POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Software Archaeology - Reconstructing the Evolution of Software Systems

Relatore:
Prof. Carlo Ghezzi
Correlatore:
Prof. Michele Lanza

Tesi di Laurea di:
Marco D'Ambros
matr. n. 639633

Anno Accademico 2003-2004

Abstract

Real world software systems require continuous change to satisfy new user requirements, adapt to new technologies and repair errors. As time goes by, software increase in size and complexity, and their design gradually decay unless work is done to maintain the systems. The problem of understanding the evolution of software has become a vital matter in today's software industry.

In this thesis we propose an approach to tackle this problem, composed of two self-contained parts. The first is aimed at collecting historical information regarding the system, and storing it in a structured way. The second part exploits visual techniques to analyze both evolutionary and structural aspects of the software, at different granularity levels:

- Coarse-grained, concerning the overall structure of the system.
- Fine-grained, concerning the inner structure of the modules composing the system.

Based on the combination of fine-grained and coarse-grained information, we present a top-down methodology to lead the entire analysis of software systems. We finally validate our approach on the Mozilla case-study.

Acknowledgements

First of all, I want to thank Prof. Dr. Michele Lanza for his kind supervision and Prof. Dr. Carlo Ghezzi for giving me the opportunity to carry out my thesis in such an uncomplicated way.

I also want to thank Prof. Dr. Harald Gall and Martin Pinzger for giving me useful suggestions about software evolution. Thanks also go to European Science Foundation for supporting me journeys to Zurich.

Special thanks go to all my friends, in particular to my current and previous "co.ca", to "la comune" fellows, to the Ultimate Milano players, to the "Malt Storm" partners and to my Chicago roommates, who shared with me such a nice experience.

And last but not least, I want to thank all the members of my family for being so patient towards my strange life-style, and in particular to my sister for providing me language suggestions and to "potino" for being so amazing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"*The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly*" [GHJV95].

This principle, known as "*Design for change*", is one of the most important and widely accepted of the Software Engineering field. Software is a living entity in a living context. It **must** adapt to dynamic environments, changing requirements and new technologies.

The aging of software is a problem too often underestimated in industrial settings. In these context the strict time constraints, related to market competition, induce to produce a new working version of the software, rather than a well designed and documented one. As time goes by, the software increases in size and complexity and its management becomes more and more difficult and expensive. Some reasons for this behavior are:

- Developers maintaining the software normally do not understand it in whole and will even apply changes violating the original architecture and design decisions.
- The documentation is outdated or partially missing.
- The original developers, or at least some of them, with domain and system knowledge, are no longer available.

Sommerville [Som00] and Davis [Dav95] have estimated that the cost of software maintenance takes from 50% to 75% of the overall cost of a software system. Such a high importance in the entire software life cycle has led to the so called "Software Evolution Theory".It concerns how a software system evolves over time, with respect to some software properties.

The major work in this field was done by Lehman *et al.* which consists in a set of empirical laws [LPR$^+$97] based on their case studies of several large software systems. These laws state that as systems grow in size, it becomes increasingly difficult to add new code unless explicit steps are taken to reorganize the overall design.

1.1 The Problem

Understanding the evolution of a software system is a complex task, especially when it is composed of a large number of components. Huge amounts of data have to be analyzed since several versions of the software system must be considered. Therefore, we need an approach which extracts useful information only, starting from the whole set of data. This approach must have the capability to provide fine-grained information as well as grouping it into higher levels of granularity.

The study of a system has two main aspects of interests, which are:

The evolution: The aim is to understand the evolution of software systems and to compare it with the existing theory [Tur96, Leh96, LPR$^+$97, LPR98] and examples [GT00].

The structure and the design: The purpose is to deeply understand the design of the system, the roles of the entities and the distribution of the responsibilities.

These two apparently uncorrelated points of view are coupled. The knowledge of one will significantly improve the analysis of the other. As suggested by Gall *et al.* [GJKT97] and Godfrey *et al.* [GT00], it is useful to examine the structure of the subsystems to gain a better understanding of the system evolution. On the other hand, to address the problem of understanding the overall structure of a software, analyzing more versions of the system gives more reliability on the conclusions drawn.

Since a complete and exhaustive study of a software takes a lot of time, especially when the system is large, in most of the cases it cannot be performed. Thus, there is a need of methodologies which are able to lead the analysis according to well defined goals, such as:

1. Understand the system history in order to deduce how it could evolve from now on.

2. Identify the major phases of the evolution of the system in order to evaluate the quality and the maturity of the software.

3. Understand the overall structure and design of the system.

4. Detect hidden dependencies among system entities.

5. Detect the entities which present design erosion. They represent candidates for a following reengineering process.

In this thesis we focus on software systems developed using CVS [CVS] as version control system (the reasons of this choice are explained in Section 2.2.1). A new problem arises for these systems: The information is poor, spread and not easily interpretable. A proper way to describe this issue consists in the **Software Archaeology** metaphor: The purpose of the *software archaeologist* is to understand what was in the minds of other developers using only the artifacts left behind. He is hampered because the artifacts were not created to communicate to the future, because only part of what was originally created has been preserved, and because relics from different eras are intermingled.

1.2 Software Evolution State of the art

There are many approaches to deal with software evolution, such as:

- *Formulating laws based on empirical observations.* The observations are based on interpreting evolution charts which represent some property on the vertical (*e.g.*, the number of modules) and time on the horizontal axis. The major work in this area was carried out by Lehman *et.* al. [LB85, LPR$^+$97, LPR98].
  This approach has recently been applied on a case study (Linux kernel [Lin]) by Godfrey *et. al.*[GT00]. They found that at the system level the growth of the Linux kernel has been super-linear which is a violation of Lehman and Turski's inverse square growth rate hypothesis [LPR98, Tur96].

- *Extracting quality related measurements from the history of a software.* Burd and Munro present a number of metrics to assess the maintainability of code [BM99]. The availability of such metrics has the potential to assess if and how maintenance changes have affected the comprehensibility of the code.
  Eick *et al.* present a number of measurements that index code decay [Eic01]. Code is defined as being decayed if it is more difficult to change than it should be. They found strong evidence that code does decay.
  Grosser *et al.* [GSV02] aim at assessing the stability in object-oriented systems which they define as the ease with which a software system or a component can evolve while preserving its design as much as possible. They use case-based reasoning based on the hypothesis that two software items which show same or similar characteristics will also evolve in a similar way.

- *Detecting entities in the current version using history information.* Demeyer *et al.* use, in the context of reverse engineering, multiple versions of a software to detect where the implementation has changed

[DDN00]. They propose a set of heuristics to detect specific changes, *i.e.*, refactorings, by applying object-oriented metrics to successive versions of a software system.
Gîrba *et al.* [GDL04] propose an approach for identifying key classes for reverse engineering activities based on the assumption that the parts which change are those that need to be understood first. The approach is based on the retrospective empirical observation that the classes which have changed most recently also suffer important changes in the near future.
Another approach to understand evolution is proposed by Gall *et al.*[GJKT97]. It is based on examining the structure of several major and minor releases of a large telecommunication switching system based on information stored in a database of product releases. The goal of this work is to identify modules or subsystems that should be subject to restructuring or reengineering activities.

- *Understanding evolution using visualization.* In [Lan01, Lan03b] Lanza introduces the Evolution Matrix, which combines software visualization and software metrics to visualize the evolution of the classes of a software system in a matrix. The evolution matrix displays multiple versions of a system at class level. Each column of the matrix represents one version of the software while each row represents the different versions of the same class. The evolution matrix allows one to read: The size of the system in a particular version (in terms of number of classes), the added and removed classes, the system growth and shrinking phases.
In [GL04] Gîrba and Lanza present a visualization approach to understand the evolution of class hierarchies. The authors introduce the notion of a history as a first class entity and define measurements which summarize the evolution of an entity or a set of entities. These measurements are then used to define polymetric views[Lan03b].
Jazayeri *et al.* applied another approach based on color [JGR99, Jaz02]. A history of a release is displayed in a color percentage bar which contains different colors. The colors represent different version numbers of certain parts of the release. This allows the observer to examine the amount of changes from one release to the next.

1.3 Our approach

In this thesis we propose a complete approach to tackle all the issues described in Section 1.1. It is complete in the sense that it covers all the phases required to analyze a software project: From the initial collecting of information to the data presentation and interpretation. The proposed approach is composed of two self-contained parts, each of which is aimed at addressing a class of problems concerning software evolution:

The release history database. It is a database containing the version information with bug tracking report data. The release history database contains the history of a system as a well defined structure. As a consequence, it represents the starting point for many types of evolution analyses, which do not necessarily coincide with ours. We implemented a set of perl and shell scripts able to populate the database in a completely automatic way.

The polymetric views. The second part of the approach is based on *polymetric views*, simple software visualizations enriched with software metrics. According to the granularity level we distinguish two clusters of views:

- **Coarse-grained polymetric views**. They are focused on the entire system from two different points of view. The first, which is the evolution, is tackled by analyzing how the system grows and shrinks, with respect to certain software metrics. The second aspect of interest concerns the overall structure of the system, for which we seek to obtain an initial understanding in terms of modules and module dependencies.
- **Fine-grained polymetric views**. They are aimed at understanding the inner structure of modules. Using the fine-grained views we can figure out the roles of the entities and the distribution of the responsibilities in a module. Moreover, we can detect design shortcomings, hidden dependencies and reengineering candidates.

Based on both the fine-grained and the coarse-grained polymetric views, we propose a top-down methodology to perform the entire analysis of a system, suggesting how to combine the views.

We implemented a tool, called *BugCrawler*[1], which supports both the release history database and the polymetric views. All the thesis results were obtained using exclusively this tool.

We validated the entire approach (release history database, fine-grained views, coarse-grained views and top-down methodology) on the Mozilla[Moz] case-study.

[1] BugCrawler is an extension of CodeCrawler [Lan03a].

1.4 Thesis contributions

The contributions of this thesis can be summarized as follows:

- The development of a release history database which combines version information with bug tracking report data.
- The presentation and discussion of several polymetric views targeting the evolution of a software system.
- The presentation and discussion of several coarse-grained polymetric views. They help to understand the overall structure of a system.
- The presentation and discussion of several fine-grained polymetric views. They are aimed at figuring out the inner structure of modules.
- The development of a top-down methodology which leads the analysis of a software system.
- The analysis of a real case-study (Mozilla) to validate our approach.

1.5 Thesis outline

The thesis is structured as follows:

- In Chapter 2 we present the main problems that have to be tackled in the analysis of the evolution of a large software system.
- In Chapter 3 we describe the release history database: Its data sources, its structure, how it can be built and, in the end, some real examples.
- In Chapter 4 we present and discuss the “coarse-grained” polymetric views. They address the problem of the system understanding from both the structural and evolutionary perspectives. We call the presented views “coarse-grained” because they are focused on the whole system.
- In Chapter 5 we describe the “fine-grained” polymetric views. They tackle the problem of understanding the inner structure of a module. We also introduce some guidelines to identify both symptoms of bad design and candidates for reengineering.
- In Chapter 6 we present a complete top-down methodology which leads the entire analysis of a software system. The proposed approach consists in a set of guidelines suggesting how to combine fine-grained and coarse-grained polymetric views. The explanation of the methodology

is done together with the analysis of a case-study (Mozilla). In this way, we have the possibility to evaluate how the approach works in practice.

- In Chapter 7 we present a summary and possible future work.
- In Appendix A we provide formal definitions of the figures used in the thesis.

Chapter 2

Challenges in Software Evolution

2.1 The problems

In this Chapter we present the main problems that have to be tackled in the analysis of the evolution of a large software system. They can be summarized as:

- Retrieve satisfying evolution information, with respect to their quantity, quality and reliability.
- Find a way to use efficiently the data.

2.1.1 Retrieving evolution information

To study the evolution of a software system, we need all the data regarding its development. Taking into account that the quality of the analysis strictly depends on the quality of the information, it must be carefully evaluated. We do so according to the following criteria:

1. **Applicability**. We want our approach to be applicable to any software system.
2. **Quantity**. We want as much information as possible. The greater the amount of data is, the greater the opportunities we have to reason about the system are.
3. **Quality**. We want detailed information covering all the aspects of the evolution of the system.
4. **Reliability**. We want data which is not affected by any kind of noise.

2.1.2 Coping with huge amounts of data

Once we have collected all the available information, what we need is a way to easily and effectively use it. The large amount of data makes it necessary to group the information at higher levels of abstraction, so that it can be easily understandable. It should then be possible to ungroup them to perform a deeper and more focused analysis.

The approach we need must achieve the following goals:

1. **Simplicity**. The data should be presented in a way that it can be directly interpreted.

2. **Scalability**. It should be possible to show data at any granularity level: From the entire system to the single entity.

3. **Effectiveness**. A lot of information should be shown at the same time.

2.2 Our solution

In this Section we present our solution to the problems introduced above. We address the issue of retrieving evolution information using version control and bug tracking systems. They contain large amounts of historical information that can give deep insight into the evolution of a software project.

Two good and widely adopted approaches to tackle the problem of extracting practical information regarding software systems are software metrics and software visualization. We use a combination of these techniques called polymetric view.

2.2.1 Version control and bug tracking systems

The data sources we have chosen are CVS (as version control system) and Bugzilla (as bug tracking system) repositories, described in detail in the following Chapter (see Section 3.1). The choice has been based on the criteria introduced in Section 2.1.1:

1. **Applicability**. It represents the best benefit of the chosen sources. Bugzilla and CVS are the most used systems in the Open Source software development. We focus on Open Source software because, for these systems, the source code is always available.

2. **Quantity**. These data sources contain huge amounts of information.

3. **Quality**. It represents the major shortcoming of the data sources. The information is spread into the repository (the CVS log files, as we see in Section 3.1.1). Moreover, the link between CVS and Bugzilla is

weak and not formally defined. As a consequence, the data cannot be directly used. A preliminary elaboration phase, which is the topic of Chapter 3, is required.

4. **Reliability**. The data contained in both CVS and Bugzilla repositories is not affected by noise.

2.2.2 Software metrics

Software metrics measure certain properties of a software system by mapping them to numbers, according to well-defined measurement rules. The numbers can then be used to describe the software, with respect to the measured properties. According to [LK94], metrics can be divided in *Design metrics* and *Project metrics.* The former assess the size and the complexity of software, while the latter deals with the dynamics of entire projects. Taking into account the information we have, we use design metrics. The selected metrics, which are referred to CVS and Bugzilla, are depicted in Table 2.1.

2.2.3 Software visualization

Software visualization is defined as "*the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*" [SDBP98]. The software visualization field is divided in: *Program Visualization* and *Algorithm visualization.* We use a sub-area the program visualization called *static code visualization*, which visualizes only information statically extracted from the source code. Once again the choice is based on the available data.

2.2.4 Polymetric View

The approach we use in this thesis is a slightly modified version of the polymetric views introduced by M. Lanza in [Lan03b], which combines software visualization and metrics.
A polymetric view uses two-dimensional visualizations to display software. In detail, it uses nodes (figures) to represent entities, while edges are used to display relationships between the entities. This basic visualization technique is enriched by rendering up to 6 metric measurements on a single node, and up to 2 metric measurements on a single edge. Figure 2.1 shows the mapping of the metric measurements on both a node and an edge.

As we can see from Figure 2.1, the metrics can be mapped on:

- **Node size**. The width and the height of a node can be used to render two metric measurements. The bigger these measurements are, the bigger the node is in its dimensions.

Module Metrics

Name	**Description**
fv	Fractal Value (see Section 4.3.2)
nob	Number of Bugs affecting the products it contains
nod	Number of Directories
nop	Number of Products
nor	Number of Revisions

Directory Metrics

Name	**Description**
age	Number of Days since the first product was created
fv	Fractal Value (see Section 4.3.2)
level	Nesting level in the hierarchy
nob	Number of Bugs affecting the products it contains
nop	Number of Products
nor	Number of Revisions

Product Metrics

Name	**Description**
age	Number of Days since it was created
agr	Average Growth Rate. It is equal to the average number of lines added minus the average number of lines removed (with respect to all the commits)
fv	Fractal Value (see Section 4.3.2)
noa	Number of different Authors
nob	Number of Bugs
nocl	Total Number of Lines Changed (with respect to all the commits)
nof	Number of Fellows (products belonging to the same directory)
loc	Number of Lines of code
nor	Number of Revisions

Revision Metrics

Name	**Description**
agr	Growth Rate. It is equal to the number of lines added minus the number of lines removed
nob	Number of Bugs
nocl	Number of Lines Changed. It is equal to the number of lines added plus the number of lines removed

Table 2.1: A list of the metrics used in this thesis.

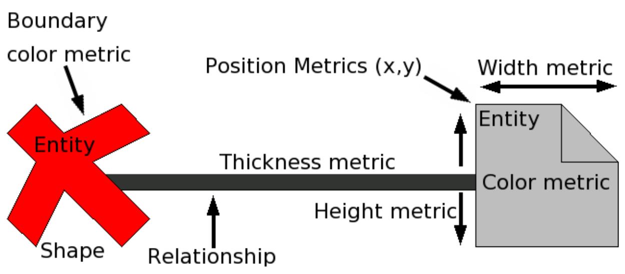


Figure 2.1: The metrics mapping.

- **Node and edge color**. The color interval between white and black can render another metric measurement. The convention is that the higher the metric value is, the darker the node (or edge) is. The color interval between red and blue can also be used. The meaning is that hot colors (red range) correspond to high values of the metric measurement, while cold colors (blue range) correspond to low values.
- **Node boundary color**. The color of the boundary of a node can render a metric measurement as well as the internal node color.
- **Node position**. The X and Y coordinates of the position of the node can also reflect two metrics measurements. Since the presence of an absolute origin within a fixed coordinate system is required, the position metrics are not applicable to all layouts (*e.g.*, a circle layout).
- **Edge thickness**. The width of an edge can render a metric measurement. The bigger the measurement is, the thicker the edge is.

In addition to this mapping, we can assign a semantic to the figure shape, linking different meanings with different shapes. For example, in Figure 2.1 the cross-shaped figure represents a bug, while the other figure represents a file. The use of “ad-hoc” figures allows an immediate understanding of the view.

A polymetric view displays entities, enriched with metrics, according to a well defined layout. The essential layouts are: Tree, circle, scatterplot, and checkerboard. More complex layouts are explained in the remaining part of the thesis, contextually to their use.
In the end, we want to stress that the visualizations are interactive, in the sense that the user can not only see but also interact (zoom, move, remove, inspect, etc.) with the elements in the polymetric views. We believe that by making such interactions possible, the process of understanding the information included in the view can be significantly improved.

Chapter 3

The Release History Database

In order to analyze a software system, from the evolution and structure points of view, we need all the information available regarding it. Unfortunately for the systems developed using CVS as versioning system, such information is poor and not easily readable. However, version information can be enriched with data from bug tracking systems that report about past maintenance activities. Once we have collected all the data, what we need is a way to organize it in a well defined structure.

3.1 Data sources

Our approach is based on two sources of information: The CVS log files (version information) and the Bugzilla bug reports.

3.1.1 CVS Version Control System

CVS [CVS] is the most used versioning system by the Open Source community. It is based on the concepts of *check-in* and *check-out*. The common CVS development process is something like:

1. Get the files (or directories) under development with a CVS check-out command.
2. Modify locally the files.
3. Send the modified files back to the repository with a CVS check-in command.

In such a perspective, it is important to stress the difference between a file and a version of a file. In CVS we have the following entities:

Product: A file, identified by its name and extension.

Revision: A version of a file, corresponding to a CVS check-in, identified by the product (file name and extension) and the revision number (unique for each revision).

Module: A high level entity which can contain directories and files. CVS provides specific commands to manage the modules.

Alias: A symbolic name (while revisions are numbers) assigned to a specific set of revisions.

All the information collected by CVS during its commits is stored in the CVS log files.

```
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
Commit jscodelib (bugid=#238324) onto branch. Needed for XTF javascript
utilities.
=============================================================================
```

Listing 3.1: A CVS log file chunk.

Listing 3.1 depicts a typical log file chunk. For each product we can find two sections in the log: The first concerns the product itself and contains:

- The RCS file, that is the file name with the complete CVS path, as it is stored in the repository.
- The Working file, that is the file name only.
- The product head revision.
- The list of branches, where they exist.
- The number of revisions.

- A text description.
- A list of aliases. For each alias the corresponding revision is also given.

In the second section we find the list of revisions coupled with their check-in notes, that are:

- The unique revision number.
- The check-in time stamp.
- The author of the check-in.
- The revision state. The possible values are “Exp” and “Dead”.
- The lines added and removed with respect to the previous check-in. For the first revision these numbers are both zero.
- The list of branches, where they exist.
- A description inserted by the author. This is the only information asked to the user, while all the other are automatically computed by CVS. In this field we can find bug references as highlighted in listing 3.1 in red.

3.1.2 Bugzilla

Like CVS, Bugzilla [Bugb] is doubtless the most used Bug Tracking System by the Open Source community. The list of companies [Bugc], organizations and projects that use Bugzilla includes IBM, NASA, RedHat, Linux kernel, OpenOffice, Apache.

Apart from its many features, the core of Bugzilla is a database containing well defined bug information. When queried, it is able to return information both in XML and HTML (see Figures 3.1 and 3.2).

For each bug stored, Bugzilla includes the following information [Buga]:

Id: The unique integer bug id identifier.

