focus and a given file f and $CC(f_i, f)$ is the change coupling between the files f_i and f. It is also possible to use other group operators instead of the maximum such as 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 change dependencies.

The value of the coupling between two files is equal to the number of transactions which include both files. Since change transactions are not recorded by CVS we reconstruct them using the sliding time window approach proposed by Zimmermann and Weißgerber in [566], 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 [137, 566].

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 for the whole history of the system we would obtain misleading results. Figure 3.13 shows an example of such a situation.

Figure 3.13 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 9 shared commits on a total of 17, 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 analyse 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 [202]. In other words, if two files were strongly coupled at the early phases 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.

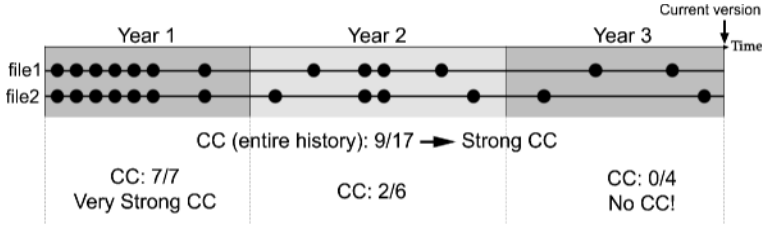


Fig. 3.13. An example of misleading results obtained by considering the entire history of artifacts to compute the change coupling value: We obtain a strong change coupling, while file1 and file2 are not coupled at all during the last year

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 analyse 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. The highlighting consists in using a yellow border for the tracked files and in showing a text label with the name of the file. Like this it is possible to detect files with a strong change coupling with respect to the latest period of time and then move backward in time and analyse 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 in a given software system? Which files are responsible for these evolutionary dependencies? In the following we apply the radar on ArgoUML, 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 analyse the evolution of couplings and to identify 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[5] We focused our analysis on the three largest modules `Model`, `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.

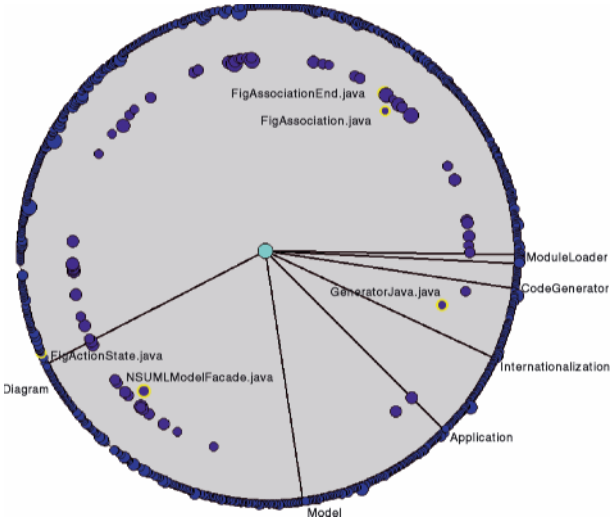
Figure 3.14b shows the Evolution Radar for the last six months of history of the `Explorer` module. From the visualization we see that the coupling with `Diagram` is much stronger than the one with `Model`, although the documentation states that the dependency is with `Model` and not with `Diagram`. 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 Figure 3.14a 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 generated another radar, shown in Figure 3.15 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. The fact that they belong to different modules buries this hidden dependency.

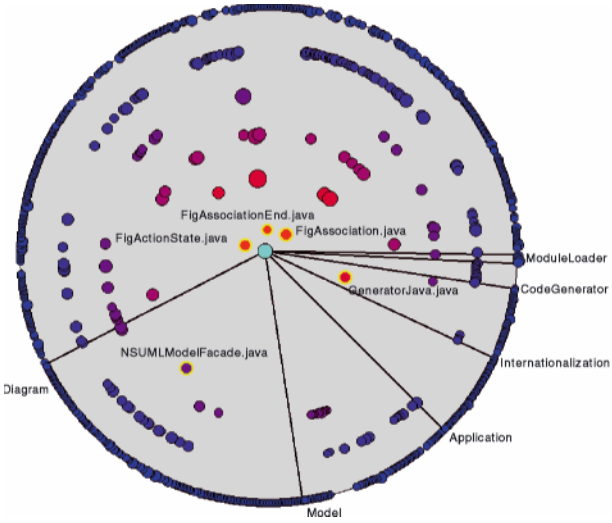
The visualization in Figure 3.14b shows that the file `GeneratorJava.java` is an outlier, since its coupling is much stronger with respect to all the other files in the same module (`CodeGeneration`). By spawning the group composed of `GeneratorJava.java` we obtained a visualization very similar to Figure 3.15, 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 3.14a and Figure 3.16a) reveals that `GeneratorJava.java` is an outlier since January 2003. This long-lasting dependency indicates design problems.

[5] We did not consider some modules for which the documentation says "They are all insignificant enough not to be mentioned when listing dependencies." [19].



(a) January to June 2005.



(b) June to December 2005.

Fig. 3.14. Evolution Radars for the `Explorer` module of ArgoUML in 2005 [137] ©[2006] IEEE

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 3.14b and Figure 3.14a show that during 2005 the file `NSUMLModelFacade.java` in the `Model` module had the strongest coupling with

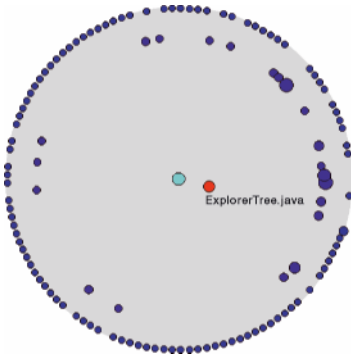


Fig. 3.15. Details of the change coupling between the `Explorer` module and the files `FigActionState.java`, `FigAssociationEnd.java` and `FigAssociation.java` [137] ©[2006] IEEE

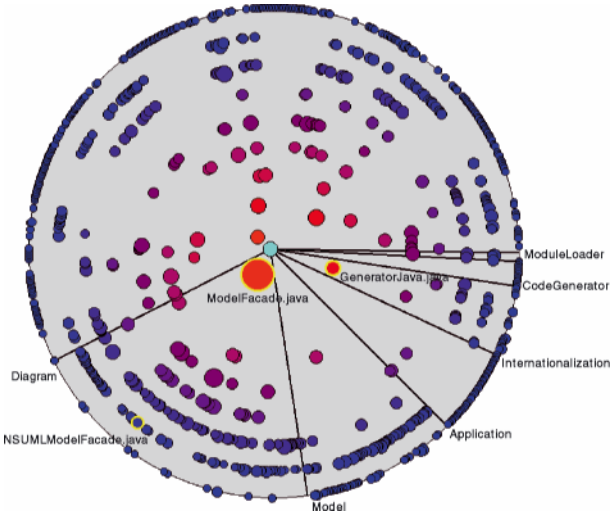
`Explorer` (module in the center). Going six months back in time, from June to December 2004 (see Figure 3.16a), we see that the coupling with `NSUMLModelFacade.java` was weak, while there was a very strong dependency with `ModelFacade.java`. This file was also heavily modified during that time interval, 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 3.16b). By looking at its source code we found out that this was a God class [439] with thousands of lines of code, 444 public and 9 private methods, all static. The `ModelFacade` class is not present in the other radars (Figure 3.14b and Figure 3.14a) because it was removed from the system in January 2005. By reading the source code of the most coupled files 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[6], 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.