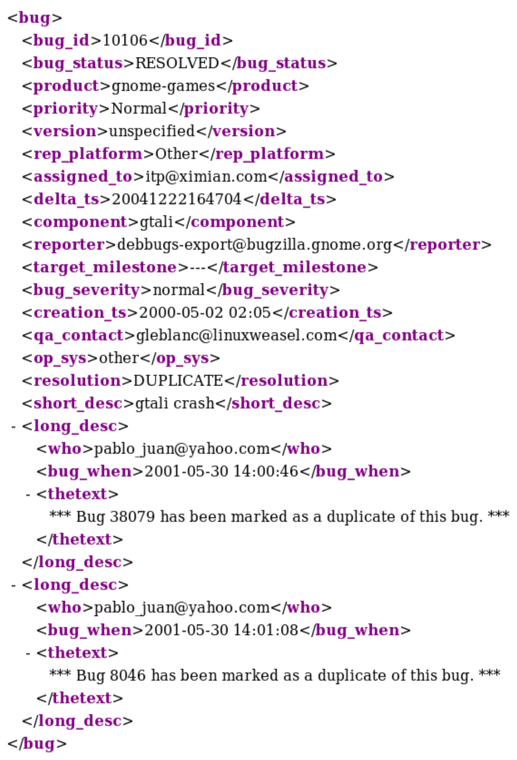
Status: Indicates the general status of a bug. Only certain status transitions are allowed. This field can be: *Unconfirmed, new, assigned, reopened, resolved, verified, closed.*

Product: Identifies the subsystem affected by the bug.

Component: Identifies the software components affected by the bug (in the Product scope).

Creation time stamp: Indicates when the bug have been reported.

Short description: Describes the problem in a couple of lines.



```
<bug>
  <bug_id>10106</bug_id>
  <bug_status>RESOLVED</bug_status>
  <product>gnome-games</product>
  <priority>Normal</priority>
  <version>unspecified</version>
  <rep_platform>Other</rep_platform>
  <assigned_to>itp@ximian.com</assigned_to>
  <delta_ts>20041222164704</delta_ts>
  <component>gtali</component>
  <reporter>debbugs-export@bugzilla.gnome.org</reporter>
  <target_milestone>---</target_milestone>
  <bug_severity>normal</bug_severity>
  <creation_ts>2000-05-02 02:05</creation_ts>
  <qa_contact>gleblanc@linuxweasel.com</qa_contact>
  <op_sys>other</op_sys>
  <resolution>DUPLICATE</resolution>
  <short_desc>gtali crash</short_desc>
- <long_desc>
    <who>pablo_juan@yahoo.com</who>
    <bug_when>2001-05-30 14:00:46</bug_when>
  - <thetext>
      *** Bug 38079 has been marked as a duplicate of this bug. ***
    </thetext>
  </long_desc>
- <long_desc>
    <who>pablo_juan@yahoo.com</who>
    <bug_when>2001-05-30 14:01:08</bug_when>
  - <thetext>
      *** Bug 8046 has been marked as a duplicate of this bug. ***
    </thetext>
  </long_desc>
</bug>
```

Figure 3.1: A Bugzilla Bug (XML format).

Long description: Is a list of comments about the bug. Each comment includes the author, the time stamp and a text field which could be a patch.

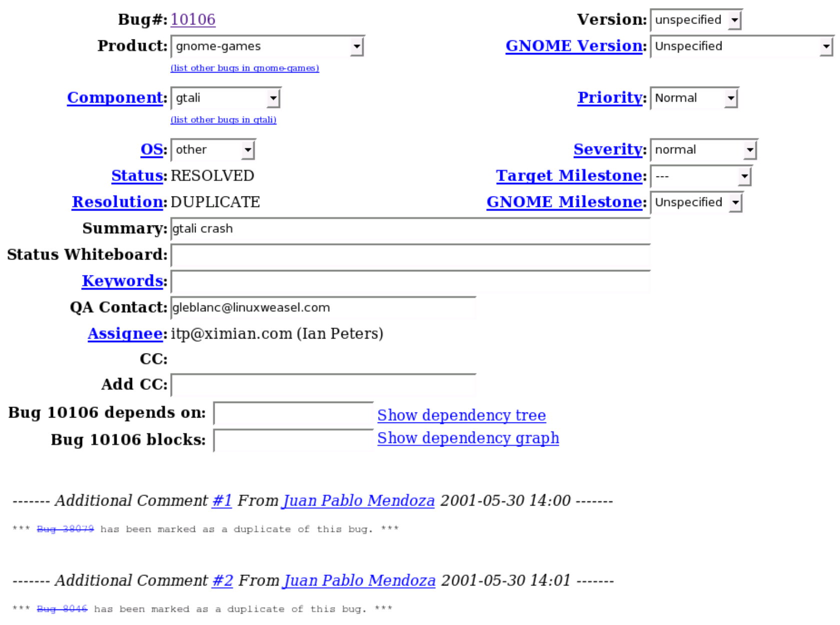
Platform: Is the hardware platform on which the bug was reported. It can be: *All*(happens on all platform; cross-platform bug), *Macintosh, PC, Sun, HP, SGI, DEC, Other.*

Operating system: Is the operating system on which the bug was reported.

Resolution: Indicates what happened to this bug. Allowed values are: *fixed, invalid, wontfix, notyet, remind, duplicate, worksforme.*

Priority: Describes the importance and order in which the bug should be fixed. This field is utilized by the programmers/engineers to prioritize their work. The available priorities range from *P1*(most important) to *P5*(least important).

Severity: Describes the impact of the bug. The allowed values are (in decreasing order): *blocker, critical, major, minor, trivial, enhancement.*



Bug#:	10106	**Version:**	unspecified
Product:	gnome-games (list other bugs in gnome-games)	**GNOME Version:**	Unspecified
Component:	gtali (list other bugs in gtali)	**Priority:**	Normal
OS:	other	**Severity:**	normal
Status:	RESOLVED	**Target Milestone:**	---
Resolution:	DUPLICATE	**GNOME Milestone:**	Unspecified
Summary:	gtali crash		
Status Whiteboard:			
Keywords:			
QA Contact:	gleblanc@linuxweasel.com		
Assignee:	itp@ximian.com (Ian Peters)		
CC:			
Add CC:			
Bug 10106 depends on:		Show dependency tree	
Bug 10106 blocks:		Show dependency graph	

------- *Additional Comment #1 From Juan Pablo Mendoza 2001-05-30 14:00* -------

```
*** ~~Bug 38079~~ has been marked as a duplicate of this bug. ***
```

------- *Additional Comment #2 From Juan Pablo Mendoza 2001-05-30 14:01* -------

```
*** ~~Bug 8046~~ has been marked as a duplicate of this bug. ***
```

Figure 3.2: A Bugzilla Bug (HTML format).

Assigned to: Is the person in charge of resolving the bug.

Reporter: Is the person who discovered the bug.

While the bug structure is always as described above, each software system uses its names to identify the bug fields. For example, the *Status* is identified with a <bug_status> XML tag in the Mozilla system, while it is called <status> in the RedHat [Red] software. As another example the *Product* is coupled with the <category> XML tag in the Linux kernel [Lin], and with the <product> XML tag in the Gnome [Gno] software. Since we want a populating process applicable to all systems using CVS and Bugzilla, we have to take this problem into account (see bug field names problem in Section 3.3.1).

3.2 Database structure

On the basis of the available information, as described in the previous Section, we have designed a relational database with the ER-structure depicted in Figure 3.3. The database is implemented with the Open Source MySql database management system [MyS].

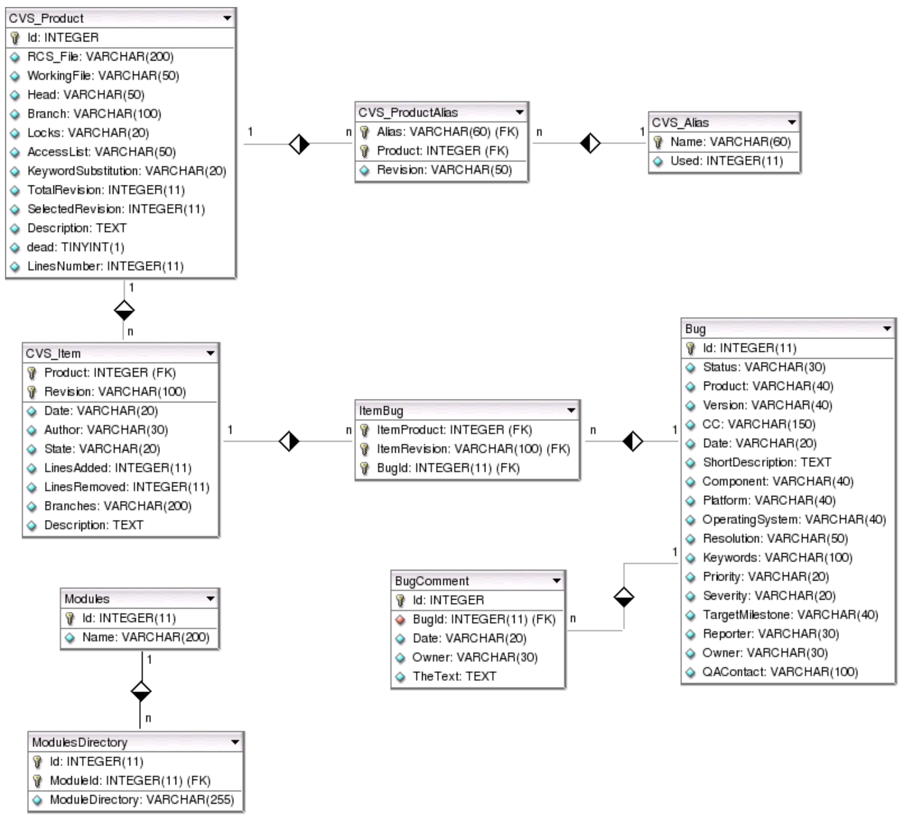


Figure 3.3: The Rhdb Entity-Relationship model.

As we can see from the diagram, the Release History Database is composed by the following tables:

Bug: Stores all the data related to the system bugs as retrieved from Bugzilla.

BugComment: Contains an entry for each long description of a bug.

CVS_Alias: Contains an entry for each Symbolic Name of the software system.

CVS_Item: Contains all the system revisions and their related information as retrieved from the CVS log files. The references with the related CVS products are kept in the **Product** field.

CVS_Product: Is composed of the data regarding the products of the system. The fields **LinesNumber** and **Dead** are not obtained directly from CVS log files. We see in Section 3.3 how they are computed.

CVS_ProductAlias: Stores the one-to-many relation between Symbolic Names and revisions.

ItemBug: Keeps the many-to-many relation between revisions and bugs.

Modules: Contains the list of system modules. Since the module information is not included in CVS log files this data has to be retrieved as described in Section 3.3.

ModulesDirectory: Stores, for each module, all the directories composing it.

3.3 Database population process

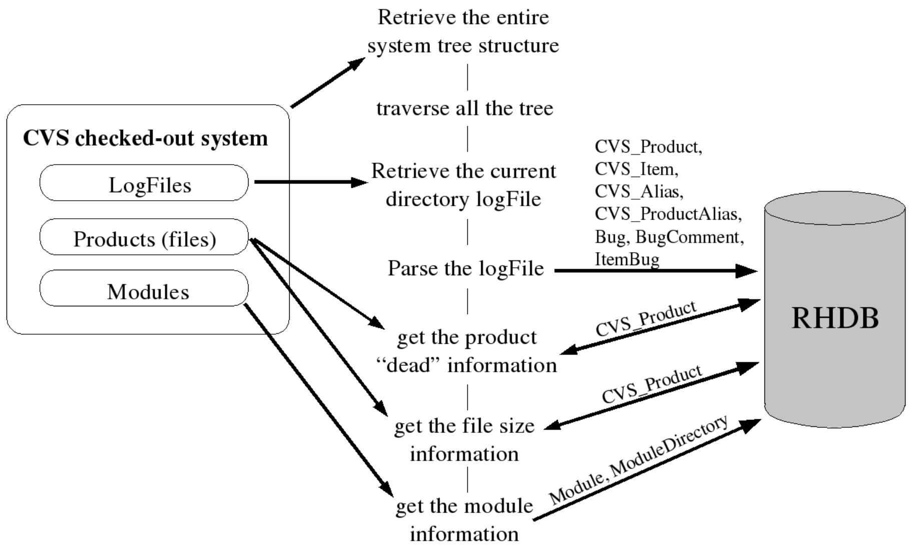

Retrieve the entire system tree structure
traverse all the tree
CVS checked-out system
LogFiles
Products (files)
Modules
Retrieve the current directory logFile
Parse the logFile
get the product "dead" information
get the file size information
get the module information
CVS_Product, CVS_Item, CVS_Alias, CVS_ProductAlias, Bug, BugComment, ItemBug
CVS_Product
CVS_Product
Module, ModuleDirectory
RHDB

Figure 3.4: A high level schematization of the populating process.

To populate our Release History Database, we have developed a set of perl and shell scripts. Thus, besides the prerequisites that the system has been developed using CVS and Bugzilla, we have to add the need of a machine with Mysql and perl interpreter installed. The populating process outline is shown in Figure 3.4.

First of all, we have to perform the CVS check-out of the entire system. Depending on the system size and the bandwidth (between our machine and the server containing the CVS repository), the complete CVS check-out can take from minutes to hours. Once we have downloaded the complete repository, we are able to populate the database. This is performed using a shell script responsible for:

1. Traversing and storing the software directory tree. This step is not strictly required, since the subsequent parsing could be done "on the fly". However, knowing the tree structure allows to determine how much work has still to be done, at any instant of time. This feedback is useful in a process which takes a lot of hours.

2. Traversing again the entire software tree. During this operation a CVS log file is retrieved and locally stored for each directory (with the `CVS log -l` command). Then, a perl script is executed with the log file and a configuration file as parameters. Finally the log file is removed. Since the log parsing represents the core of the populating process, we see it in detail in Section 3.3.1.

3. Once step 2 is completed, the database is almost fully populated. Only little information is lacking. One of them is the *Product state*, which should not be confused with the *Revision state* introduced in Section 3.1.1. We consider a product *dead* (state=dead) when it is moved to the *Attic* subdirectory (with respect to the directory in which the product is stored). On the contrary, a revision is considered *dead* when it is marked as so by its author, during the check-in. A dead revision does not imply that its corresponding product is dead too. The only deduction is that this product has a dead branch; only if the product does not have other living branches it is dead. A dead product has a dead revision for each branch.

   To add the product state information, which corresponds to the *Dead* field in the CVS_Product table, we use again a perl script. It queries the database for the products having an "Attic" string in their *RCS_File* field. Then it modifies these fields by removing "Attic"(from the directory tree point of view, the files are moved to their original location) and finally it sets the *Dead* field to true. For all the other products the *Dead* flag is set to false.

4. The next phase consists in filling the *LinesNumber* field belonging to the CVS_Product table. To do so, a perl script selects the *RCS_File* for each living product. Then, using this information, the corresponding file in the CVS checked-out system is located and its size is calculated. The computed value is stored in the database. For the dead products the *LinesNumber* field is set to *NULL*, because the corresponding file does not exist in the file system.

5. To conclude the populating process, we need to store the Modules structure, in terms of directories included. This data is neither present in the checked-out system, nor in the log files. We can obtain it performing a `cvs co -c` command. It returns the list of modules and, for each of them, its list of directories. Once retrieved this information,

we have to include it into the database, filling the Modules and ModulesDirectory tables. The parsing and the storing is performed by a perl script.

3.3.1 The log parsing

In this Section we describe how the data is taken from the log files and how it is stored in the database. The CVS repository is not the unique source of information, but it is coupled with the Bugzilla repository. Before starting, we summarize the problems that we have to take into account:

Bug field names problem: Different systems use different names to call the same bug field. This problem can be fully overcome using a bug configuration file for each software system. This file includes the bug field names and the Bugzilla repository url used for downloading bug data.

Bug references problem: The bug link, stored in the CVS revision description field, is not formally defined. Since it is an intrinsic shortcoming of CVS, we are not able to entirely avoid it, we can only try to restrict it. In other words, there is no way to be sure of automatically catching all the bug references; the only manner would be to manually read all the description fields. In order to find bug links, we use string matching[1].

Execution time problem: The script execution can take dozen of hours (we see why in Section 3.3.2), and thus it cannot be supervised by a user all the time. As a result, the script should be entirely automatic. Moreover, it should manage errors like corrupted files without interrupting itself, writing the errors on its own log file.

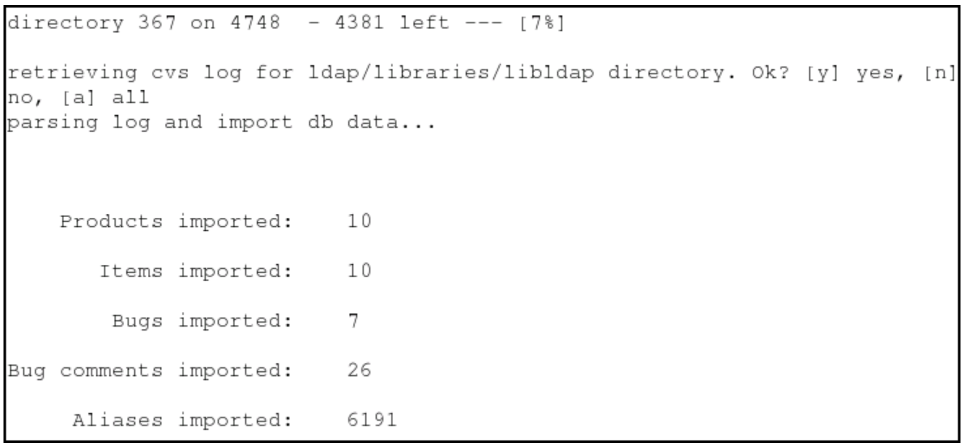
The log parsing is performed by a perl script, which is launched by the shell script, passing it the log file just retrieved and a configuration file (see Bug field names problem). First of all, the script parses the configuration file, importing the parameters as local variables. Then it starts the real parsing, product by product. For each product, it reads all the revisions and stores them into the database. Whenever the script finds a description tag, besides adding it to the database, it looks for bug references (as explained in Bug references problem). If a Bug Id is found, then the program downloads the bug report (in XML format) performing a `wget` command. In case the result

[1] In detail we look for:

- `bug*d{3,}`
- `id*#*d{3,}`
- `b*id*d{3,}`

where `*` means any character and `d{3,}` means at least three digit characters.

of the `wget` is corrupted, the script goes ahead, writing the output coupled with the Bug Id on its own log file (Execution time problem). This allows us to perform a further analysis when the script terminates its work. Finally, the downloaded bug report is parsed and, if it does not exist in the database, it fills a row in the Bug table and a set of rows in the BugComment table. During its execution the program gives some feedback to the user, writing it on the screen. Besides the percentage of work (in terms of number of directories), the number of each entity stored up till now in the database is given. Figure 3.5 shows a screen-shot of the scripts interface.



```
directory 367 on 4748  - 4381 left --- [7%]

retrieving cvs log for ldap/libraries/libldap directory. Ok? [y] yes, [n]
no, [a] all
parsing log and import db data...



    Products imported:    10

       Items imported:    10

        Bugs imported:    7

Bug comments imported:    26

     Aliases imported:    6191
```

Figure 3.5: A scripts execution screen-shot.

3.3.2 Execution time considerations

It could be surprising that the populating process execution time can reach more than 20 hours. The parsing of log files is fast. On the contrary, the `wget` command is relatively slow, because it has to wait for the reply from the Bugzilla repository server. We have calculated, with a sample of 1000 bug reports, that on average a `wget` takes 2.5 seconds. Furthermore, we have another bottleneck, again concerning network time instead of computation time. This is the `cvs log -l` command, which can take, depending on the log size, from 108 to 459 seconds with an average of 195 seconds (on the entire Mozilla sample).

3.3.3 Examples

To conclude this Section we present 4 real examples of our scripts application. We choose big systems because we want to test the reliability of the process. The other criterion followed is related to the availability of the CVS repository: This implies choosing Open Source systems.

The software systems selected for the case-studies are:

- Mozilla[Moz]: A well known web suite including browser, e-mail client, web composer.
- KDE[KDE]: A powerful graphical desktop environment for Linux and Unix workstations.
- Gnome[Gno]: A Unix and Linux desktop suite and development platform.
- Apache HTTP Server[Apa]: An Open Source HTTP Server for modern operating systems including Unix and Windows NT.

The obtained results[2] are summarized in Table 3.1. The data confirms that the bottleneck of the populating process is the `wget` command. The scripts execution time is maximum for the system having the highest number of bugs (Mozilla), while it is shorter for the system having the largest size (KDE).

System	**Size**	**Scripts ex time**	**Product**	**Rev**	**Bug**	**ItemBug**
Mozilla	494MB	22h	85792	755274	26635	166601
KDE	2.7GB	16h	334705	4159499	5421	33166
Gnome	466MB	6.5h	27448	366442	5892	34221
Apache	30MB	15m	2049	37292	96	240

Table 3.1: Release History Database examples.

3.4 Related work

To our knowledge, version control systems, and in particular CVS, were the subject of many studies.
In [GJKT97] Gall *et al.* propose an approach based on examining the structure of several major and minor releases of a large telecommunication switching system based on information stored in a database of product releases. The historical evolution of the structure is tracked and the adaption made are related to the structure of the system. The goal of this work is to identify modules or subsystems that should be subject to restructuring or reengineering activities.
Ball *et al.* [BAHS97] focus on the visualization of statistical data derived from the version control system.

[2] The scripts were executed on a Pentium4 machine with 1GB of RAM using Suse9.1[Sus] as operating system.

In [GJK03] Gall, Jazayeri and Krajewski use the data stored in CVS to detect logical coupling of modules across the evolution of a software system. They propose a Relation Analysis (RA) in which classes are compared based on dates and authors of changes. With this information, parts of the system that were changed together can be discovered.

Little effort was spent in coupling version control and bug report data. In [FPG03b] Gall *et al.* describe how to build a Release History Database starting from CVS and Bugzilla data. The populating process part of our work is inspired on theirs and, since they have taken Mozilla as a case study, we are able to compare the results. Before doing that, we have to briefly outline the main differences of the two approaches:

- The database ER structure is slightly different. Apart from the tables, initially empty, they use for the following evolution analysis, the *author* entity is represented by a table in their Rhdb, while it is a field in ours. Moreover, since they do not consider Module information, the Modules and ModulesDirectory tables do not appear in their Database.

- The populating process proposed by Gall *et al.* provides an heuristic method to find *merge points* across branches (see [FPG03b] at Section 4 for detail). Our approach does not consider this topic.

The numeric results obtained with the two approaches are shown in Table 3.2 (with respect to the Mozilla system). The numbers depicted are the Database table sizes, in number of rows. In order to make the comparison significant, we perform the check-out at the same date as Gall *et al.* did in [FPG03b], namely December 14th, 2002.

	Product	**Revision**	**Bug**	**ItemBug**
Our approach	38432	361904	22409	122792
Gall approach	36662	433833	28456	158491

Table 3.2: A comparison between our and Gall approaches.

Table 3.2 shows that the approach proposed by Gall *et al.* is better from the revision and bug points of view, while our technique is better from the product point of view.

3.5 Conclusion

In this Chapter we have described the Release History Database: Its data sources, its structure, how it can be built and, in the end, some real examples. We believe that the Rhdb could be an excellent starting point for software evolution analysis. The information it contains is rich of semantics from an evolutionary perspective.

3.5.1 Benefits

From our point of view, a benefit of the Rhdb consists in its capability of aggregating two data sources with a weak relationship (the informal link in the CVS description field). This is important because bug reports tell us a lot of the history of a system. They give information about the quality of the design and the code; they help uncovering logical coupling at any level of abstraction.

Another advantage of our approach comes from the technological choices. Having a MySql database allows to:

- Apply data mining techniques.
- Import the data in many programming languages.

The last benefit is the fact that the populating process is completely automatic. The user just has to download the CVS repository, to fill the bug report configuration file and to provide the Bugzilla repository url. Assuming large availability of disk space and CPU time, a considerable number of Rhdbs can be built and analyzed.

3.5.2 Limits

Our approach is limited in the following way:

- *Coupling with our evolution technique.* When we designed the Rhdb structure and construction method, we had in mind how we intended to use it. In other words, the Database was thought ad-hoc for our application. However, we believe that this instrument could be useful for different evolutionary approaches.
- *No merge points detection.* An algorithm to find merge points (like the one used in [FPG03b]) is not provided.
- *Last revision only.* The last revisions of files only are available. In fact, only these revisions are checked-out, while all the other information is retrieved from the CVS log files. Having all the revisions available at the same time means knowing the detailed source code history of every file. This could allow a better understanding of the evolution of the system.

Chapter 4

Software Archaeology in the Large

In this Chapter we address the problem of analyzing a large software system. We are interested in understanding both its evolution and its overall structure. These two apparently uncorrelated points of view are coupled. The knowledge of one will significantly improve the analysis of the other. As suggested by Gall *et al.* [GJKT97] and Godfrey *et al.* [GT00], it is useful to examine the structure of the subsystems to gain a better understanding of the system evolution. On the other hand, from the system history we can retrieve information which could guide the inspection of the system structure.

With the term "in the Large" we mean:

- From the evolutionary point of view, we consider the entire system.
- From the structural point of view, we focus on the highest level entities (the modules). We want to find out which are the most important modules and which relationships hold among them. We also want to understand how the responsibilities and the bugs are distributed among the modules.

Our approach to analyze a system consists in applying a set of highly scalable and interactive polymetric views. Each of them highlights a particular aspect of the entities involved. The examination of the system is therefore performed through the inspection of the views.

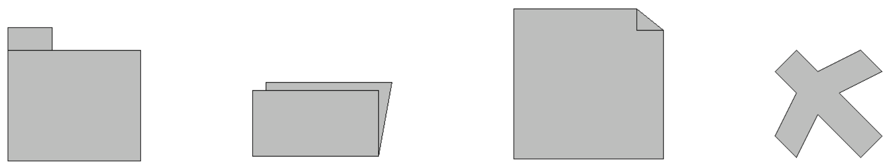
All the views are presented using the same template.

4.1 Preliminaries

Before explaining the views designed for the coarse-grained analysis, we need to introduce the set of figures composing them. While some figures are so simple that they are self-explanatory, the others require a semantic description. Therefore, we will go for the figures presentation, dividing them in **simple figures** and **complex figures**.

4.2 Simple Figures

The simplicity of the following figures does not strictly require an explanation. However, for completeness and symmetry with the other Sections, we provide it.



(a) Module Figure. (b) Directory Figure. (c) Product Figure. (d) Bug Figure.

Figure 4.1: Simple Figures.

Name	**Module**	**Directory**	**Product**	**Bug**
Possible entities	Module	Directory	Product	Bug, Dead revision
Possible metrics mapping	Width, Height, Color, Boundary color			
Example	Fig. 4.1(a)	Fig. 4.1(b)	Fig. 4.1(c)	Fig. 4.1(d)
Customization	None			

4.3 Complex Figures

The complex figures we implemented are:

1. Discrete Time Figure (see Section 4.3.1).
2. Fractal Figure (see Section 4.3.2).

4.3.1 Discrete Time Figure

Name	**Discrete Time Figure**
Possible entities	Module, Directory, Product
Possible metrics mapping	-
Example	Figure 4.2

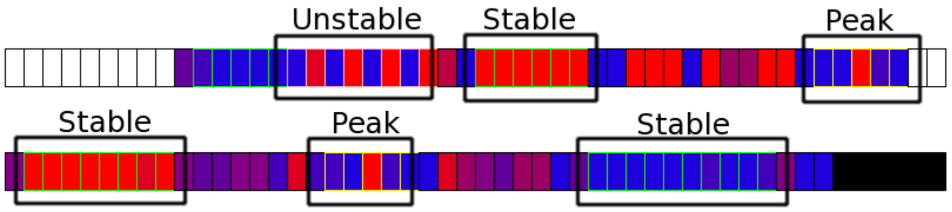


Figure 4.2: Discrete Time Figure with phases.

Notes

The Discrete Time Figure represents the production of revisions (or bugs) over time. Each rectangle composing the figure is related to an interval of time, while its color corresponds to the number of revisions (or bugs) checked-in during this period of time. The color is chosen with respect to a set of thresholds, where the number of possible colors is equal to the number of thresholds plus one. For example, if the set of thresholds is $T = \{t_1, t_2, t_3\}$ then the set of colors is $C = \{r_1, r_2, b_1, b_2\}$ where the choice of colors is made upon:

$$\begin{array}{lll} r_1 & \text{if} & nor(\text{or } nob) > t_3 \\ r_2 & \text{if} & t_3 \leq nor(\text{or } nob) < t_2 \\ b_1 & \text{if} & t_2 \leq nor(\text{or } nob) < t_1 \\ b_2 & \text{if} & nor(\text{or } nob) \leq t_1 \end{array}$$

Independently from the number of thresholds, the first color (r_1) is always a pure red, while the last color ($b_{n/2}$ where $n = |C|$) is a pure blue. All the other colors range from r_1 and $b_{n/2}$ where hot colors mean high production of revisions (or bugs) and cold colors mean low production. As we see from Figure 4.2, in addition to the standard colors there are two special ones: Transparent and black. The former means that the time period corresponding to the rectangle precedes the first revision (or bug) of the entity (Module, Directory or Product) or follows the last. The latter means that the entity is dead in the considered time interval[1].

[1]The entity is dead if the time interval follows the last revision (or bug) and all the

The Discrete Time Figure implements a **phases detection** mechanism. We examine the sequence of rectangles looking for patterns (the so called phases). If a pattern is found the lines of the rectangles composing it are rendered according to the policy described below. In detail, we recognize the following phases:

Stable: It represents a period of time during which the production of revisions (or bugs) is relatively constant (no matter if it is high, medium or low). A sequence of rectangles is recognized as a stable phase if: It is long enough and there are no oscillations (or small oscillations) in the color values. For the stable phase the lines of the rectangles are **green**. An example is shown in Figure 4.2.

Peak: It is composed of three periods of time having the following characteristic:

- During the first and the third periods, which are long, the production of revisions (or bugs) is high (low).

- During the second period, which is short, the production of revisions (or bugs) is low (high).

A sequence of rectangles is detected as a peak phase if: The first and the third periods are sufficiently long, and the difference between the values of *nor* (or *nob*) in the first (or third) and in the second periods is adequately high. For the peak phase the lines of the rectangles are **yellow**, as we can see from Figure 4.2.

Unstable: It is the contrary of the stable phase. We call unstable phase a period of time during which the production of revisions (or bugs) oscillates from low to high values and vice versa. A sequence of rectangles is detected as an unstable phase only if: There are at least 6 periods of time such that the difference between the production of revisions (or bugs) of each pair of adjacent periods is enough high. Moreover, the difference between the values of *nor* (or *nob*) in time periods having a distance equal to two must be sufficiently low. For the unstable phase the lines of the rectangles are **pink** (see Figure 4.2).

The formal definitions of the phases are provided in Section A.1.

Customization

The Discrete Time Figure is highly customizable. The choice of the parameter values is fundamental: The figure efficacy depends on it. There is no optimum choice *a priori*, it depends on the entity (or entities set) under analysis. The parameters are:

branches are marked as dead.

- The number of the thresholds $t_i \in T$. If this number is too high, then the resulting figure is difficult to interpret. On the other hand, the higher the number of thresholds is, the more the information contained in the figure is detailed.

- The values of the $t_i \in T$. These values can be automatically computed or manually set. The first alternative is useful if we need a "general" impression of the entity. The second one could be helpful to underline a particular aspect of the entity.

- The number of rectangles composing the figure. This parameter represents the time granularity of the figure: The higher the value is, the finer the granularity is, *i.e.*, the shorter the time intervals are.

Variations

Discrete Time Combo Figure. It is composed of two Discrete Time Figures. These figures represent the same entity but while the first is related to the production of revisions, the second is referred to the production of bugs (see Figure 4.3). Vertically aligned rectangles in the two Discrete Time Figures correspond to the same period of time.

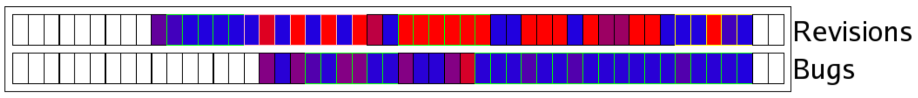
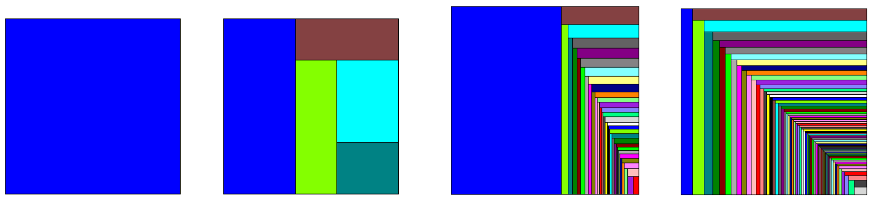


Figure 4.3: Discrete Time Combo Figure.

4.3.2 Fractal Figure

Name	**Fractal Figure**
Possible entities	Module, Directory, Product
Possible metrics mapping	Width, Height
Example	Figure 4.4



(a) One developer

(b) Few developers

(c) Mainly one developer

(d) A lot of developers

Figure 4.4: Fractal Figures.

Notes

The Fractal Figure gives an immediate view of how the entity was developed. We can easily figure out whether the development was done mainly by one author or a lot of people contributed to it. The figure is composed of a set of rectangles having different sizes and colors. Each rectangle, and thus each color, is related to an author who worked on the entity[2]. The area of the rectangle is directly proportional to the percentage of check-ins performed by the author over the whole set of check-ins. The ratio $\frac{\text{rectangle area}}{\text{figure area}}$ is directly proportional to the ratio $\frac{\text{author check-ins}}{\text{total check-ins}}$.

In Figure 4.4 four typical development patterns are depicted: Only one author (4.4(a)), a small number of authors (4.4(b)), many authors but unbalanced (one performed half of the work, all the others performed the second half, 4.4(c)), many balanced authors (4.4(d)). The Fractal Figure gives us a qualitative impression of the development process. To have a quantitative idea, we have designed the so called **Fractal Value**, which is formally defined as:

$$\text{Fractal Value} = 1 - \sum_{a_i \in A} \left(\frac{nc(a_i)}{NC}\right)^2, \quad NC = \sum_{a_i \in A} nc(a_i) \tag{4.1}$$

[2]If the entity is a product then “worked” means performed at least one check-in. If it is a module or directory then “worked” means performed at least one check-in of a product contained in the directory or module.

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.

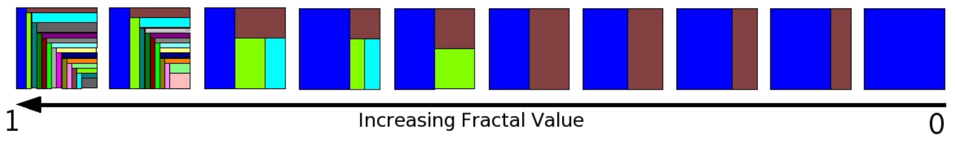


Figure 4.5: The Fractal Value trend.

The fractal value measures how much "fractalized" the figure is, that is how much the work spent on the corresponding entity is distributed among different developers. Looking at equation 4.1, we notice that:

- Since the square equation is sub-linear between 0 and 1, the smaller the quantity $\frac{nc(a_i)}{NC}$ is (that is always lesser or equal to 1), the more it is reduced by the square power. Therefore, the smaller a rectangle is, the lesser 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 (as shown in Figure 4.5).

Customization

None.

Variations

- *Fixed Width Fractal Figure:* The figure width is fixed to a constant value. Therefore, the percentage of commits performed by one author is proportional to the height of the rectangle.
- *Fixed Height Fractal Figure:* The figure height is fixed to a constant value. Therefore, the percentage of commits performed by one author is proportional to the width of the rectangle.

4.4 Understanding the evolution of the system

The first aspect of interest, in studying the evolution of a software system, is how the system under analysis grows and shrinks during its life time. The problem is that there is not a unique definition of growth, since it refers to a dimension of the system. Our approach consists in considering three metrics:

1. The number of revisions (*nor*).
2. The number of lines of code (*loc*).
3. The number of bugs (*nob*).

While the first two were extensively used [Tur96, Leh96, LPR$^+$97, LPR98, GT00], the latter represents a new dimension to inspect. From our point of view, analyzing the system growth means understanding how the *nor*, *loc* and *nob* vary. To do so we have designed two clusters of views. The first, called SYSTEM GROWTH VIEWS, is composed of three elements, each of which shows a particular aspect of the system growth. The second, called SYSTEM PRODUCTION TREND VIEW, includes two elements depicting the *nor* and the *nob* trend during the system life time.

4.4.1 System Growth Views

These polymetric views are aimed at:

- Understanding the system history in order to deduce how it could evolve from now on.
- Comparing the evolutions of different systems developed with different paradigms (OpenSource, Commercial).
- Comparing the evolution of the system with the existing software evolution theory [Tur96, Leh96, LPR$^+$97, LPR98] and example [GT00]. Find out where the system is aligned with the theory and, where it is not, try to understand the reasons for such a difference.
- Identifying the phases of the evolution of the system to evaluate the quality and maturity of the software project.

The SYSTEM GROWTH VIEWS are also applicable at finer granularity levels. They can show how a single module, directory or product grows and shrinks during its life time. The modified versions of the view are called respectively MODULE GROWTH VIEWS, DIRECTORY GROWTH VIEWS and PRODUCT GROWTH VIEWS.

Name	System Revisions Growth View	
Layout	Scatterplot	
Scope	Entire System	
Nodes	**Entity**	Growth Point
	Figure	Fixed Rectangle Figure
Edges	-	
Metrics	**Size{Width ; Height}**	{- ; -}
	Position{x ; y}	{Time(N. of Days) ; *nor* }
	Color	-
Sorting	-	
Appearance	See Figure 4.6	

Main Idea

This simple view shows how a software system grows during its life time.

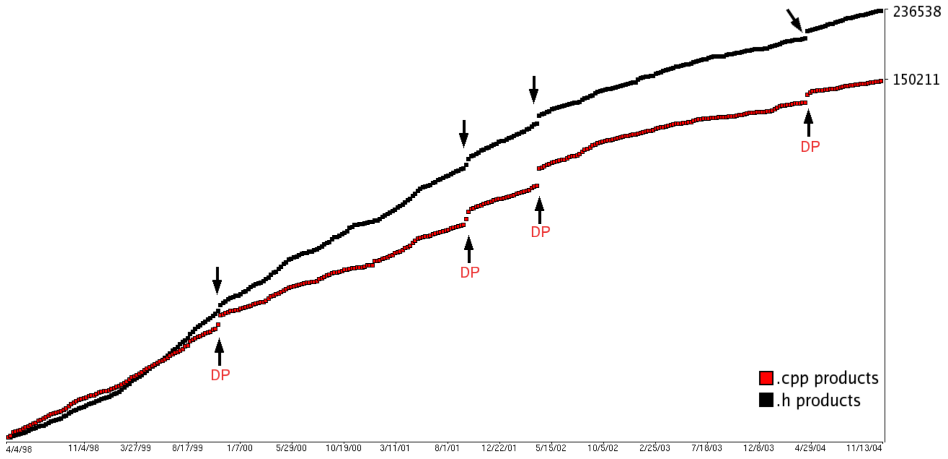


Figure 4.6: The superimposition of two System Revisions Growth View applied to Mozilla. As metric we use the number of days for the horizontal position and the number of revisions for the vertical position.

Notes

The System Revisions Growth View helps discovering out phases in the evolution of a subject system. We can detect them according to the different *nor* growth rates. It means that, assuming *nor* as a function of the number of days ($nor = f_{nor}(nod)$), each phase is identified by its nearly constant $\frac{\mathrm{d}f_{nor}(nod)}{\mathrm{d}nod}$ derivate value.

The entities composing the System Revisions Growth View are Growth Points, which encapsulate the following information:

- A time stamp.
- The difference between the time stamp and the time stamp of the first revision of the system expressed in number of days.
- The number of revisions checked-in until the time stamp.

Key Points

- *Dead Phases.* They consist in flat chunks of the curve, where $\frac{\mathrm{d}f_{nor}(nod)}{\mathrm{d}nod} = 0$. We call them Dead phases because during these intervals of time no work was spent on the system. The same phenomenon with the *nob* or *loc* metrics does not have the same implication. In fact, zero bugs could be the consequence of a bugs fixing phase, while $loc = 0$ could mean that the lines added are equal to those removed.
- *Discontinuity points.* They indicate a production of revisions above average during a time step. A subsequent analysis could explain the causes of such a high value of *nor*.
- *Changes in the curve slope.* They suggest that in these points there were substantial innovations in the system. They act as boundaries between two different phases of the evolution of the system. The possible reasons of such a big change in the system are various, such as:
  - Addition of a new product/project to the entire system. The product is not necessarily new, it could be a porting to another architecture or operating system.
  - Removal of a product/project or a part of it.
  - Some maintainers change.
  - The budget assigned to the system changes (funds are added or removed).
  - Considerable changes in the system design due to a reengineering or a refactoring application.

Customization

The only parameter we can change in the view is the **time step**, that is the time distance between two adjacent Growth Points. Changing the time step implies changing the Growth Points density, because[3]

Growth Points number = (total number of days)/(time step)

[3]Since the time is measured in days, the minimum value allowed for the time step is one day.

However, decreasing the time step too much is useless as the Growth Points will overlap. Increasing it should show discontinuities otherwise invisible. The time step default value is one week.

Variations

- System Lines Growth View. The number of lines of code can be used for the vertical position metric, instead of the number of revisions. The information given by the *loc* metric is potentially more detailed with respect to those retrieved from *nor*. In fact, this metric weighs each check-in with the work performed, expressed in terms of number of lines. The *loc* metric suffers from the effect of the lines removal. As an example we can have a $nor = 1000$ and a $loc = 0$ in the same situation: It means that, over the 1000 check-ins, on average the lines added are equal to those removed.

- System Bugs Growth View. This variation is obtained by using the number of bugs for the vertical position metric, instead of the number of revisions. It shows how bugs are discovered and reported during the system life time. The curve interpretation is far from being immediate. In fact, a high *nob* growth rate could be related to a "strong" testing phase as it could imply a bad development phase. The same can be stated for low values of *nob*. They could be the consequence of an excellent development process or a poor testing phase. Furthermore, we have to take into account that many bugs are not discovered during the testing and some other exist in the context of a specific application only. Then, there are bugs which are generated by fixing others. All these reasons make the curve interpretation difficult.

Example

See Section 6.1.1.

4.4.2 The System Production Trend View

Name	System Production Trend View	
Layout	Scatterplot	
Scope	Entire System	
Nodes	**Entity**	Time Interval Production Node
	Figure	Fixed Rectangle Figure
Edges	Time Interval Production Edge	
Metrics	**Size{Width ; Height}**	{- ; -}
	Position{x ; y}	{Time(Number of Intervals) ; *nor* or *nob* }
	Color	-
Sorting	-	
Appearance	See Figure 4.7	

Main Idea

This view gives an indication of the liveliness of the system. Looking at the view we can figure out the moments of maximum development and how the work is distributed over time.