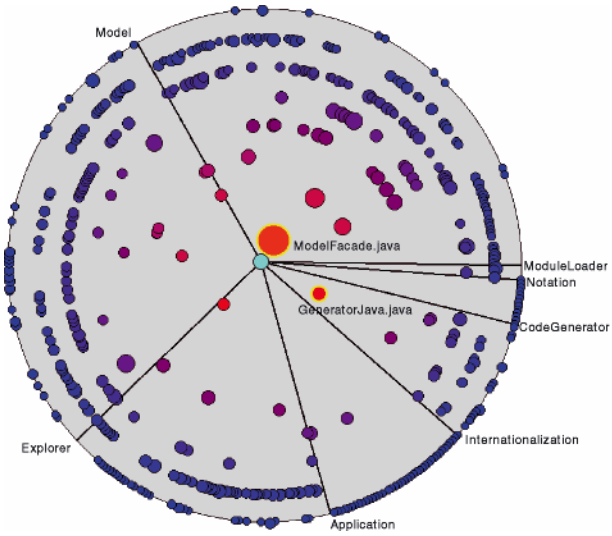
We showed examples of how to use the Evolution Radar to detect problematic parts of the ArgoUML system, which represent good candidates for reengineering. The main findings of the discussed example scenario are:

- The `Diagram` and `Explorer` modules are the most coupled. Since this dependency is not mentioned in the module relationships page in the documentation,

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



(a) Explorer module



(b) Diagram module

Fig. 3.16. Evolution Radars of the `Explorer` and `Diagram` modules of ArgoUML from June to December 2004 [137] ©[2006] IEEE

either the modules should be restructured to decrease the coupling or at least 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 analysed and, if possible, refactored. `GeneratorJava.java` has a persistent coupling with the `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.

Related Work

The concept of change (i.e., logical) coupling was first introduced by Gall et al. [185] to detect implicit relationships between modules. They used logical coupling to analyse the dependencies between the different modules of a large telecommunications software system and showed that the approach can be used to derive useful insights on the architecture of the system. Later the same authors revisited the technique to work at a lower abstraction level. They detected logical couplings at the class level [187] and validated it on 28 releases of an industrial software system. The authors showed through a case study that architectural weaknesses such as poorly designed interfaces and inheritance hierarchies could be detected based on logical coupling information.

Ratzinger et al. [433] used the same technique to analyse the logical coupling at the class level with the aim of learning about, and improving the quality of the system. To accomplish this, they defined *code smells* based on the logical coupling between classes of the system.

Other work has been performed at finer granularity levels. Zimmermann et al. [567] 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. Breu and Zimmermann [80] 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 [561] Bouktif et al. [77] improved precision and recall of co-chancing files detection with respect to previous approaches. They introduced the concept of change patterns in general and the particular Synchrony change-pattern for co-changing files. They proposed an approach to detect such change-patterns in CVS repositories based on dynamic time warping.

Similar to the Evolution Radar the EvoLens visualization technique can be used to analyse the change coupling relationships between source files and software modules. Instead of radar views it uses a graph-based visualization that allows the user to navigate the change coupling information from the level of modules down to the source files. It allows the user to reveal detailed change couplings to the cost of always having a radar view of the whole system. The basic ideas and underlying concepts of the EvoLens Views have been developed by Ratzinger et al. [432].

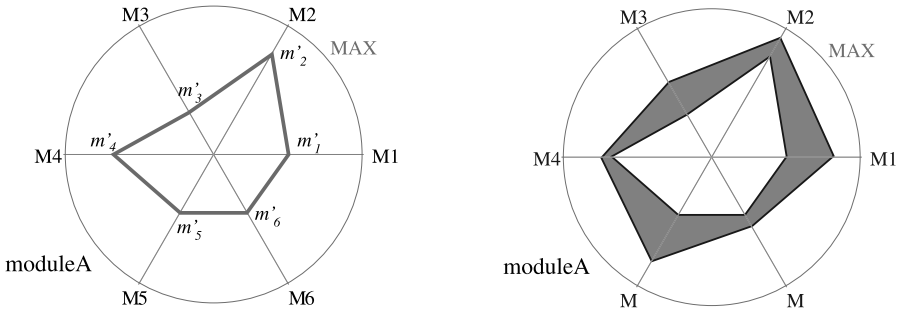
Beyer and Hassan in [59] presented the Evolution Storyboards, a visualization technique for software evolution that offers dynamic views. The storyboards emphasizes the history of a project using a sequence of panels, each representing a particular time period in the life of the project. To create each panel they compute a co-change graph and use a layout in which the stronger the coupling between two files is, the closer they are placed, thus revealing clusters of frequently co-changed files. They showed two main applications of the tool: First they analysed how the structure of a software system decayed or remained stable over time, by comparing the clusters of co-changed files with the authoritative system decomposition. In the second application, they detected files which implement cross-cutting concerns, by detecting the files which are always moving from panel to panel, meaning that these files are coupled (close in the layout) with many others during the life of the project.