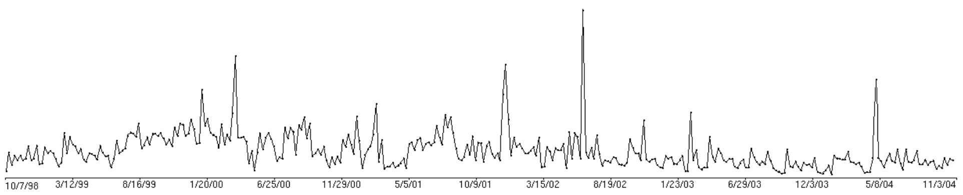


Figure 4.7: The System Production Trend View applied to Mozilla. As metrics we use the number of time intervals for the horizontal position and the number of revisions for the vertical position.

Notes

The System Production Trend View is composed of a sequence of points representing the Time Interval Production Nodes. This type of entity encapsulates two pieces of information: A time period and the number of revisions (or bugs) produced during it. The points are rendered in the view with a vertical position proportional to the number of revisions (or bugs) checked-in during the time interval. The horizontal position reflects the time intervals order, that is how many time periods precede the current.

The edges link two adjacent nodes. The purposes of the view in a software evolution context are various:

1. Assess the system “granularity”. The graph is often characterized by high and frequent oscillations. With a time interval of one week, this means that the production of revisions (or bugs) in following weeks has a weak correlation. Our goal is to figure out which is the time interval that makes the graph oscillations soft (*i.e.* the values of *nor* computed in adjacent nodes correlated). This time interval is the system granularity. This investigation can be done by building the view many times with an increasing time interval length. If the oscillations are frequent and high with any time interval length, then the system has not a granularity. In such a case we postulate that the production of revisions (or bugs) in the system is a self-similar process [HS98] (looks "roughly” the same on any scale).

2. Understand whether a relation between the production of revisions (or bugs) and the time period exists or not. As an example, we could expect to find low values of *nor* during the last weeks of the year. To find such a relation we have to render a view for each year, overlap them and look for recurring patterns.

3. Perform a statistical analysis of the graph in terms of mean and variance. Then we can compare the system with itself in different years and with other systems.

4. Detect phases in the graph (we see how to do it in detail in the key points discussion).

5. Figure out the major phases of the evolution of the system (three or four at most) to understand in which phase it is and how it could evolve from now on.

6. Check whether a relation between the *nor* and the *nob* trends exists or not. For example we could have that a *nor* positive peak is followed by a *nob* one. In such a case the bugs concerning the *nob* peak are related to the revisions checked-in in the *nor* peak with high probability. So we suppose that the *nor* peak is associated to an addition of features in the system. As another example we could have the same situation but, this time, the *nob* peak is negative. We suppose that the *nor* peak is associated to a bug fixing phase.

   A way to find relations, if they exist, consists in superimposing the *nor* and the *nob* graphs in one view.

Key Points

- *Peaks.* They indicate a production of revisions (or bugs) much above or below the average. We need to perform a further inspection to understand the reasons of their existence.

- *Mean and variance.* The computation of these quantities gives a numerical idea of how the *nor* (or *nob*) metric varies over the considered time interval.

- *Major phases.* We can detect them by looking at the general trend in a view with a time interval equal to a week. Then, to validate major phases we need to analyze views with longer time intervals.

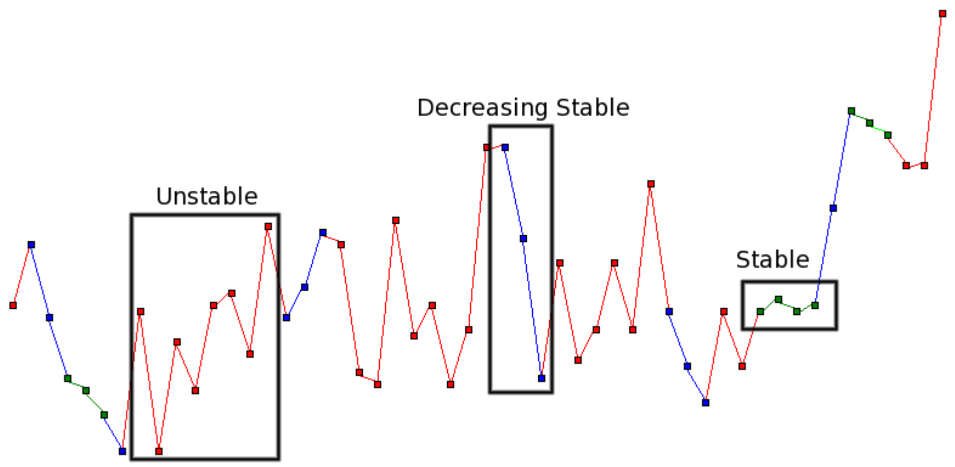


Figure 4.8: The System Production Trend View applied to Mozilla. The time period is from Jan 1, 2000 to Jan 1, 2001. The detected phases are highlighted.

- *Phases.* They are formally defined patterns involving adjacent points in the graph. The formal definition allows an automatic detection of the phases. To distinguish different phases, we assign to each of them a specific color, according to the policy shown in Figure 4.8. The phases point out a period of time during which the production of revisions (or bugs) follows a certain type of pattern. In detail we have:

  Stable: Corresponds to a period of at least three time intervals during which the value of *nor* (or *nob*) has small variations. In other words, we have a stable phase when the time intervals are sufficiently correlated from the *nor* (or *nob*) point of view.

Increasing/Decreasing Stable: Is related to a time period composed of at least three basic time intervals during which the *nor* (or *nob*) growth rate[4] has small variations.

Unstable: Is everything remaining after the detection of the previous phases. An unstable phase is a time period during which the value of *nor* (or *nob*) oscillates without a particular pattern.

Figure 4.8 depicts examples of the phases defined above, for which the formal definitions are provided in Section A.2. Note that, by definition, the set of phases covers the entire graph.

Customization

The customization possibility is fundamental for this view, since its efficacy depends on it. The view parameters are:

- The time interval length expressed in seconds. The default value is 604800 which corresponds to one week.
- The beginning and the end dates.
- The values of the thresholds used to detect the phases (see Section A.2).

Variations

For this view we can use the *nor* metric, as well as the *nob* one, for the vertical position metric of the Time Interval Production Nodes. Using the *nob* metric we expect less and lower peaks than with the *nor* metric. The *nob* graph is expected to oscillate less than the *nor* one.

It would be interesting to analyze the superimposition of the two views, that represents the relation between the revisions and bugs trends. In particular, we should look for cross-correlations like:

- A revisions peak followed by a positive bugs spike. It could be related to an "addition of features" phase.
- A revisions peak followed by either a decreasing stable bugs phase or a negative bugs spike. It could imply a "bug fixing" phase.

Example

See Section 6.1.4.

[4] The *nor* (or *nob*) growth rate is positive for the Increasing Stable Phases and negative for the Decreasing Stable Phases.

4.5 Understanding the design and the overall structure of the system

Understanding the structure of huge software systems is a complex task. The large number of components makes it difficult to figure out where the analysis should start.

We have to tackle the problem incrementally, starting from the highest level entities. We present a cluster of polymetric views, aimed at obtaining a first overview of a subject system, which are:

- CVS Module View (see Section 4.5.1).
- Discrete Time Combo Module View (see Section 4.5.2).
- Module Bugs Correlation View (see Section 4.5.3).
- Product TimeLine View (see Section 4.5.4).

4.5.1 The CVS Module View

Name	CVS Modules View	
Layout	Checkerboard	
Scope	Entire System	
Nodes	**Entity**	Module
	Figure	Module Figure
Edges	-	
Metrics	**Size{Width ; Height}**	{*nop* ; *nop*}
	Position{x ; y}	{- ; - }
	Color	Number of Bugs
Sorting	Number of Products	
Appearance	See Figure 4.9	

Main Idea

The CVS Module View gives a first insight of a software system in terms of modules.

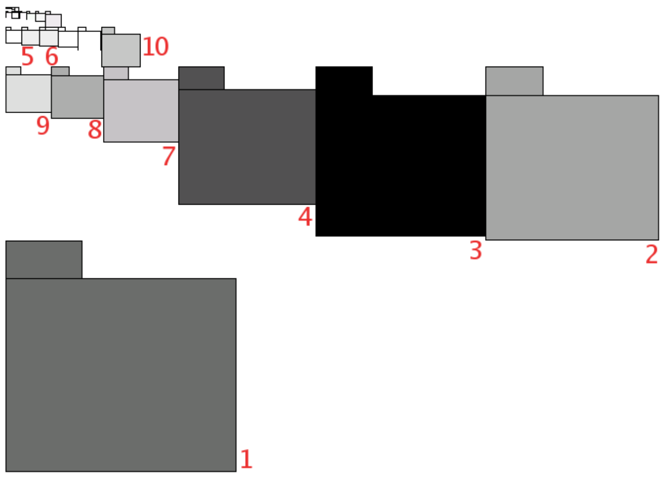


Figure 4.9: The CVS Modules View applied to Mozilla. As metrics we use the number of products for both the width and the height and the number of bugs for the color.

Notes

This view shows all the system modules enriched with the *nop* and *nob* metrics. It gives an idea of which are the modules that play a key role in the system, and which are the minor and marginal ones. The view is also helpful to find small modules affected by a large amount of bugs.

Key Points

- Big and dark figures represent the principal modules of the system. They are good points to start a fine-grained analysis.
- Small and light figures represent marginal modules. In most cases we don't need to further inspect them.
- Small and dark figures are related to modules which contain few products and many bugs. Since these modules are symptoms of bad design, we need to analyze them.

Example

Figure 4.9 shows the CVS Module View applied to the Mozilla system. Since a complete analysis would take time and it is already present in Chapter 6 (see Section 6.2.1), we outline only few observations. Looking at the picture we identify four principal modules, marked as 1,2,3 and 4. The modules 7,8,9 and 10 are medium from both the number of products and number of bugs points of view. There are also two small modules (5,6) affecting by a relatively high number of bugs. We consider the rest of the entities marginal because they are small and light.

Customization

None.

Variations

Instead of using as size the number of products, we can use the number of revisions. In this way, the more work was spent on a module, the bigger the corresponding figure is. This altered view, called CVS Revisions Module View, should be used together with the default one in order to either confirm or invalidate the conclusions drawn.

4.5.2 The Discrete Time Combo Module View

Name	Discrete Time Combo Module View	
Layout	Checkerboard	
Scope	Entire System	
Nodes	**Entity**	Module
	Figure	Discrete Time Figure
Edges	-	
Metrics	**Size{Width ; Height}**	{- ; -}
	Position{x ; y}	{- ; -}
	Color	-
Sorting	Quantity of red	
Appearance	See Figure 4.10	

Main Idea

This view gives a first impression of the modules liveness. We can see when modules were born and, in some cases, when they died. We can also understand which parts of the system changed more frequently.

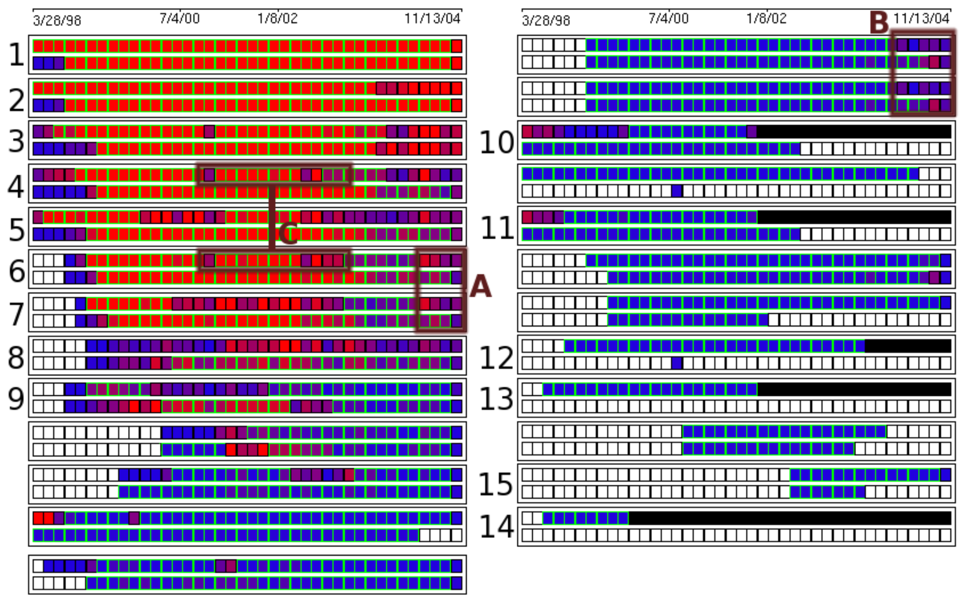


Figure 4.10: The Discrete Time Combo Module View applied to Mozilla.

Notes

The Discrete Time Combo Module View shows for all the modules how their production of revisions and bugs are distributed over their life time. The figures semantic is explained in Section 4.3.1. The thresholds used to render the view are the same for all the figures and, if automatically computed, they are based on the entire revisions and bugs sets (concerning all the modules).

Key Points

- The figures which contain a lot of red rectangles represent the modules which play a key role in the system. We call them *Hot Modules*. Since the system development was concentrated on these modules, a fine-grained analysis should start from them.
- The figures which contain only red rectangles at the end represent the modules which are more likely to change in the future, as assumed in [GDL04][5]. We call them *LTC Modules* (*L*ikely *T*o *C*hange).
- The figures which contain black rectangles represent dead modules. From the first black rectangle position we can figure out the death date. It would be interesting to perform a further analysis to understand the causes of death (for example the module responsibilities were moved to another module).
- Similar color patterns appearing on different figures could implicate a logical coupling [GHJ98] among the modules. We need to confirm or invalidate this hypothesis, performing a following inspection.

Example

Figure 4.10 shows the Discrete Time Combo Module View applied to the Mozilla system[6]. Looking at the view, we notice the following system properties:

[5] The assumption is that products which have changed most recently also suffer important changes in the near future.

[6] To have a numerical idea of the figures meaning, we report both the threshold values (automatically computed) and the time interval:

- Revision Thresholds = $\{100, 200, 300, 400, 500, 600, 700\}$.
- Bug Thresholds = $\{37, 74, 111, 148, 185\}$.
- Time Interval = 60 days.

- There are three hot modules (1,2,3) on which the productions of both revisions and bugs were concentrated during the entire system life time.
- There are five figures (4,5,6,7,8) having an intense middle phase and a decreasing ending phase. They represent important modules which tend to be relatively stable (with respect to the first three modules) in the last year of the system life.
- There is a figure (9) for which the ratio $\frac{\text{red bug rectangles}}{\text{red revision rectangles}}$ is maximum. It represents a module that is likely to include one, or more, critical products.
- There are five dead modules (10,11,12,13,14). It is interesting to notice that three of them (10,11,13) died at the same moment t_1. This fact suggests that there was a big change in the system, since even three modules were involved. We also notice that module 15 started to exist at t_2, where $t_2 = t_1 + 3 \times \text{time interval} = t_1 + 180$ days. It could be possible that the responsibilities of 10,11 and 13, or part of them, were moved to the "new" module 15.
- There are stable phases only and there is no LTC Modules. The reason is related to the values of the thresholds. They are set according to the system granularity, and while such values are suitable to compare different modules, they are inadequate to find something inside a module. The best way to highlight phases and LTC Modules is to create ad-hoc views, composed of one module, or one class of modules[7], only.
- There are one weak (involving only revisions) and two strong (involving both revisions and bugs) color patterns (C,A,B). Once a color pattern is detected, before going any further with the internal analysis of the modules, it would be better to confirm the presence of the pattern using an ad-hoc view (as in the previous point).

Customization

See the Discrete Time Combo Figure customization in Section 4.3.1.

Variations

We could be interested in the production of revisions or bugs only. In this case, we make use of the simple Discrete Time Figure, instead of the combo one, applied to *nor* or *nob* metrics. These views are called Discrete Time Module View and Discrete Time Bug Module View respectively.

[7]A set of modules is considered a class if their *nor* and *nob* values are comparable.

4.5.3 The Module Bugs Correlation View

Name	Module Bugs Correlation View	
Layout	Circle	
Scope	Entire System	
Nodes	**Entity**	Module
	Figure	Module Figure
Edges	Module to Module	
Metrics	***Module Node***	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{- ; -}
	Color	Number of Revisions
	Module to Module Edge	
	Width	Number of Shared Bugs
	Color	Number of Shared Bugs
Sorting	-	
Appearance	See Figure 4.11	

Main Idea

The purpose of this view is to detect dependencies between different modules. A bug denotes a problem in the software. If it is shared by two different entities (modules in this case), then they are somehow correlated. The higher the number of shared bugs is, the stronger the correlation between entities is.

Notes

The Module Bugs Correlation View allows to easily uncover the module dependencies by displaying edges with thickness and color proportional to the number of shared bugs. The more the edge is dark and thick, the stronger the relation between the involved modules is.

The number of revisions metric, applied to the color of the module figures, is helpful to figure out whether the relation holds between similar entities (from the development effort point of view) or not.

Key Points

- Dark and thick edges represent strong relations. Since coupling between entities could be a symptom of bad design, we are interesting in inspecting the modules linked to the edges.
- Relatively dark and thick edges which link figures having different colors, represent relations linking different modules (from the number

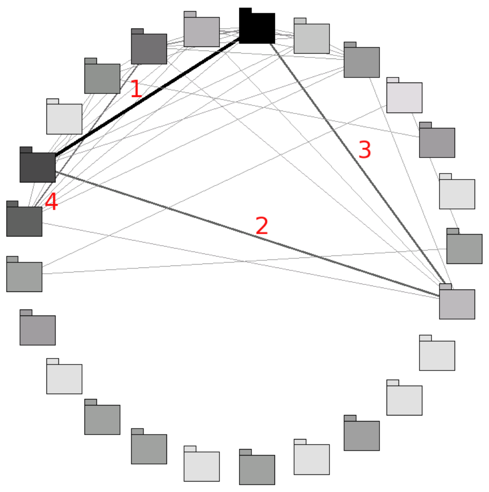


Figure 4.11: The Module Bugs Correlation View applied to the Mozilla system. As metrics we use the number of revisions for the figure color and the number of shared bugs for both the edge color and width.

of revisions point of view). We need to further analyze the modules together, in order to understand their role.

Example

In Figure 4.11 the view applied to the Mozilla system is shown. To make the view easy to read, only the edges with a number of shared bugs greater than 150 are displayed.
Before going any further, it is important to stress that the example should be carefully interpreted. In fact, there are modules for which the intersection is not empty. This implies the possible presence of "false" correlations, in which the bugs are not really shared. These bugs affect the same files that belong to different modules.

Looking at the view we find four principal relationships. However, all of them could be false correlations, since all the pairs of modules involved have an not empty intersection. To validate this hypothesis, we have to remove the directories in common (from one module only), and then recompute the number of shared bugs.
As a result, we find that all the correlations are reduced in strength[8]. Furthermore, the dependency marked as 1, that is at first impression the strongest, is a false correlation.

[8]The numbers of shared bugs decrease as follows: from 2526 to 642 for correlations 2 and 3; from 2250 to 117 for correlation 4; from 6157 to 0 for correlation 1.

Customization

None.

Variations

The number of bugs can be used for the module color metric, instead of the number of revisions. The modified version of the view is useful to study the relation between absolute number of bugs and shared bugs. Are the critical modules those which share the most bugs? Given two modules a and b, where a is critical and b is not, does a strong bugs sharing relation imply that b is also critical?

Another variation consists in altering the view scope: From the system to the module context. Nothing changes but the module figures are substituted by directory figures. This variation of the view is presented in the following Chapter (Section 5.2.4).

4.5.4 The Product TimeLine View

The detail concerning the Product TimeLine View are described in the following Chapter (see Section 5.3.2), because it is a fine-grained view. However, since it scales-up well, this view is also helpful in a coarse-grained context. In Figure 4.12 we can see the view applied to entire Mozilla system[9]. In order to have a suitable format, able to fit in the page, the view is split in chunks. Then, these chunks are placed side by side, with blue lines in the middle. Inside each chunk all the products are displayed in the very left, with a vertically alignment. Then, for each product all its revisions are shown horizontally aligned in a time based scale. The time value is taken according to the revision check-in time stamp. In other words, the distance between two revisions is related to the time between the first and the second check-ins.

Figure 4.12 shows that, at the system scope, the Product TimeLine View appears as a nebula of points. From the "density patterns" present inside it, we can deduce important information about the structure of the system. For example, a vertical line denotes a big commit, involving all the products aligned with the line. In general, we can assume that products belonging to the same density pattern are somehow correlated. Some examples taken from Figure 4.12 are:

1. Few commits performed at the same time. This is the situation in which we assume the strongest correlation. If the products involved belong to different modules, then there could be a logical coupling between the modules.

2. A lot of commits having similar time distribution. Such a large amount of check-ins implies that the products involved are important for the system.

3. Commits concentrated in the first half of the system time life.

4. Commits concentrated in the middle of the system time life.

5. Commits concentrated in the second half of the system time life.

Any time a density pattern is found, we can generate another view using the products belonging to the pattern only.

[9]Where "entire" means all the .cpp files.