In [178] Fischer and Gall presented EvoGraph, a lightweight approach based on the RHDB data to evolutionary and structural analysis of software systems. They compounded change history and source code information spaces in a single approach, to support the comprehension about the interaction between evolving requirements and system development. The EvoGraph technique is composed of five phases: (1) File selection: Source files which exhibit an extraordinary logical coupling with respect to cross-cutting change transactions are selected. (2) Co-change visualization. (3) Fact extraction: For the selected files in the preceding step, the detailed change transaction information is collected from the RHDB and as result change vectors are created for every file within a transaction. (4) Mining: The change vectors are the input to mining of change transaction data step; The output is a description of the longitudinal evolution of structural dependencies of selected files. (5) Visualization: The structural dependencies are visualized in an electrocardiogram style diagram.

3.4.3 Trend Analysis and Hot-Spots 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 which stem from the decomposition of a system into manageable implementation units. Such units, for example, are packages, source code directories, classes, or source files. The objective of ArchView 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 ArchView have been developed in the work of Pinzger et.al [417].

ArchView obtains metric values of each module and dependency relationship from the RHDB and assigns them to a feature vector m. Feature vectors are tracked



(a) One release. (b) Two releases.

Fig. 3.17. Kiviat diagram of moduleA representing measures of six source code metrics $M_1, M_2, ..., M_6$ of one release

over the selected n releases and composed to the evolution matrix E. The values in the matrix quantify the evolution of a software module:

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

The matrix contains n feature vectors with measures of i metrics. Evolution matrices are computed for each module. They form the basic input to our ArchView visualization approach that we will present next.

The ArchView approach uses the Polymetric Views visualization technique presented by Lanza et al. [309]. Instead of using rectangles to present modules and metric values ArchView uses *Kiviat* diagrams which are also known as *Radar* diagrams. These diagrams are suited to present multiple metric values available for a module as described next.

Figure 3.17 (a) 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:

$$l\left(m'_i\right) = \frac{m'_i * cl}{max(m'_i)} \tag{3.3}$$

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 the one shown in Figure 3.17.

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 it represents no particular 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.

Figure 3.17 (b) depicts the same set of measure for `moduleA` but this time of two releases. By filling the area between the releases the change of metric values are highlighted. To distinguish the changes between different source code releases we use gray tones. This allows the user to spot trends in metric values as we will show in the following two analysis scenarios.

Analysing the Size and Complexity of Software Modules

The first scenario concerns an analysis of the growth in size and program complexity of software modules. We demonstrate this by visualizing typical size and program complexity metrics taken from three releases 0.92, 1.3a, and 1.7 of Mozilla content and layout modules. We configure ArchView with the following set of metrics: Halstead intelligent content (`HALCONT`), Halstead mental effort (`HALEFF`), Halstead program difficulty (`HALDIFF`), McCabe Cyclomatic Complexity (`CYCL`), and lines of code (`LOC`). The resulting view is depicted in Figure 3.18.

The interesting modules are represented by Kiviat diagrams with large, filled, gray polygons. They indicate strong increase and decrease in the size and program complexity of software modules whereas small polygons represent more stable modules. Following this guideline we can easily see that `NewLayoutEngine` and `DOM` (Document Object Model) are the two largest and most complex modules in the Mozilla content and layout implementation. For instance, in release 1.7 `DOM` comprises 197.498 and `NewLayoutEngine` 156.438 lines of C/C++ code. In contrast, the `XML` module comprises 23.471 lines of code.