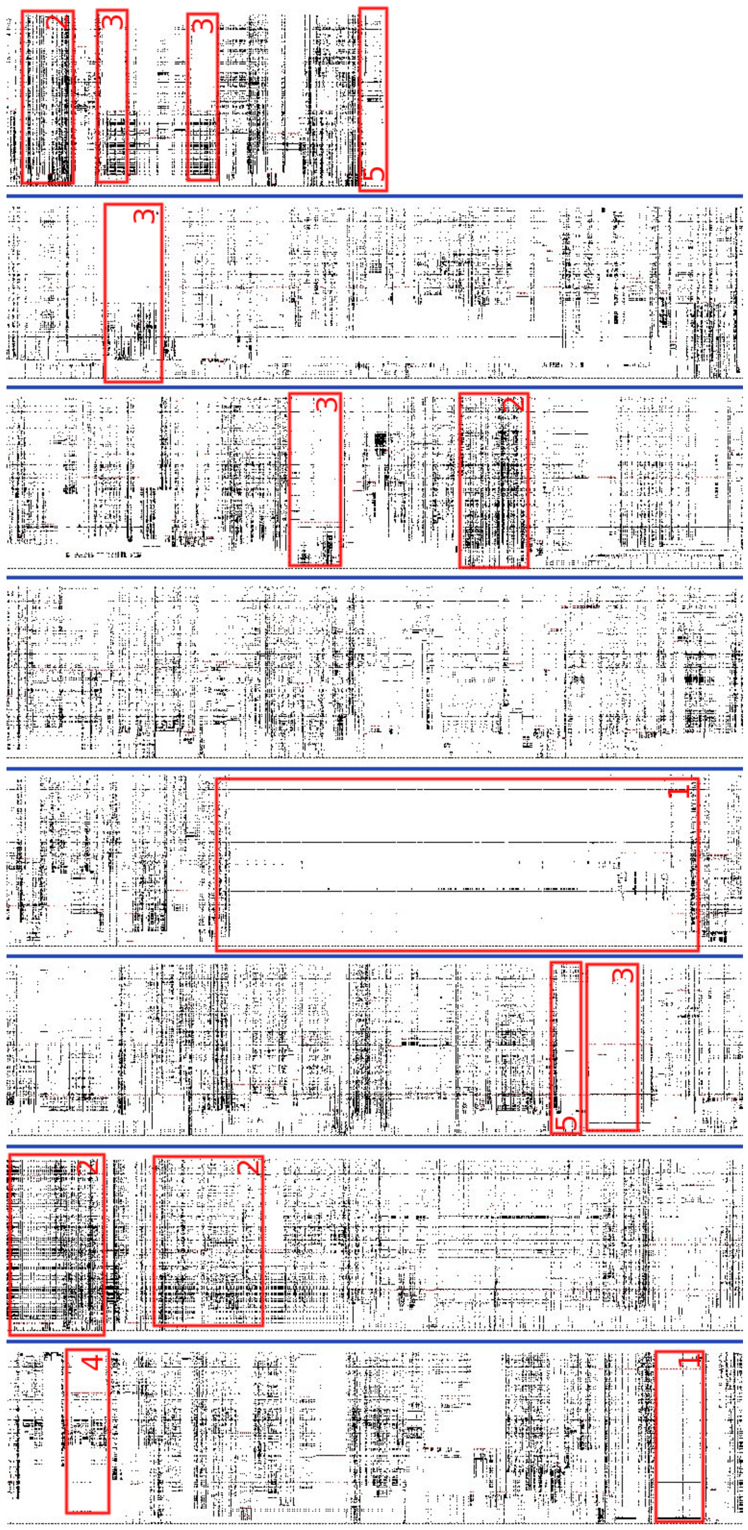


Figure 4.12: The Product TimeLine View applied to the entire Mozilla system.

4.6 Related work

In [HP96] Holt and Pak use a tool, called GASE, to explore software evolution. Their approach uses colors to contrast new, common and deleted parts of a software system. With this scheme, the developer can easily see structural change that might otherwise be difficult to discern.

Fischer *et al.* [FPG03a] combine release history data with information from problem reports to detect otherwise hidden relationships between features. They suggest first to instrument and track features, secondly to establish the relationships of modification and problem reports to these features, and thirdly to visualize the tracked features for illustrating their non apparent dependencies. Such visualization of interwoven features can indicate locations of design erosion in the architectural evolution of a software system.

Ball and Eick [BE96] concentrated on visualization of different aspects related to code-level such as code version history, difference between releases, static properties of code, code profiling and execution hot spots, and program slices. The basic concepts used in the visualization of the above aspects are colors and pixel representations of source code lines.

In [JGR99] Gall *et al.* presented an approach to use color and 3D to visualize the evolution history of large software systems. Colors were primarily used to highlight main events of the system evolution and reveal unstable areas of the system. In the interactive 3D presentation it is possible to navigate through the structure of the system at a coarse level and inspect several releases of the software system.

4.7 Conclusion

Understanding the evolution and the structure of large software systems is a complex task. The amount of information describing the history of a system is usually huge, and their interpretation is far from being immediate. There are many different aspects that have to be taken into account to evaluate the software quality and to deduce how the system could evolve from now on.

4.7.1 Summary

In this Chapter we have presented two clusters of polymetric views. They address the problem of the system understanding from both the structural and the evolutionary points of view. Each view has the capability to highlight one (or even more) particularity of the software system under inspection. From their combination we are able to get an insight into the system.

For each view, we have explained how it should be used. We have also suggested possible inner pattern interpretations. Furthermore, real examples, showing how the polymetric views act in practice, have been provided.

4.7.2 Benefits

The main benefits of our approach are the following:

- *Reduction of complexity.* It is possible to understand the structure of huge and complex software projects without having to read source code at all.
- *Scalability.* Each polymetric view is able to give a great amount of information in a condensed way. Moreover all the proposed views scale up to large software systems, *e.g.*, more than 2 *Mloc*.
- *Applicability.* Our approach is language independent. It is not only applicable to any programming language, but also to most of the software paradigms.
- *Use of bugs as cross entities.* Analyzing how bugs are shared by different modules makes it possible to uncover otherwise hidden dependencies. This allows to gain a good knowledge of the modules relationships.
- *Customizability.* The polymetric views are highly customizable by changing the metrics, the layouts, the figures and the figure parameters. These changes are important, because every software system has its particularities to which the view must be adapted to.

4.7.3 Limits

Our approach is limited in the following way:

- *Computation time.* The huge quantity of information considered makes the computation of some views (Discrete Time Combo Module View, Module Bugs Correlation View and Product TimeLine View) very long. As an example, the time required to render the Product TimeLine View shown in Figure 4.12 was about 50 hours[10].

[10] On a Pentium4 machine with 1GB of RAM using Suse9.1[Sus] as operating system.

Chapter 5

Software Archaeology in the Small

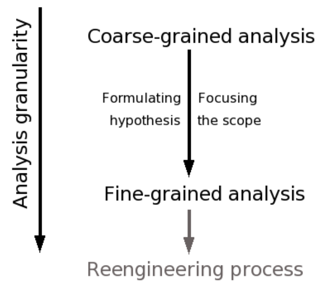
In Chapter 4 we have seen how to detect the principal modules of a subject system. We have presented a set of views that give a first insight of the system, helping to classify its modules. The following step in the evolution analysis is played by the fine-grained study. Its main goal is to understand the internal structure of the modules, going from the directory to the single revision point of view. The fine-grained analysis is designed to figure out how the responsibilities are allocated in a single module and among different modules. This knowledge not only could be helpful for a reengineering process, but could also act as the process starting point.

While for the coarse-grained views the scope was fixed at the system level, now it can vary from the module to the single revision point of view.

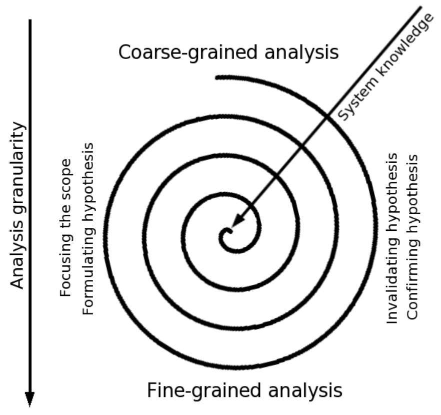
5.1 Introduction

Before going into detail with the fine-grained approach, we explain the relation between the two methods. As a first thought, one could believe that the two analysis are self-contained processes, with the only restriction of performing the fine-grained after the coarse-grained analysis. Once the latter is achieved, the former should start, on the basis of the knowledge acquired (as depicted in Figure 5.1(a)). From this point of view, the information gained with the fine-grained phase completes that taken from the previous analysis, where the two knowledge sets have an empty intersection. The entire analysis finishes together with the fine-grained phase, making it possible to start a reengineering process.

This analysis model is not correct: The fine and the coarse-grained phases are strongly coupled. It is impossible to perform them as independent processes. On the contrary, they give feedback to each other, allowing a deep understanding of the system.



(a) Cascade model.



(b) Spiral model.

Figure 5.1: Evolution analysis schemes.

A possible interaction between the two stages is shown in Figure 5.1(b). As we see, an hypothesis is formulated in the coarse-grained phase and it is either confirmed or invalidated using the fine-grained information. Another possibility is to perform the coarse-grained stage in order to focus the fine-grained one to a particular entity or a particular relation between entities. The overall model can be represented with a spiral in which the two stages alternate each other, increasing our knowledge of the system.

5.2 Analyzing the system at the directory granularity

The first phase of the evolution analysis is focused on the directories. We want to figure out which are the directories containing a large amount of products and which relations hold among them. We also want to understand how the responsibilities and the bugs are distributed among the directories.

The cluster of views designed for studying a subject system at the directory granularity is composed of:

1. Critical Directory Tree View (see Section 5.2.1).
2. Discrete Time Directory Tree View (see Section 5.2.2).
3. Fractal Directory Tree View (see Section 5.2.3).
4. Directory Bugs Correlation View (see Section 5.2.4).
5. Directory Black Holes View (see Section 5.2.5).

5.2.1 The Critical Directory Tree View

Name	CRITICAL DIRECTORY TREE VIEW	
Layout	Tree	
Scope	Module or Hierarchy	
Nodes	**Entity**	Directory
	Figure	Directory Figure
Edges	Directory to Subdirectories	
Metrics	**Size{Width ; Height}**	{- ; *nop*}
	Position{x ; y}	{- ; -}
	Color	*nob*
Sorting	Horizontal alphabetical order for hierarchies	
Appearance	See Figure 5.2	

Main Idea

This view gives a first insight of a subject module in terms of directories.

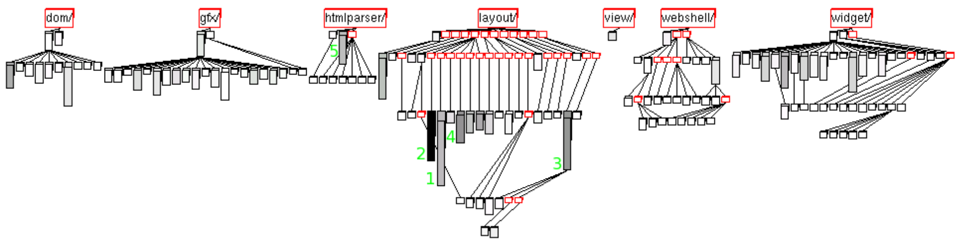


Figure 5.2: The CRITICAL DIRECTORY TREE VIEW applied to the SeaMonkeyLayout module of Mozilla. As metrics we use the number of products for the figure height and the number of bugs for the figure color.

Notes

The CRITICAL DIRECTORY TREE VIEW displays the hierarchies of directories composing a module. It helps us to understand the inner structure of a module and to find the so called *critical directories*. The critical directories are characterized by containing a great percentage of the module products and bugs. They are likely to hold also a considerable amount of the module responsibilities.

With respect to Figure 5.2, the figures having the lines colored in red represent *container directories*, which are directories including subdirectories only (no products). Since they have a structural function only, without any responsibilities, we choose to distinguish them from the other directories.

Key Points

- The tall and dark figures represent the critical directories, which contain a lot of products and bugs. We need to further inspect them at the product granularity to figure out the distribution of both bugs and revisions among products. Moreover this type of directories are likely to contain God products[1].
- The figures having red lines represent the container directories.
- Short and dark figures are small directories affected by a large amount of bugs. Some possible reasons of such a pattern are:
    1. There are few products but they have a large number of revisions. This implies that these products changed a lot during the directory life time. Is it because they was badly designed? Is it due to the key role they play? We need to analyze the internal structure of the directory to answer these questions.
    2. There are few products having few revisions too. The large amount of bugs could be due to bad design or bad implementation.
    3. All the bugs, or most of them, affect a single product, which is likely to be a God product.

  The number of revisions metric plays a key role with these entities. Before proceeding with the internal analysis, it would be better to compare the view with the CRITICAL DIRECTORY REVISION TREE VIEW variation (explained in the variations Section).

Example

Figure 5.2 shows the view applied to the SeaMonkeyLayout module of Mozilla. The key hierarchy is the *Layout* one, since it contains the following critical directories:

- **layout/html/content/src/** (marked as 1). It is the biggest in the module with a *nor* value equal to 119 and a *nob* value equal to 821.
- **layout/html/base/src/** (marked as 2). It is the most affected by bugs with a *nor* value equal to 77 and a *nob* value equal to 3114.
- **layout/xul/base/src/** (marked as 3). It has the second highest value of *nor* equal to 92 and a *nob* value equal to 1280.

[1] A God product is the equivalent of a God class[Rie96] in a CVS context.

- **layout/html/forms/src/** (marked as 4). It has the second highest value of *nob* equal to 1320 and a *nor* value equal to 45.

We also observe the presence of:

- The **webshell** hierarchy composed of small directories only.
- The **htmlparser/src/** directory (marked as 5) which is critical for the **htmlparser** hierarchy.
- The **dom**, **gfx** and **widget** hierarchies which include average directories from both the products and bugs points of view.

These observations are confirmed by the Critical Directory Revision Tree View depicted in Figure 5.3 (especially the key role of the **layout** hierarchy).

Customization

None.

Variations

The number of revisions could be used for the height metric, instead of the number of products. The best way to use this new view (called Critical Directory Revision Tree View) is in combination with the default one. The variation is helpful to find out on which directories the majority of the development effort was spent. However, this variation of the view has a shortcoming: It does not scale up well, as we can see from Figure 5.3.

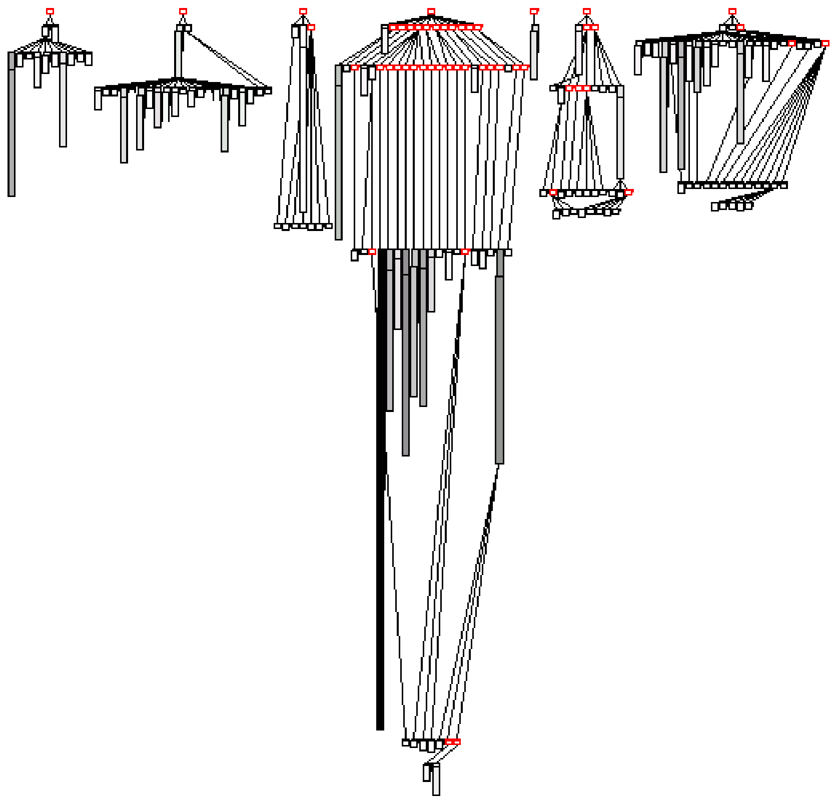


Figure 5.3: The CRITICAL DIRECTORY REVISION TREE VIEW applied to the SeaMonkeyLayout module of Mozilla. As metrics we use the number of revisions for the figure height and the number of bugs for the figure color.

5.2.2 The Discrete Time Directory Tree View

Name	Discrete Time Directory Tree View	
Layout	Tree	
Scope	Module or Hierarchy	
Nodes	**Entity**	Directory
	Figure	Discrete Time Directory Figure
Edges	Directory to Subdirectories	
Metrics	**Size{Width ; Height}**	{- ; -}
	Position{x ; y}	{- ; -}
	Color	-
Sorting	Horizontal alphabetical order for hierarchies	
Appearance	See Figure 5.4	

Main Idea

The purpose of the Discrete Time Directory Tree View is to understand how the work is distributed not only over the different hierarchies, but also over time. The view gives an idea of the directories liveness.

Notes

This view allows us to figure out the relationships among directories, according to particular patterns we describe in detail in the Key Points description. The view is also helpful to understand how responsibilities move from a directory to another one. This is possible by looking at the directories birth and death dates.

Key Points

- Figures which contain a lot of red and no black rectangles represent directories that lived for the whole system life time. Since the production of revisions for these directories was always high, they are important not only for their hierarchy, but also for the entire module.

- Figures composed of blue rectangles only represent directories which contain products that changed a little during the system life time. These products could be related to the definition of constants.

- A set of figures like those shown in Figure 5.5(a) and 5.5(b)[2] could represent a movement of responsibilities. As we can see from the

[2]The examples are taken from the points marked as 1 and 2 in Figure 5.4.

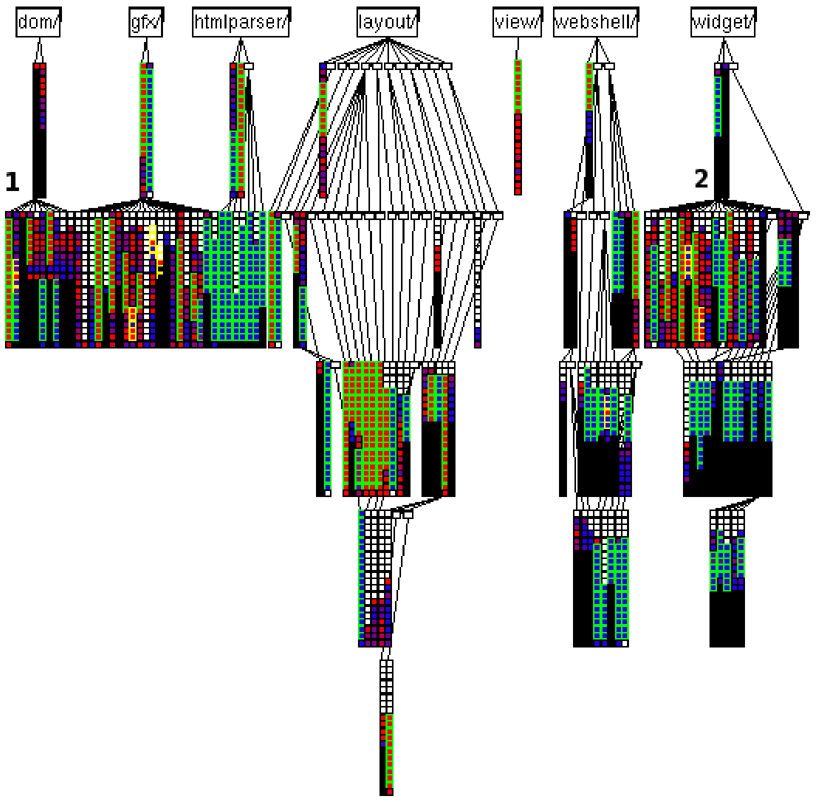
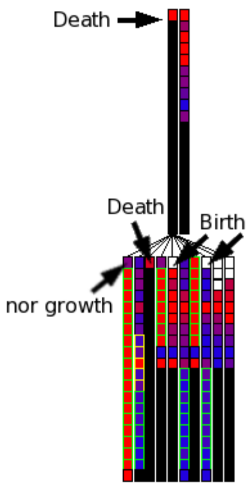


Figure 5.4: The Discrete Time Directory Tree View applied to the SeaMonkeyLayout module of Mozilla.

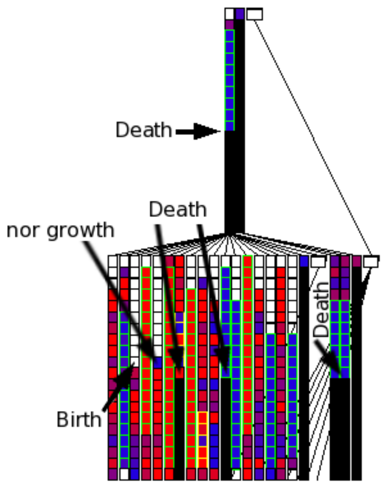
examples, such a situation is characterized by a set of events which happen together (or almost together). These events are:

1. The death of one or more directories, recognizable by the change (*Not black rectangle* $\Rightarrow$ *Black rectangle*) in the sequence of rectangles. The more the figure is red before its death, the more the moving of responsibilities is likely to have happened, since the directory was important for the hierarchy until that point.
2. The birth of one or more directories, recognizable by the change (*Transparent rectangle* $\Rightarrow$ *Not transparent rectangle*) in the sequence of rectangles.
3. An increasing of the *nor* value, recognizable by the change (*Blue rectangle* $\Rightarrow$ *Red rectangle*) in the sequence of rectangles.

Let d_1 be the directory involved in the death event and d_2, d_3 the directories involved in the birth and *nor* growth events respectively. d_1 is the starting point entity, while d_2 and d_3 are the possible target entities. From our experience, d_2 and d_3 are often either subdirectories or "*sibling*" directories of d_1, never parent directories. The reason could be that d_2 and d_3 include a specialization of the contents of d_1. For example, they could represent a porting to a particular platform or operating system, while d_1 is the general version. From an overall point of view, the moving of responsibilities towards subdirectories suggests that the Mozilla system becomes more and more complex and distributed over different directories.



(a) The **dom** hierarchy of Mozilla.



(b) The **widget** hierarchy of Mozilla.

Figure 5.5: Two examples of movement of responsibilities.

Example

Figure 5.4 is an example of the view applied to the SeaMonkeyLayout module of Mozilla. We already notice the presence of two movements of responsibilities in the Key Points discussion.

We observe that all the directories at level one have existed since the beginning of the system. As we can see from Table 5.1, the more the level increases, the more the percentage of directories which was introduced late grows (with respect to the total number of directories belonging to that particular level). This fact confirms the hypothesis drawn above: During its life time the Mozilla system became more and more complex and distributed over different directories.

The second consideration concerns the Layout hierarchy. The high percentage of *Hot directories* (this name derives from the *Hot Module* name

Level	Late birth	Total	Percentage of late birth
0	Only containers		
1	0	11	0%
2	39	70	55%
3	30	39	76%
4	17	18	94%
5	2	2	100%

Table 5.1: SeaMonkeyLayout module late birth directories.

introduced in Section 4.3.1) it contains, with respect to the other hierarchies, confirms that this hierarchy is important for its parent module, as we conclude using the Critical Directory Tree View.

Customization

The choice of the figure dimensions is fundamental using the Tree Layout. We have two needs which are in contradiction:

- Short figures in order to fit them on the screen.
- A sufficient number of time intervals in order to have figures rich of information.

The problem is that the more the figures are short, the lower the number of time intervals (in which the figures are divided) are. A too small number of time intervals implies that the granularity of the information included in the figures is too coarse to be useful. Satisfying both the needs (for views containing a considerable amount of directories) will result in figures for which the line colors only are visible, while the internal colors are not. However, we are going to see that this is not necessarily a shortcoming. Figure 5.6 is an example of such a situation. We see that the phases only are visible, because they are represented by the line colors. Semantically, it is like rising the level of abstraction, since now we reason in terms of phases. With this kind of reasoning there is a loss of information because phases contain differential data. They tell us about the differences among the values of *nor* (or *nob*) in different time periods, whereas nothing is said about their absolute values. For example, a stable phase (green) could be related to both high and low values of *nor* (or *nob*). In the same way, a peak phase (yellow) could mean either a high value of *nor* between low values or vice versa.

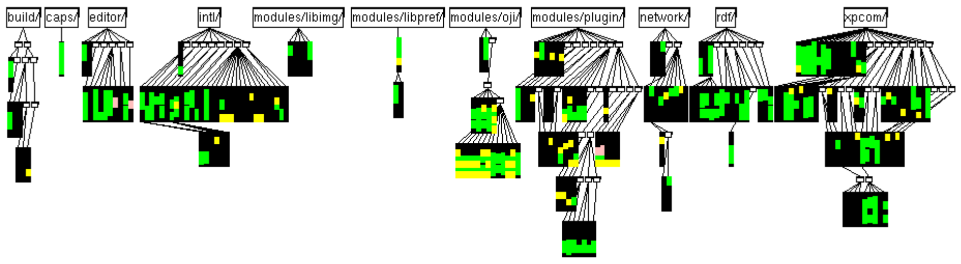


Figure 5.6: The scalability of the Discrete Time Directory Tree View. The view is applied to the RaptorDist module of Mozilla.

Variations

Both the Discrete Time Bug Figure and the Discrete Time Combo Figure can be used instead of the Discrete Time Revision Figure.
Using the Combo Figure the view is difficult to understand. The Bug and the Revision figures are side by side and it is hard to distinguish them, especially with large scale views.

5.2.3 The Fractal Directory Tree View

Name	Fractal Directory Tree View	
Layout	Tree	
Scope	Module or Hierarchy	
Nodes	**Entity**	Directory
	Figure	Fractal Fixed Width Figure
Edges	Directory to Subdirectories	
Metrics	**Size{Width ; Height}**	{- ; Number of Bugs}
	Position{x ; y}	{- ; -}
	Color	-
Sorting	Horizontal alphabetical order for hierarchies	
Appearance	See Figure 5.7	

Main Idea

The purpose of the Fractal Directory Tree View is to understand how the work spent on the directories composing a subject module was distributed among different authors.

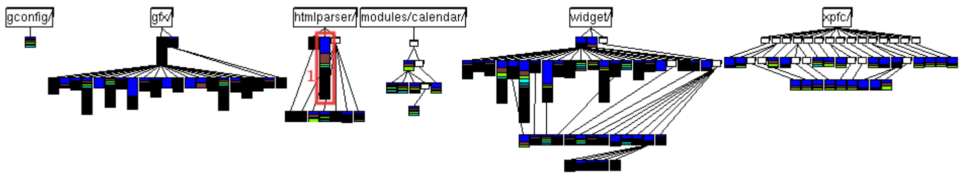


Figure 5.7: The Fractal Directory Tree View applied to the CalendarClient module of Mozilla. As metric we use the number of bugs for the figure height.

Notes

A directory can be mainly developed by few authors (few developers patterns), or the work can be equally divided over a large number of developers (a lot of developers patterns)[3]. This view helps us to figure out the hierarchies development types by looking at the directories belonging to them. We distinguish the following hierarchies:

- *Many developers.* It is composed of *mainly one developer* and *a lot of developers* directories.

[3]The relation between development types and figure appearances is explained in Section 4.3.2.

- *Few developers.* It is composed of *one developer* and *few developers* directories.
- *Mixed.* It contains all the directory types.

Using this view, we have also the opportunity to analyze the relation between different directories on the basis of the similarity between the figures representing them. In fact, similar development models, that is similar figures, could be related to a logical coupling. The more the figures are similar, the higher the probability of the coupling is.

In the end we can study how the number of bugs is related to the development model.

Key Points

- The more the Fractal Value of a directory tends to 1, the more this directory is likely to be affected by a large number of bugs[4].
- Tall figures having few predominant colors represent mainly one developer directories affected by a great amount of bugs.
- Figures with similar patterns could be logically coupled. We need to further inspect them.

Example

Looking at Figure 5.7 we distinguish the following hierarchies development models: **gconfig**, **modules/calendar** and **xpfc** are few developers while **gfx**, **htmlparser** and **widget** are Mixed. In the **htmlparser** hierarchy we notice the presence of a mainly one developer directory (marked as 1 in Figure 5.7) which is affected by a large number of bugs. We should further inspect it.

Before concluding we focus our attention on the widget hierarchy, shown in Figure 5.8. All the third and fourth levels directories have a small amount of bugs, while they are different from the development type point of view. Concerning the second level directories they are heterogeneous from both the development model and the number of bugs points of view. The directory marked as 1 has a high value of *nob* and a mainly one developer pattern. We should analyze it at the product granularity.

[4] This observation is based on the data retrieved from the Mozilla system.

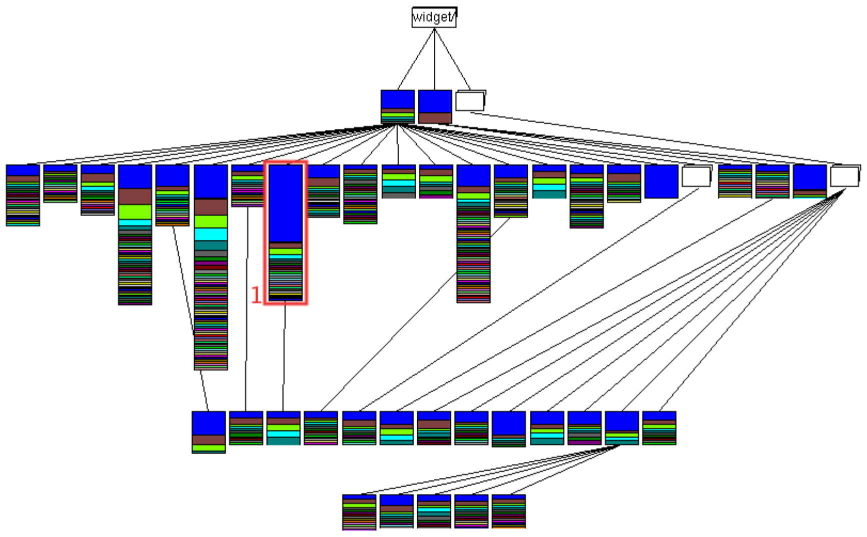


Figure 5.8: The Fractal Directory Tree View applied to the **widget** hierarchy of Mozilla.

Customization

None.

Variations

Both the number of products and the number of revisions can be used for the height metric, instead of the number of bugs.

Using the number of products we can detect big directories having a mainly one developer or few developers patterns, in which few authors are responsible for a lot of products. This distribution of the work can be due to:

- The products composing the directory have a small number of revisions, perhaps checked-in at the same time by the same author (an example is shown in Figure 5.13(b)).
- The directory contains a localized and specific knowledge owned by one or few authors only.

Using the number of revisions we can detect directories characterized by having a large amount of revisions and a mainly one developer pattern. We need to inspect their internal structure in order to discover how the work is distributed among the products. The central role of an author could be either “on average”, that is generated by the majority of the products, or due to few predominant products (see Section 5.3.3). Such products are likely to be God products.

There is a last variation of the Fractal Directory Tree View, which consists in using the normal Fractal Figure instead of the Fractal Fixed Width Figure. The benefit of the variation is the possibility of using both the height and the width for mapping metrics. On the other hand, using these figures the view is less scalable than the Fractal Directory Tree View and hard to interpret.

5.2.4 The Directory Bugs Correlation View

Name	Directory Bugs Correlation View	
Layout	Circle	
Scope	Module or hierarchy	
Nodes	**Entity**	Directory
	Figure	Directory Figure
Edges	Directory to Directory	
Metrics	***Directory Node***	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{- ; -}
	Color	Number of Revisions
	Directory To Directory Edge	
	Width	Number of Shared Bugs
	Color	Number of Shared Bugs
Sorting	-	
Appearance	See Figure 5.9	

Main Idea

The purpose of the Directory Bugs Correlation View is to detect dependencies between different directories composing a subject module.

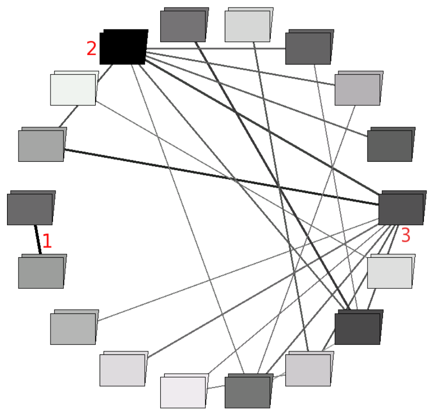


Figure 5.9: The Directory Bugs Correlation View applied to the SeaMonkeyLayout module of Mozilla. As metrics we use the number of revisions for the figure color and the number of shared bugs for both the edge color and width.

Notes

This view is a variation of the MODULE BUGS CORRELATION VIEW. As a consequence it is just described in the previous Chapter (see Section 4.5.3). All we want to add is that, since the intersection between different directories is always empty, the so called "*false correlations*" do not exist. Thus, we don't need to perform the validation phase, which is instead required with the MODULE BUGS CORRELATION VIEW.

A similar view exists also at the product granularity, which is called PRODUCT BUGS CORRELATION VIEW. The scope, in this case, can vary from a hierarchy to a single directory. The false correlations still do not exist. Examples of the DIRECTORY BUGS CORRELATION VIEW and PRODUCT BUGS CORRELATION VIEW are reported in the following Chapter (see Section 6.2.4).

5.2.5 The Directory Black Holes View

Name	DIRECTORY BLACK HOLES VIEW	
Layout	Black Holes	
Scope	Module	
Nodes	**Entity**	Directory
	Figure	Directory Figure
	Entity	Bug
	Figure	Bug Figure
Edges	Directory To Bug	
Metrics	***Directory***	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{- ; -}
	Color	-
	Bug	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{- ; -}
	Color	-
Other mappings	Bug color	Bug owner
Sorting	Number of bugs for the directories	
Appearance	See Figure 5.10	

Main Idea

This view gives a detailed idea of the dependencies between directories composing a subject module. It allows also to detect the most shared bugs.

Notes

The DIRECTORY BLACK HOLES VIEW makes use of an ad-hoc layout called Black Holes. A set of directories (or more generally a set of entities) are displayed using a checkerboard layout, showing for each of them all its bugs circularly aligned around the directory. The cross-over edges represent shared bugs.

This view can be thought as a specialization of the DIRECTORY BUGS CORRELATION VIEW. The information encapsulated by the edges in the latter, that is the bugs shared, is now explicitly shown. Thus, we are again in a position to understand the directories dependencies. We can also figure out how the correlations are structured.
The view allows us to detect, and then to analyze, bugs which are shared by many directories. They are likely to include interesting data in the bug comment field. A useful piece of information we can exploit concerns the

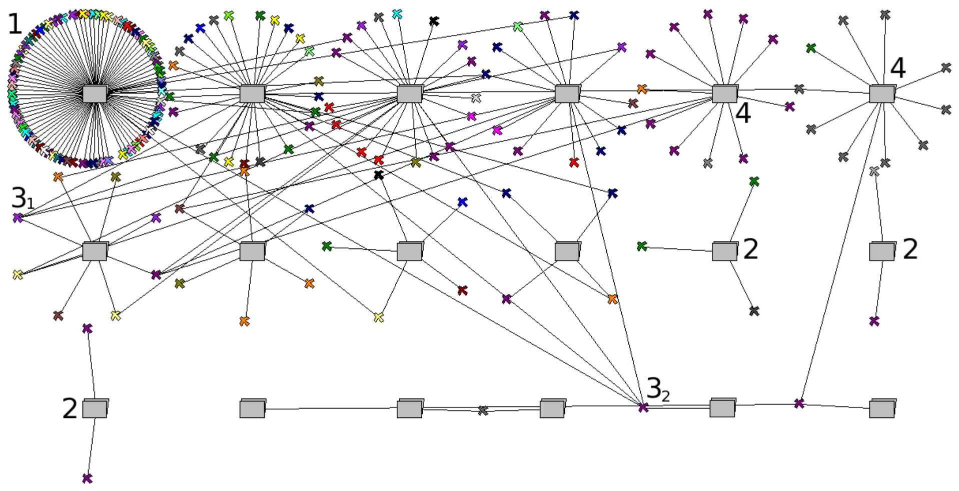


Figure 5.10: The DIRECTORY BLACK HOLES VIEW applied to the ThunderbirdTinderbox module of Mozilla.

owners of the bugs. Looking at the bug figure colors we can distinguish the directories according to the ratio $r = \frac{\text{number of different bugs owners}}{\text{number of bugs}}$:

- Low values of r are related to one owner or mainly one owner.
- High values of r are related to a lot of owners.

The DIRECTORY BLACK HOLES VIEW has a drawback: It is less scalable than the BUGS CORRELATION VIEW. When applied to big modules, it is hard to interpret.

Key Points

- Directory figures surrounded by a large amount of bug figures represent critical directories. Since they look like black holes, both the view and the layout get this name.
- Directories which do not share bugs are called *self-contained*. They are likely to encapsulate specific data or responsibilities, especially if the number of different bug owners is small.
- Bugs linked with many directories represent correlations between directories. Therefore, these bugs can be helpful to understand the directory dependencies, looking at the description and bug comment fields.

Example

Figure 5.10 depicts an example of the Directory Black Holes View applied to the ThunderbirdTinderbox module of Mozilla, in which we find:

- A black hole figure (marked as 1) representing a critical directory. It is characterized by a large number of different bug owners.
- Three *self-contained* directories (marked as 2). Since they are affected by only two or three bugs, we do not need to further inspect them.
- Two directories (marked as 4) having a "mainly one owner" pattern.
- Two bugs shared by four and five directories respectively (marked as 3_1 and 3_2). Reading the bug comments we discovered that the problems are related to:
  - A virtual function that should not be virtual, for the bug marked as 3_1.
  - A "*general purpose stack*" which creates a lot of confusion, for the bug marked as 3_2.

  Since the resolution field is *fixed* for both the bugs, we found also the solutions and the patches for these problems.

Customization

None.

Variations

The bug gravity or priority can be used for the bug figure color, instead of the owner. In this way, we are able to answer questions like: Are the most shared bugs those with the highest priority (or gravity)?

Another variation consists in altering the view scope: From the module to the directory context. Nothing changes but the directory figures are substituted by product figures (the view is called Product Black Holes View). All the observations and the key points still be valid, taking into account that now we are arguing about products.

5.3 Analyzing the system at the product granularity

The second phase of the evolution analysis is focused on the products. We want to find out which are the biggest products with respect to either the number of revisions or the number of lines of code or both (God Products). We are also interested in understanding which relations hold among the products and how the responsibilities and the bugs are distributed among them.

The cluster of views designed for studying a subject system at the product granularity is composed of:

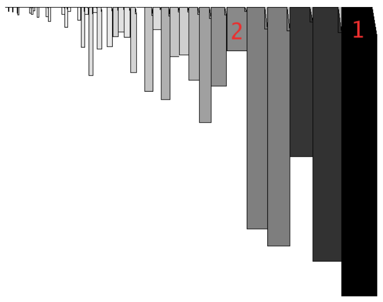
1. God Product View (see Section 5.3.1).
2. Product TimeLine View (see Section 5.3.2).
3. Fractal Product View (see Section 5.3.3).

5.3.1 The God Product View

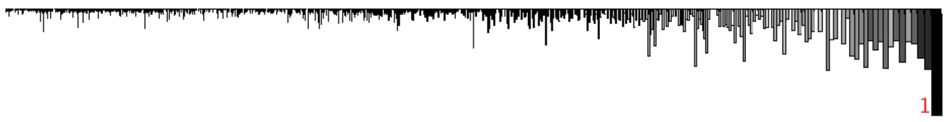
Name	God Product View	
Layout	Horizontal line	
Scope	From the directory to the entire system	
Nodes	**Entity**	Product
	Figure	Product Figure
Edges	-	
Metrics	**Size{Width ; Height}**	{*nor* ; *loc*}
	Position{x ; y}	{- ; -}
	Color	Number of Bugs
Sorting	Number of revisions	
Appearance	See Figure 5.11	

Main Idea

This view gives a first insight of a high level entity (a module, a hierarchy of directories or a directory) in terms of products. It also allows us to detect *God Product*.



(a) Directory Scope (**layout/html/base/src** of Mozilla).



(b) Module Scope (SeaMonkeyLayout of Mozilla).

Figure 5.11: Two applications of the God Product View with different scopes. As metrics we use the number of revisions for the width, the number of lines of code for the height and the number of bugs for the color.

Notes

The God Product View is able to scale up to the entire system scope. It is presented in this Section because it is usually used either in a directory or in a hierarchy context. The view shows all the products belonging to a particular entity enriched with the *nob*, *nor* and *loc* metrics. In this way, we can detect the most critical products, which are characterized as follows:

- Huge size in terms of lines of code.
- A large amount of revisions.
- A large number of bugs.

All the properties mentioned above can be summarized by stating that a God product tends to own too many responsibilities. For this reason the presence of such an entity is a symptom of bad design.

Key Points

- Tall, wide and dark figures represent products which are likely to be God products. We need to perform a code inspection to validate this hypothesis.
- Wide and short figures represent relatively small products (in terms of lines of code) on which a great development effort was spent. If the figures are also dark, then the corresponding products are likely to play a key role in the higher level entity (module or directory). It is also possible that the products encapsulate complex data. We can confirm these suppositions by performing a code inspection.
- Flat figures represent dead products. Since the files are not present in the CVS checked-out system, the *loc* values are conventionally set to zero (see Section 3.3).

Example

Figure 5.11 shows two examples of the God Product View applied to the Mozilla system. Concerning Figure 5.11(a) the most interesting product is the one marked as 1. It has the highest values of *nor*, *loc* and *nob* (951, 7835 and 380 respectively) and thus it is likely to be a God product. The product marked as 2 with "just" 1186 lines of code, has 543 revisions and 176 bugs. Looking at Figure 5.11(b) we detect another God product candidate (marked as 1). Its *nor*, *loc* and *nob* are the highest in the whole set of products (1301, 13683 and 457 respectively).

Customization and Variations

None.

5.3.2 The Product TimeLine View

Name	**Product TimeLine View**	
Layout	Time Based Evolution Matrix	
Scope	From the System to the single Product	
Nodes	**Entity**	Product
	Figure	Product Figure
	Entity	Revision
	Figure	Rectangle Figure
Edges	-	
Metrics	***Product***	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{- ; -}
	Color	-
	Revision	
	Size{Width ; Height}	{- ; -}
	Position{x ; y}	{Time ; - }
	Color	-
Other mappings	Revision y position	Corresponding Product
	Revision Color	Revision Author
Sorting	Vertical alphabetical order for the products	
Appearance	See Figure 5.13	

Main Idea

This view gives a detailed idea of how the high level entity (a module, a hierarchy of directories or a directory) was developed. To do that it exploits the check-in information: The time stamp and the author. Performing a careful inspection of the view, we have the opportunity to gather information concerning both the products and their dependencies.