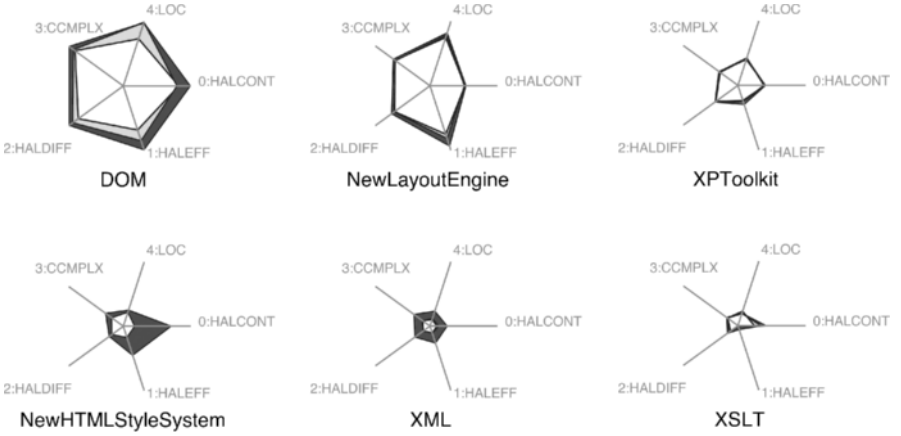


Fig. 3.18. Kiviat graph of six Mozilla content and layout modules showing the growth in program size and complexity. Light gray polygons indicate changes between releases 0.92 and 1.3a. Dark gray polygons show changes between 1.3a and 1.7

Large gray polygons are contained by the Kiviat diagrams of the modules `DOM`, `NewHTMLStyleSystem`, and `XML` indicating that the implementation of these three modules changed the most. Looking at the selected size and program complexity metric values we found out that the values of the three modules first increased from release 0.92 to 1.3a and then decreased from release 1.3a to 1.7. For instance, the `HALCONT` metric of the `DOM` module from release 0.92 to release 1.3a increased from 15.167 to 18.228 followed by a decrease to 14.714 in release 1.7. Apparently, from release 0.92 to 1.3a functionality was added to these three modules which during the implementation of the last release then was refactored or removed. In comparison to these three 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 a past release will be likely to change in future releases the three modules `DOM`, `NewHTMLStyleSystem`, and `XML` are the candidates who are critical for the evolution of the content and layout implementation of Mozilla.

Detecting System Hot-Spots

System hot-spots are modules with high activity indicated by different measures such as the number of problems affecting a module or the number of changes in a module. In this scenario of analysing system hot-spots we focus on providing answers to the following three questions: Which are the modules with frequent bugs? Which are the most critical modules? Which modules became stable? The answers to these questions can be found in the RHDB in particular in the Bugzilla data. We quantify the criticality and stability of a software module by the number of source code modifications (i.e., CVS log entries) performed for fixing bugs that were reported

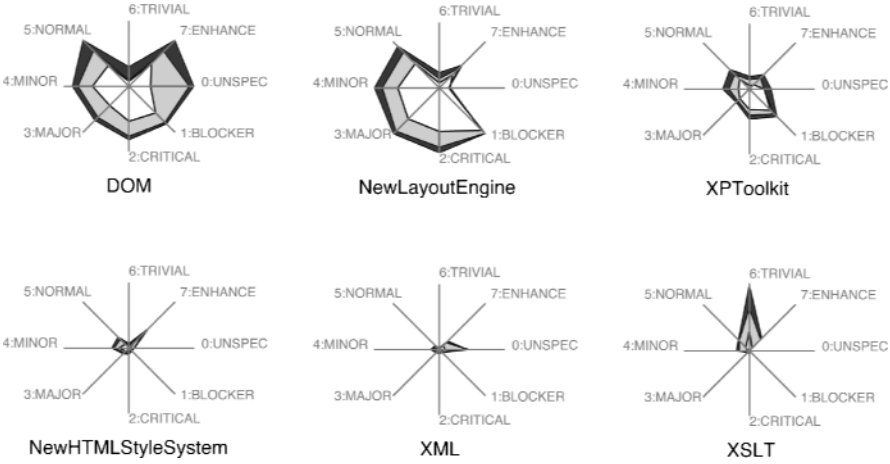


Fig. 3.19. Kiviat graph of six Mozilla content and layout modules showing criticality and stability. Light gray polygons indicate changes between releases 0.92 and 1.3a. Dark gray polygons show changes between 1.3a and 1.7