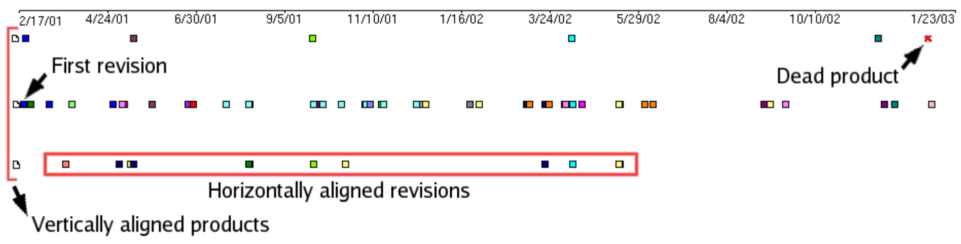


Figure 5.12: The structure of the PRODUCT TIMELINE VIEW.

Notes

The Product TimeLine View makes use of an ad-hoc layout called Time Based Evolution Matrix. Figure 5.12 shows a simple example of the view, which describes the structure of the layout. All the products are displayed to the very left using a vertical alignment. Then, for each product, all its revisions are rendered according to the following rules:

1. The vertical position is equal to the corresponding product vertical position. Thus, all the revisions belonging to a product are horizontally aligned among themselves and with the product.

2. The horizontal position is computed using the following formula:

$$x_i = x_{\min} + sf \times \big(nod(r_i) - \text{Min}_{r \in R}(nod(r))\big) \tag{5.1}$$

   where sf is a scale factor, R is the set composed of all the revisions in the view and $nod(r)$ is the difference between the time stamp of r and January 1, 1901, expressed in number of days (for the first checked-in revision $x_i = x_{\min}$). What equation 5.1 states is that, given two revisions referring to the same product, the horizontal distance between them is proportional to the time elapsed between the first and the second commits.

3. The color is related to the author whom performed the check-in. Equal colors correspond to the same author.

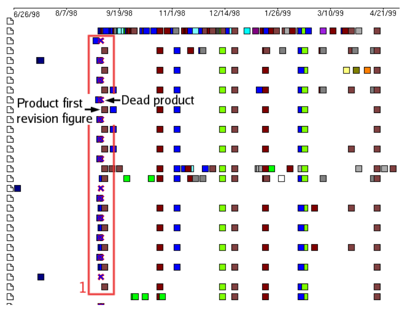
4. Cross-shaped figures with red boundaries represent the last revisions of dead products. Coloring the boundaries differently from all the other figures (for which the boundaries are black) makes the dead products visible even in large scale views.

Key Points

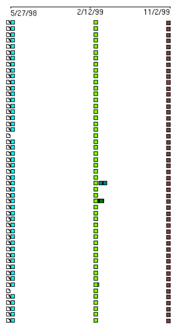
- Given a product, the first rectangle figure (from left to right) horizontally aligned with it represents the first revision of this product. The figure's horizontal position corresponds to the product birth date. We call such figures "*product first revision figures*".

- Vertically aligned figures having the same color represent a single check-in. Products sharing one or more check-ins are likely to be correlated. The higher the ratio $\frac{\text{number of check-ins in common}}{\text{total number of revisions}}$ is, the stronger the correlation is.

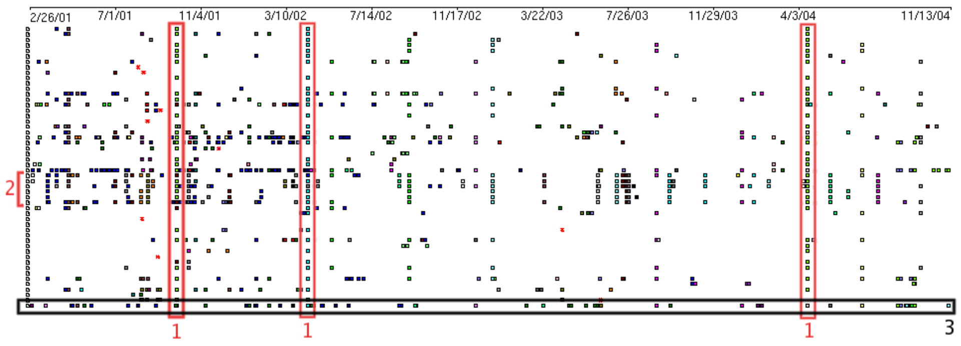
- Horizontal lines filled with a large amount of figures represent products which play key roles in the high level entity.



(a) A moving of responsibilities (the **layout/html/content/src** directory of Mozilla).



(b) A strong correlation (the **modules/calendar/src/libcal/ical** directory of Mozilla).



(c) The Directory **widget/src/mac** of Mozilla.

Figure 5.13: Three applications of the Product TimeLine View.

- One or more vertically aligned cross-shaped figures followed by one or more *product first revision figures* could denote a movement of responsibilities (as shown in Figure 5.13(a)).

We can either confirm or invalidate all these hypothesis by performing code inspections.

Example

In Figure 5.13(c) some examples of the previously introduced Key Points are depicted. The three red rectangles marked as 1 concern big commits involving almost all the living products of the directory. This implies a considerable correlation. The products marked as 2 are bound by a strong correlation, since they share a lot of check-ins. The last observation regarding the Figure 5.13(c) is about the product marked as 3. The large number of revisions it contains and their distribution over time indicate that this product is likely to play an important role in the directory.

Figure 5.13(a) shows an example of a moving of responsibilities (marked as 1). The death of some products is immediately followed by the birth of some others. Performing an inspection we discovered that the moving of responsibilities is nothing but a renaming (**nsHTMLBR.cpp** ⇒ **nsHTMLBRElement.cpp**, **nsHTMLBase.cpp** ⇒ **nsHTMLBaseElement.cpp**, and so on always with the "Element" suffix).

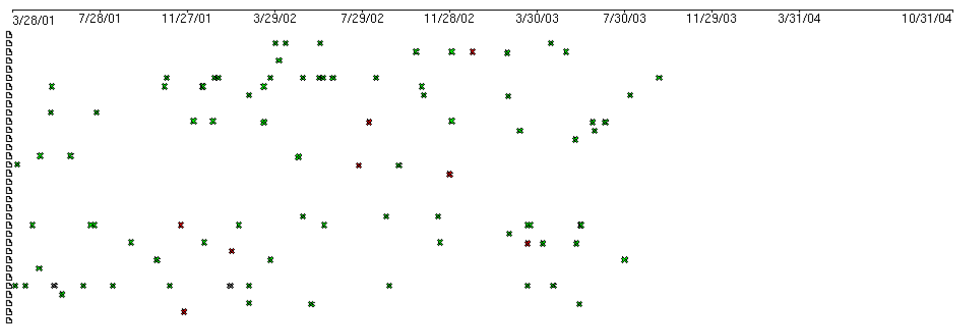
Customization

None.

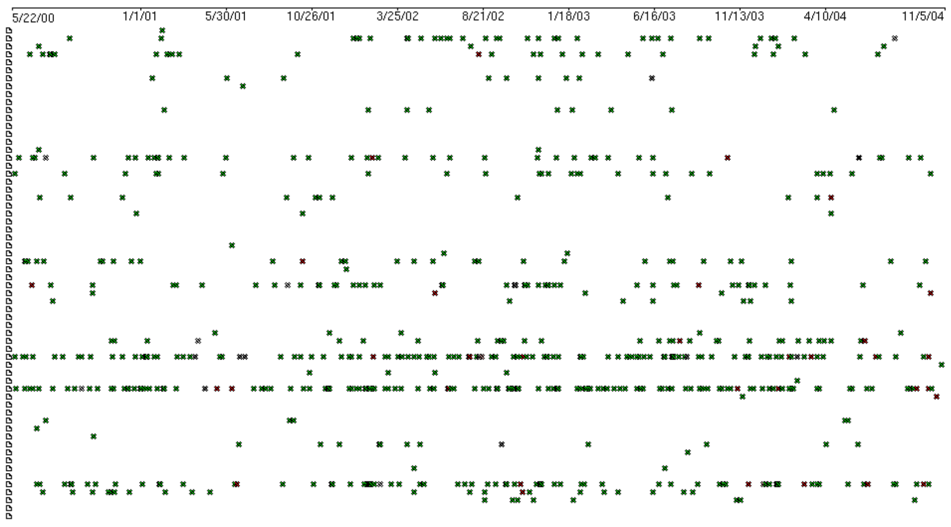
Variations

The Time Based Evolution Matrix Layout can be used with bugs instead of revisions. In this case, the horizontal position of a bug figure is computed according to the bug report time stamp. The figure color represents the bug Resolution (see Section 3.1.2 for detail) with the following correspondences: Green ⇒ fixed, red ⇒ notyet, pink ⇒ wontfix, white ⇒ worksforme, black ⇒ invalid, gray ⇒ duplicate or remind.

The PRODUCT BUGS TIMELINE VIEW, as the variation is called, can be helpful to argue about the high level entity maturity. A view having most of the bugs concentrate to the left suggests that the entity is either stable or no more developed (an example is depicted in Figure 5.14(a)). We expect that, from now on, the number of bugs for this entity will not increase a lot. On the other hand, a view having a large amount of bugs to the right denotes an entity either under development or under testing (an example is shows in Figure 5.14(b)). The number of bugs which will affect this entity from now on, will probably remain high. The same holds for a view containing a lot of bug figures, which are equally distributed over the horizontal space.



(a) The **widget/src/windows** directory of Mozilla. It is a stable or no more developed entity.



(b) The **gfx/src/beos** directory of Mozilla. It is an under development or testing entity.

Figure 5.14: Two examples of the Product Bugs TimeLine View variation.

5.3.3 The Fractal Product View

Name	FRACTAL PRODUCT VIEW	
Layout	Checkerboard	
Scope	From module to directory	
Nodes	**Entity**	Product
	Figure	Fractal Figure
Edges	-	
Metrics	**Size{Width ; Height}**	{*nor* ; *nor*}
	Position{x ; y}	{- ; -}
	Color	-
Sorting	Number of Revisions	
Appearance	See Figure 5.15	

Main Idea

The FRACTAL PRODUCT VIEW gives an impression on the development model of both the high level entity and its inner products. It helps us to understand in which way the products contribute to the high level entity development type. Have all of them more or less the same weight? Is there a big main product which influences the high level entity the most? The view makes us able to answers these questions.

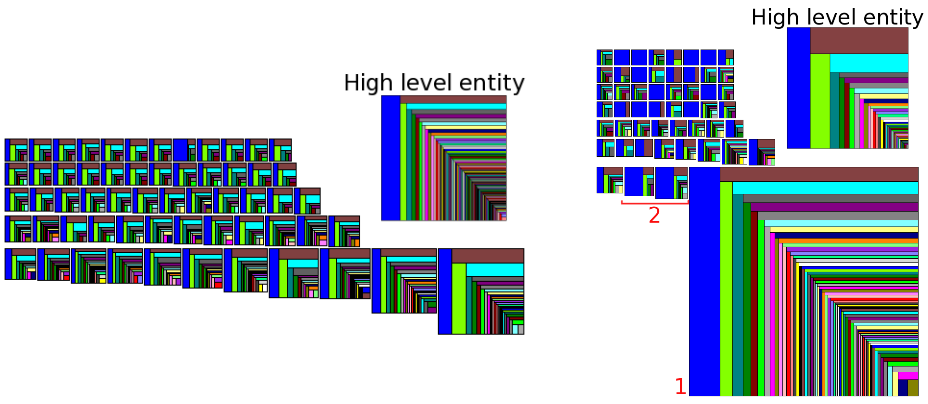
Notes

The main difference between the PRODUCT TIMELINE VIEW and the current view consists in the dimensions considered. The FRACTAL PRODUCT VIEW does not make use of the time dimension; only the authors information is taken into account. On one hand this is a shortcoming, since the view encapsulates less semantics. On the other hand, the choice of not considering the time makes the FRACTAL PRODUCT VIEW much more scalable than the PRODUCT TIMELINE VIEW, up to the module scope[5].

Key Points

- Views which contain figures having both similar sizes and similar patterns represent “balanced” high level entities. It means that all the products equally contribute to the high level entity development model.
- Figures having huge dimensions, with respect to the average figure size, represent products which are important in the high level entity context.

[5]In a large scale PRODUCT TIMELINE VIEW the internal rectangle colors, that are the authors, are not visible.



(a) Balanced weight (the **xpinstall/src** directory of Mozilla).

(b) Main product (the **editor/libeditor/html** directory of Mozilla).

Figure 5.15: Two applications of the FRACTAL PRODUCT VIEW. As metrics we use the number of revisions for the size.

The development types of the products are reflected in the high level entity. We need to perform a following analysis to understand whether such products own too many responsibilities or not.

- Figures characterized by having similar patterns[6] represent products which are likely to be logically coupled. This is especially true if the shared pattern is different from that characterizing the high level entity (for example a couple of “a lot of developers” products into a “mainly one developer” directory). The way to discover the reasons of the coupling is a code inspection.

Example

Figure 5.15 shows two examples of the view. For practicality’s sake, in the pictures the fractal figures representing the high level entities are also depicted.

Figure 5.15(a) is an example of a balanced directory, while a critical product is present in Figure 5.15(b)(marked as 1). In the second example, the figures marked as 2 are also interesting. They are in contradiction with the directory, since they have a “mainly one developer” pattern, while the directory is “a lot of developers” entity.

[6]The figure patterns represent the entity development types, as explained in Section 4.3.2.

Customization

None.

Variations

The number of bugs can be used for the size metric, instead of the number of revisions. In this way, big figures are related to critical products which are likely to be God products.

5.4 Related work

In [Lan01, Lan03b] Lanza depicts several releases of a software system in a matrix view. Each class is represented by a rectangle whereas opposite edges visualize a specific metric. Based on the evolution matrix classes are assigned to different evolution categories such as, for example, pulsar (class grows and shrinks repeatedly) or supernova (size of class suddenly explodes).

In [GL04] Gîrba and Lanza focus on the evolution of class hierarchies, which provide a grouping of classes based on their similar semantics. Thus, understanding a hierarchy as a whole reduces the complexity of understanding big systems. The proposed approach is based on polymetric views designed according to measurements which summarize the evolution of an entity or a set of entities.

In [DLS00] Ducasse *et al.* provide a query tool to identify software entities which are of interest regarding their life cycle in the context of the overall evolution of the whole system.

5.5 Conclusion

Comprehending the internal structure of a module allows to better understand both the evolution of a system [GJKT97] and its overall structure. Furthermore, from the inspection of a module, we can not only evaluate its design quality, but we can also find out where it shows design erosion. Such parts of the module represent candidates for a following reengineering process, aimed at improving the module design.

The problem is often difficult to tackle, especially when the module is big. The large amount of available information makes it necessary to find a way to easily represent the data.

5.5.1 Summary

In this Chapter we have described two clusters of polymetric views, addressing the problem of understanding the inner structure of a subject module. They show the module characteristics at different granularity levels: The first cluster works with directories, while the second one works with products. Changing the scope of the analysis allows us to gradually gain information concerning the entities which belong to the module. First of all, we figure out how the module is composed in terms of directory hierarchies. Then, we discover the directory relationships and, finally, we detect the most critical products.

For most of the views, we have presented which patterns can be found and which are their possible meanings. We have also introduced some guidelines to identify both symptoms of bad design and candidates for reengineering. As in the previous Chapter, real examples showing how the polymetric views act in practice are provided.

5.5.2 Benefits

The main benefits of our approach are the following:

- *Scalability, applicability and customizability.* Since they come from the use of the polymetric views, they are described in the previous Chapter (see Section 4.7.2).
- *Different points of view.* The module entity is shown from different points of view. In this way, we can either confirm or invalidate an hypothesis drawn with a view, using another view.
- *Reengineering candidates detection.* It makes our approach useful in a reengineering context.

5.5.3 Limits

Our approach is limited in the following way:

- *No source code analysis.* The only available data regarding the source code is the number of lines (*loc*). Information like the number of methods or attributes (in an Object Oriented context) are not provided.
- *Directory.* The level of abstraction between the module and the product is the directory. The problem is that a directory represents a container of products, without having any equivalent in most of the programming languages (with the exception of Java). Therefore, products belonging to the same directory are likely to share responsibilities, but this is not formally required.

Chapter 6

Software Archaeology: A Top-down Methodology

In this Chapter we present a complete top-down methodology to analyze a software system developed using CVS as version control system. The proposed approach does not consist of a fixed list of steps that have to be followed. On the contrary, it is composed of a set of guidelines which suggest how the coarse-grained and fine-grained views should be combined according to what we are looking for. There are many paths which can be followed, and the choice depends also on the system under analysis.

To better explain our evolution technique, it is contextually presented with a case-study. In fact, we are convinced that with principles it is easy to make generalizations. On the other hand, it is with examples that we can see what is going on. The example needed to guide us in the methodology explanation must meet the following requirements:

- The software must be developed using CVS as version control system and Bugzilla as bug tracking system.
- The CVS source must be available.
- The system must be so large that we need the help of a tool to study it.
- The software has many years of development.
- The software has a large number of versions.

Among many candidates which meet the listed requirements we have chosen the Mozilla system, which is just described in Chapter 3 (see Section 3.3.3).

The aim of the case-study is not performing a complete and exhaustive analysis of Mozilla, which besides would require a lot of time. It is just an example of how the views introduced in this thesis can interact in a real case.

6.1 Understanding the evolution of the system

The first part of the methodology concerns the study of the evolution of the software. In this context, the first aspect of interest is how the system under analysis grows and shrinks during its life time, with respect to well defined system metrics. We inspect this point using the SYSTEM GROWTH VIEWS. As a second point of view, we want to figure out how the work was distributed over time and which were the moments of maximum development of Mozilla. We do that by analyzing the trends of both the number of revisions and number of bugs over time using the SYSTEM PRODUCTION TREND VIEWS.

Before analyzing the views, we explain the choices of the metrics used. Lehman suggests using the number of "modules" as the best way to measure the size of a large software system [LPR+97]. On the contrary, in [GT00] W. Godfrey and Q. Tu use the number of uncommented lines of code. We decide to use:

- The number of revisions: Since a revision represents a new version of a file, it can be viewed as a new file. Thus, from this point of view, our approach meets Lehman's suggestions.

- The number of lines of code: It gives a potentially more detailed information than the number of revisions. Studying the evolution of Mozilla using the *nor* metric only would mean losing some of its history. Moreover, using the *loc* metric gives us the possibility to compare our case-study with another Open Source example: The Linux Kernel [GT00].

- The number of bugs: It represents a new dimension to inspect.

We consider the growth of the system with respect to the time dimension, rather than the version number (as suggested by Lehman *et al.* in [LPR98]). The time dimension is more suitable for Open Source software and, in fact, it is also used by W. Godfrey and Q. Tu in the Linux Kernel evolution analysis ([GT00]). The Open Source development does not have the industrial constraints characterizing the commercial software. As a consequence, the relation between version number and time is often difficult to understand.

6.1.1 Number of Revisions Growth

Figure 6.1 shows two superimpositions of the SYSTEM REVISIONS GROWTH VIEW. In the black one what we call the system is the set of all the ".cpp" products, while in the red one the system is composed of all the ".h" products. Both refer to the Mozilla project.

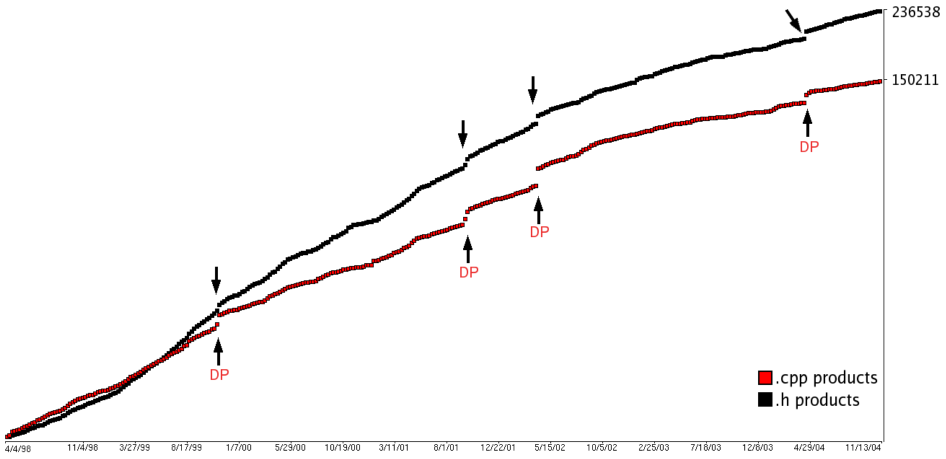


Figure 6.1: The System Revisions Growth View applied to Mozilla. As metric we use the number of days for the horizontal position and the number of revisions for the vertical position.