during a given observation period such as the time between two releases. To further detail criticality and stability we take the different severity levels of bugs ranging from blockers, to minor and trivial bugs into account (severity levels are taken from the Bugzilla repository). This leads to the following set of measures: number of modifications for bugs with unspecified severity (`UNSPEC`), number of modifications for blocker bugs (`BLOCKER`), number of modifications for critical bugs (`CRITICAL`), number of modifications for major bugs (`MAJOR`), number of modifications for minor bugs (`MINOR`), number of modifications for normal bugs (`NORMAL`), number of modifications for trivial bugs (`TRIVIAL`), and number of modifications for suggested enhancements (`ENHANCE`). We configured ArchView with these measures and selected six Mozilla content and layout modules from release 0.92, 1.3a, and 1.7. The resulting system hot-spot view is depicted by Figure 3.19.

The large gray polygons of the `DOM` and `NewLayoutEngine` indicate that these two modules got the highest number of CVS log entries for fixing bugs. For instance, up to release 1.7 `DOM` got 254 modifications from 130 bugs rated as blocker and 904 modifications from 487 bugs rated as critical. `NewLayoutEngine` got 309 CVS log entries from 121 blocker and 1.097 log entries from critical bug reports. Interestingly, most of the modifications in the implementation of these two modules were due to bugs of high severity and only few due to trivial bugs. This fact is indicated by the cut of the `TRIVIAL` measure occurring in both Kiviat diagrams. Compared with these two modules the other content and layout modules needed less modifications to fix bugs. For instance, `XSLT` got 7 modifications due to 3 blocker bugs and 48 modifications due to 12 critical bugs. Interestingly, the Kiviat of `XSLT` shows a peak in the number of trivial bugs (`TRIVIAL`). 123 modifications due to 3 trivial bugs were performed which is more than twice as much as the values of the other five modules (e.g., `DOM`

got 55 and `NewLayoutEngine` 43 modifications). Apparently, a large number of files had to be touched to fix the three trivial bugs. For instance, 56 files were modified to fix bug #88623. This raises the question about how "trivial" this bug was when modifications had to be done in 56 source files.

Concerning the criticality and stability of software modules we investigated the trend of these measures. More specifically, stable modules are indicated by small dark gray polygons meaning less bugs and modifications during the development of release 1.7. According to Figure 3.19 the modules `NewHTMLStyleSystem` and `XML` were affected by almost zero changes. They represent the most stable content and layout modules. The other four modules seem to be more critical as indicated by larger light and dark gray polygons whereas the light gray polygons dominate the diagrams. This means that a large amount of bug fixing took place in the time period from release 0.92 to 1.3a which then decreased in the time of developing release 1.7. For instance, the number of modifications for fixing blocker bugs decreased from 65 between the releases 0.92 and 1.3a to 8 modifications between 1.3a and 1.7. That is a clear indicator that the other four modules were critical in previous releases but became stable in release 1.7.

Related Work

A number of approaches have been developed that address the visualization of data of several software releases. For instance, Riva et al. use 2D and 3D graphs to visualize and navigate the release history of a software system [188]. Time is visualized on the z coordinate expressed in release sequence numbers (RSN). The structure of each software release is visualized using 3D graphs with a tree layout. Each graph is spatially positioned along the z-coordinate showing the sequence of releases. A cube denotes a subsystem or a software module. Edges indicate the decomposition of each release into subsystems and modules. One measure can be mapped to the 3D-graphs using the color attribute. For instance, they mapped the version number of a module such that a module that is not present in a release is represented by a black cube, a module in version 1 is represented by a red cube, in version 2 by a blue cube, etc.

An approach similar to the approach of Riva et al. is presented by Collberg et al. in [121]. They developed GEVOL, a graph-based system for visualizing the evolution of software systems. Each state of a system is represented by a graph. Colors are applied to indicate change over time such as when particular parts of the program were first created and modified, which programmer modified which parts, or which parts have grown in complexity. All nodes start out being red, then grow paler for every time they have remained unchanged. When a node changes again it return to red. When a user notices an interesting event, such as a code segment changes frequently, he can click on a node to examine the set of authors who have affected these changes.

Lanza developed an approach called the Evolution Matrix [308]. Instead of using a tree layout Lanza uses a matrix layout. The Evolution Matrix displays the evolution of the classes of a software system. Each column of the matrix represents a version of the software, while each row represents the different versions of a class. Figure 3.20

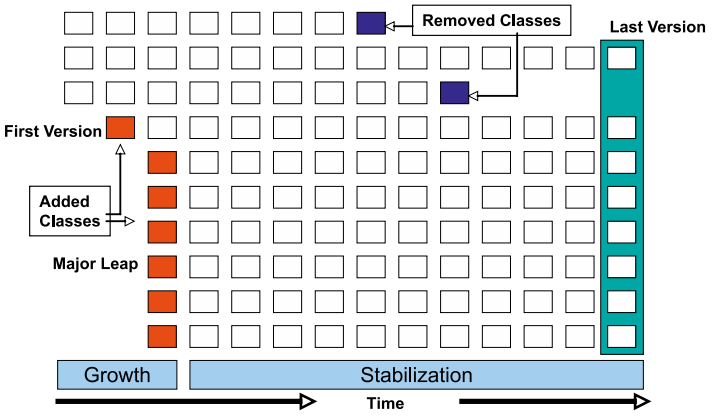


Fig. 3.20. Evolution Matrix with four typical characteristics of an evolving software system [308], ©ACM, 2001

shows an example of an Evolution Matrix with typical characteristics of a system such as the classes of the first version, removed classes, a major leap in the evolution, and the last version of the system.

Following the Polymetric View principle a number of measures can be mapped to the width, height, and color of rectangles that represent classes. Recurring patterns in the matrix arise that led to a categorization of the evolution of classes. For instance, a class that is alternately growing and shrinking in size is called `Pulsar`. Pulsar classes can be seen as hot-spots in the system: for every new version of the system changes on a pulsar class must be performed. Other categories of classes, for instance, are `Supernova` (class suddenly explodes in size) or `Dayfly` (class with a short life-time).

In [548] Wu et al. present Evolution Spectographs, a visualization technique that combines metrics and gradient colors to portray the evolution of software systems. A spectograph is shown as a matrix similar to the Evolution Matrix in which time is presented on the X axis and the dimension of files is presented on the Y axis. A row in the matrix represents the change history of a file and a column stores change metrics for all the files during a particular time period. Each cell represents a file in a particular period. After a file is changed its color becomes lighter and lighter as long as there is no change made to that file. In other words, the file starts to cool down if no future change occurs to it. Using this color function Evolution Spectographs can be used to highlight system growth and dependency change in one chart.

3.5 Conclusion

Mining software repositories is a fairly recent research topic that has been embraced by both the software evolution and the empirical software engineering community. As we have seen in this chapter there are two major challenges:

- *Technical challenge.* The challenge resides in modeling and handling various kinds of informations. The sheer amount of information available in source repositories also poses scalability problems that have however been tackled to a large extent. As we have seen, in most cases researchers have chosen to use relational databases to handle the data as they allow for easy querying.
- *Conceptual challenge.* Once the data is retrieved and modeled, a major challenge resides in doing something meaningful with the available data. As we have seen in the various approaches, at the beginning there is always a number of research questions that need to be answered, and subsequently researchers develop the necessary mechanisms to answer those questions. A heavily used technique in this case is visualization, as it allows (if well chosen) to detect patterns in the sea of data that one has to navigate.

As a major future challenge we see the current dichotomy between forward engineering and software evolution. We believe that software repositories, currently mostly used for retrospective analyses, need to become an integral part of any software project, and as such should not be separated from the most recent version, which is usually the focus of all maintenance efforts.