First of all, we observe that the evolution of Mozilla is aligned with Lehman and Turski's inverse square growth rate hypothesis [LPR98, Tur96], that is: *"As the system grew, the rate of growth would slow, with the system growth approximating an inverse square curve"*.
Our statistical analysis shows that the growth rate of the *nor* metric over time fits well into an inverse square model. If t is the time expressed in number of days since the first version was released, then the following functions are good models of the growth:

$$nor(t) = \quad -11752 + 720 \sqrt[0.7]{t} \qquad \text{Products .h} \qquad (6.1)$$
$$nor(t) = \quad 10^2\big(- 283.75 + 11.86 \sqrt[0.7]{t}\big) \qquad \text{Products .cpp} \qquad (6.2)$$

where $t \in T = \{0, .., t_{\max}\}$.
The relative errors, given by the equation 6.3, are 0.09 for model 6.1 and 0.2 for model 6.2[1].

$$\text{Relative error} = \frac{\sum_{t_i \in T} \left| \dfrac{nor(t_i) - \widehat{nor}(t_i)}{nor(t_i)} \right|}{|T|} \qquad (6.3)$$

Continuing the inspection of the view, we notice that there are four dis-

[1] A better model for the .cpp products growth is given by the equation:

$$nor(t) = 10^4\big(- 72.28 + 110.42 \sqrt[0.2]{t} - 60.16 \sqrt[0.4]{t} + 13.73 \sqrt[0.6]{t} - 1.07 \sqrt[0.8]{t}\big)$$

which is anyway sub-linear. The relative error for this model is equal to 0.07

continuity points[2] (marked as DP) for each curve (with the same horizontal positions) and zero dead phase. The latter is a good symptom, since it implies that the system was always living. We claim that Mozilla obeys to the *Continuing Change Lehman Law*[LB85], that is:
"Software systems must be continually adapted else they become progressively less satisfactory".
Regarding the discontinuity points, we need to perform a following inspection to understand the reasons of their presence. However, since a complete analysis would take a lot of time and it goes further from the purposes of the case-study, we outline only how the inspection can be done. First of all, we have to apply the MODULE REVISIONS GROWTH VIEW to all the modules. Then, analyzing this set of views, we can figure out which modules contain the discontinuity points under inspection as well. We finally apply the DIRECTORY REVISION GROWTH VIEW to all modules containing the discontinuity points, and select the directories as we did with modules. In this way we can locate the discontinuities. The method described in the second part of the case-study (Section 6.2) could then be used to understand the causes of the presence of the discontinuity points.

If we assign to the ".cpp" products the meaning of implementation and to the ".h" products the meaning of functionality, then we can state that:

- The two curves have a similar shape, suggesting that the ".cpp" files were always modified in the context of their ".h" counterparts. Our conclusion is that the functionality and the implementation changed together.

- In a first period the ".h" curve is over the ".cpp" one, while from a certain point onwards not only they exchange each other, but also the gap between them grows at constant rate. We conclude that at the beginning of the system the functionality was dominant, while in the remaining part of the system life the implementation changed faster.

6.1.2 Number of Lines of Code Growth

Figure 6.2 shows two superimpositions of the SYSTEM LINES GROWTH VIEW, where the colors meaning are the same as in Figure 6.1.

Also from the *loc* point of view, the evolution of Mozilla follows an inverse square model. The models of the growth are given by the following equations:

$$loc(t) = 10^3\big(-78.5 + 17.4\sqrt{t}\big) \qquad \text{Products .h} \tag{6.4}$$

$$loc(t) = 10^4\big(-30 + 5.8\sqrt{t}\big) \qquad \text{Products .cpp} \tag{6.5}$$

[2]The discontinuity points are related to the following time stamp: 11/6/1999, 9/22/2001, 4/6/2002, 4/17/2004.

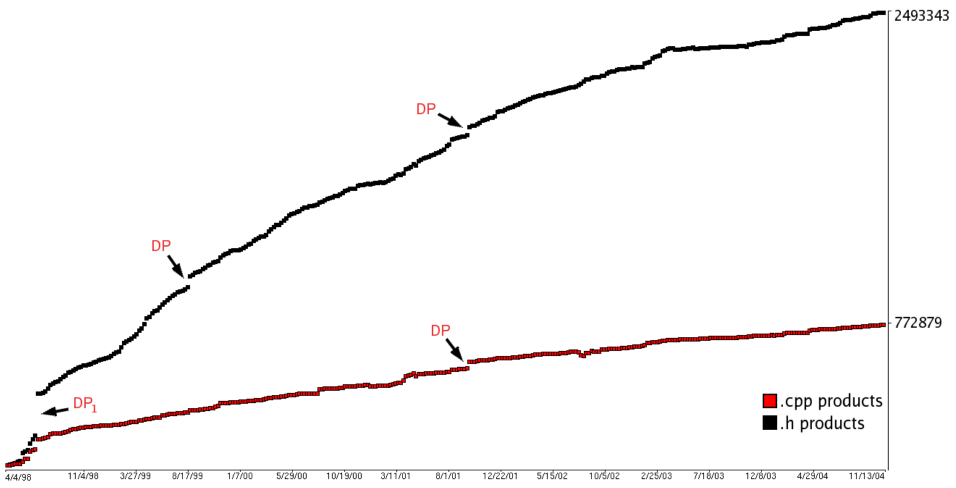


Figure 6.2: The System Lines Growth View applied to Mozilla. As metric we use the number of days for the horizontal position and the number of lines of code for the vertical position.

The relative errors (given by equation 6.3) are 0.08 for model 6.4 and 0.15 for model 6.5.
These results are in contradiction with the Linux Kernel case-study. In [GT00] it is shown that the Linux Kernel growth, in terms of *loc*, is super-linear.

The ".cpp" file sizes increase much faster than the ".h" ones. This confirms that the implementation changed faster than the functionality (see Section 6.1.1).

Looking at Figure 6.2 we notice the existence of five discontinuity points[3]. One of them only corresponds to a discontinuity point found in Figure 6.1. The point marked as DP$_1$ is related to an addition of 230546 lines of code, which is the 9% of the size of the last version of the system. Using the methodology described in the previous Section (6.1.1) we found two commits having a number of lines added greater than 10000. The corresponding products (which are **lib/libnet/mkimap4.cpp** and **network/protocol/imap4/mkimap4.cpp**) represent reengineering candidates.

We consider the flatness of the red curve to be a good sign, as it suggests that new functionalities were not being indiscriminately added. The same situation was found in the Linux Kernel [GT00].

The final observation, generalizable to all the ".cpp"-".h" curves pair, concerns the angle between them. If it is roughly constant, as it is for Mozilla, then we postulate that the methods length is on average constant.

[3] The discontinuity points are related to the following time stamp: 6/20/1998 (black and red curves), 8/14/1999 (black curve) and 9/22/2001 (black and red curves).

On the contrary, an increasing or decreasing angle corresponds to an increasing or decreasing methods length respectively.

6.1.3 Number of Bugs Growth

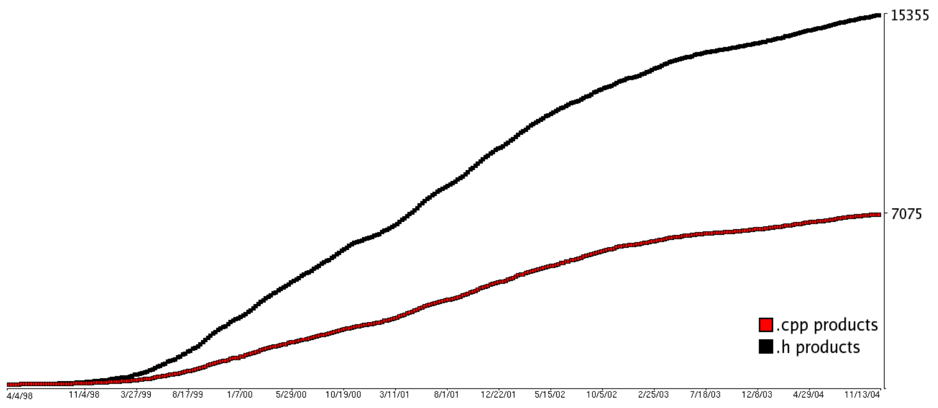


Figure 6.3: The System Bugs Growth View applied to Mozilla. As metric we use the number of days for the horizontal position and the number of bugs for the vertical position.

Figure 6.3 shows two superimpositions of the System Bugs Growth View. The models of the growth are super-linear. They are given by:

$$\begin{aligned} nob(t) &= -0.004 - 0.52t + 0.004t^2 && .h \\ nob(t) &= 211 - 855.2t + 1472t^{1.2} - 997t^{1.4} + 328t^{1.6} - 52t^{1.8} + 3.2t^2 && .cpp \end{aligned}$$

The relative errors (given by the equation 6.3), are 0.1 for the first and 0.06 for the second model.

Summarizing the results obtained, the *nob* growth is super-linear, while both the *nor* and *loc* growth are sub-linear. Some possible reasons for this difference are:

- As the system increases in size and complexity, changes take longer and are more likely to introduce bugs.
- Each bug fixed may introduce more than one bug.

We claim that the number of bugs metric is a new and attractive perspective for studying the evolution of software systems. It would be interesting to compare this result with other case-studies.

Looking at the view, we notice an initial flat phase in both the curves. From our point of view, the presence of this phase is due to:

- The first version of Bugzilla was released in August 1998. Therefore, from April to August 1998 no bug was reported.
- At the beginning Bugzilla was not as known as it is now. It could be possible that some developers did not know its existence and, as a consequence, they did not report bugs using it.

6.1.4 Number of Revisions Trend

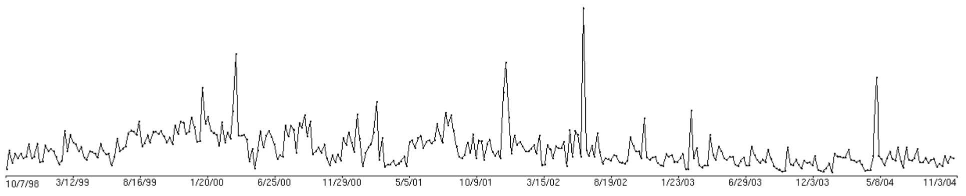


Figure 6.4: The SYSTEM PRODUCTION TREND VIEW applied to the Mozilla system. As metrics we use the number of time intervals for the horizontal position and the number of revisions for the vertical position.

Figure 6.4 shows the view representing the production of revisions for the Mozilla system. The *nor* trend has four peaks, for which the corresponding time stamps coincide with the time stamps of the discontinuity points detected in Figure 6.1.

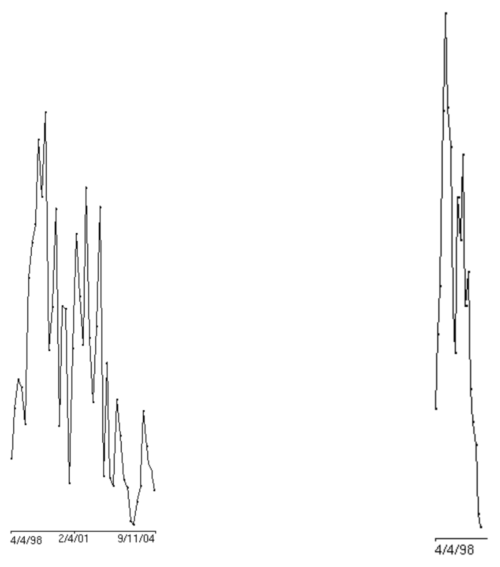
Looking at the view, we find neither recurring patterns nor major phases. In order to detect major phases, we need to use a longer time interval. We consider Figures 6.5(a) and 6.5(b) which are rendered with a time intervals 56 days and 112 days long respectively.

Analyzing these new views we draw the following conclusions:

1. The production of revisions for the Mozilla system is a self-similar process (see Section 4.4.2).
2. The Mozilla development can be divided in three major phase: First the system continuously grew, then there was an oscillating phase and finally the *nor* value tended to decrease, suggesting a stabilization stage.

As a next step in the analysis, we want to gain a numerical idea of the production of revisions. We do so using both phase detection (see Section 4.4.2) and statistical analysis. The results obtained by applying them are reported in Table 6.1[4], split by year.

[4] The year 1998-1999 corresponds to 4/9/1998-1/1/1999, while the year 2004-2005 is related to the pair of dates 1/1/2004-11/13/2004. All the other years are considered from 1/1 to 1/1.



(a) 56 days time interval. (b) 112 days time interval.

Figure 6.5: Two System Production Trend Views with different time intervals applied to Mozilla.

The data in Table 6.1, and in particular the Mean values, confirms the existence of three major phases. The *nor* mean value increased from 1998 to 2000; then from 2000 to 2002 it oscillated and, finally, from 2002 to 2004 it tended to decrease. From the mean values we draw another conclusion: The maximum development of Mozilla was from 1999 to 2002.
Looking at the variance column, we notice that the highest value corresponds to the year 2002-2003, during which the *nor* trend had its highest peak.

The last observation concerns the phases. Each year the number of unstable phases is greater (or, only in one case, equal) than the number of both the stable and the increasing/decreasing stable ones. This means that, given two adjacent weeks w_1 and w_2, in most of the cases the numbers of commits performed during w_1 and w_2 are uncorrelated. From our point of view, this is due to the Open Source development model. The large number of developers and their geographic distribution could make the number of commits a variable quantity. It would be interesting to compare this data to data retrieved from another Open Source software system, in order to either confirm or invalidate our hypothesis.

Year	Phases			Statistic	
	Stab.	**Inc/Dec Stable**	**Unstab.**	**Mean**	**Variance**
1998 - 1999	10	4	25	556.8	34551.2
1999 - 2000	11	8	32	1085.4	252187
2000 - 2001	7	11	33	778.9	173056
2001 - 2002	15	13	24	858.8	277713
2002 - 2003	23	6	23	622	395225
2003 - 2004	16	4	25	451.5	314314
2004 - 2005	19	10	23	363.6	75911.1

Table 6.1: Statistical information regarding the production of revisions for Mozilla.

6.1.5 Number of Bugs Trend

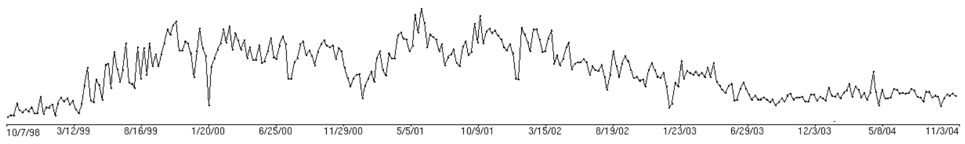


Figure 6.6: The System Production Trend View applied to the Mozilla system. As metrics we use the number of time intervals for the horizontal position and the number of bugs for the vertical position.

Figure 6.6 depicts the view representing the production of bugs for the Mozilla system.
This time the graph has a general trend: First the number of bugs tended to grow, then it remained on average constant (with a lot of oscillations) and, in the end, it decreased. As we did with the revisions graph, we build a couple of new views with longer time intervals[5]. Using them, not only we validate the existence of the major phases detected into the general trend, but also we recognize the production of bugs as a self-similar process (see Section 4.4.2).

Table 6.2 reports the results of both the statistical analysis and the phase detection. Looking at the mean column, we get another confirmation of the presence of the three major phases: from 1998 to 2000 the *nob* mean value grew up to 49.1, then from 2000 to 2002 it varied ($64.6 \Rightarrow 71.8 \Rightarrow 55.9$) and finally from 2002 to 2005 it decreased to 24. From the values of the variance and from the number of unstable phases we draw the same conclusion stated

[5]The views, which are not shown for brevity's sake, have time interval lengths equal to 56 days and 112 days respectively.

	Phases			**Statistic**	
Year	**Stab.**	**Inc/Dec Stable**	**Unstab.**	**Mean**	**Variance**
1998 - 1999	24	0	15	4.71	24.4
1999 - 2000	4	10	38	49.1	650.4
2000 - 2001	7	10	34	64.6	232.1
2001 - 2002	4	10	38	71.8	302.5
2002 - 2003	23	4	25	55.9	300.2
2003 - 2004	18	8	6	28.6	147.1
2004 - 2005	13	8	24	24	36.3

Table 6.2: Statistical information regarding the production of bugs for Mozilla.

for the revisions trend: The Open Source development model makes the numbers of bugs reported in adjacent weeks uncorrelated quantities.

Before concluding, we analyze the superimposition of Figure 6.4 and Figure 6.6, in order to find patterns involving both the revisions and the bugs trend (the superimposition is not depicted for brevity's sake).
Unfortunately our analysis does not produce any results. Neither recurring patterns nor cross-correlations[6] were found.

[6]See Section 4.4.2.

6.2 Understanding the design and the structure of the system

Our methodology does not consist in a set of rules defining which views to apply and on what parts of the subject system. It outlines **a** possible way to combine the views (shown in Figure 6.7), according to what we are looking for.

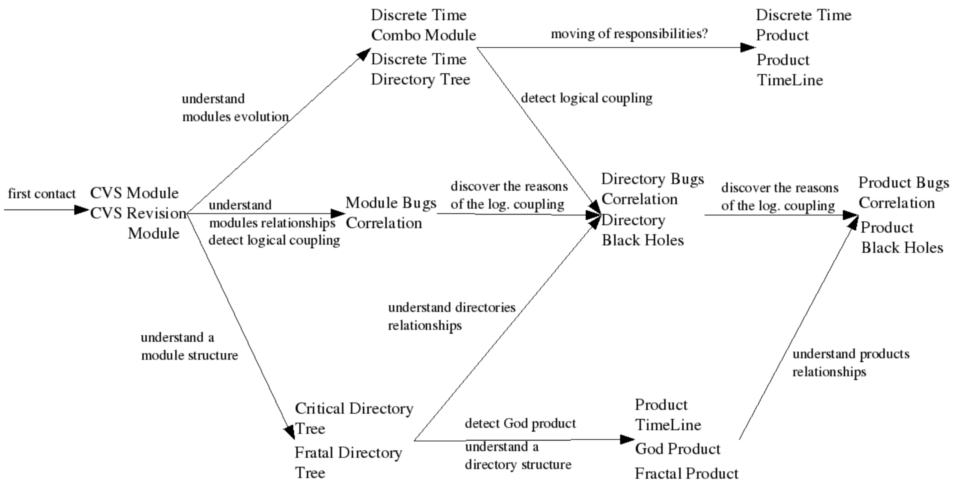

first contact
CVS Module
CVS Revision Module
understand modules evolution
Discrete Time Combo Module
Discrete Time Directory Tree
moving of responsibilities?
Discrete Time Product
Product TimeLine
detect logical coupling
understand modules relationships
detect logical coupling
Module Bugs Correlation
discover the reasons of the log. coupling
Directory Bugs Correlation
Directory Black Holes
discover the reasons of the log. coupling
Product Bugs Correlation
Product Black Holes
understand a module structure
understand directories relationships
understand products relationships
Critical Directory Tree
Fratal Directory Tree
detect God product
understand a directory structure
Product TimeLine
God Product
Fractal Product

Figure 6.7: Our approach to study the design and the structure of a subject system.

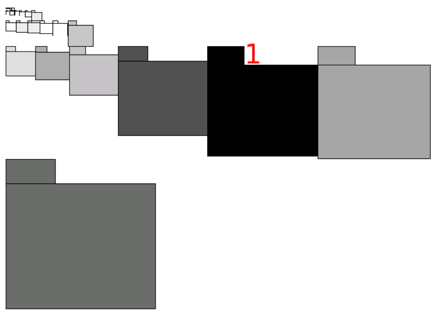
The scheme depicted in Figure 6.7 should be read taking into account that:

- There is no unique or ideal path through the views.
- Different views can be applied at the same stage depending on the current context.
- The decision to use a certain view depends on the results obtained with the currently displayed view.

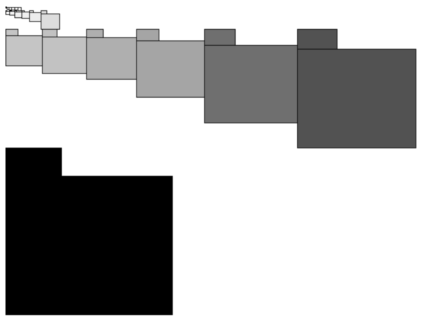
6.2.1 Understanding the overall structure of the system in terms of modules

The purpose of this part of the methodology is to address the *first contact problem* (with respect to Figure 6.7). We want to figure out which are the most important modules, taking into account all the available metrics. From now on, we consider the system as the set of all the ".cpp" products.

To attack a system, a good starting point is represented by the pair CVS Module View and CVS Revision Module View, which are shown in Figure 6.8. They give a first insight of the software, dividing the modules according to their sizes and importance.



(a) The CVS Module View. As metrics we use the number of products for the size and the number of bugs for the color.



(b) The CVS Revision Module View. As metrics we use the number of revisions for the size and the number of bugs for the color.

Figure 6.8: The CVS Module View and CVS Revision Module View applied to Mozilla.

Looking at Figure 6.8(b) we introduce the following empirical law:

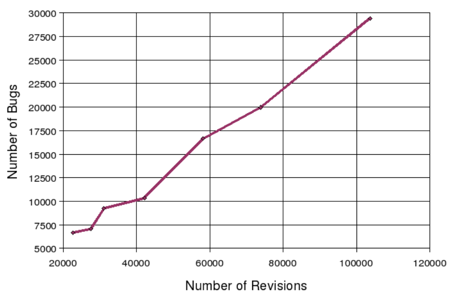
Mozilla Empirical Law 1 *given a module, the higher the number of revisions it contains, the higher the number of bugs affecting it.*

In other words, greater the work performed on a module is, greater the production of bugs is.

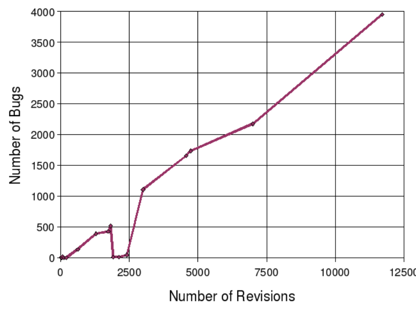
The first criterion we are going to use for reasoning about modules is the number of revisions. The aim is to identify the most important modules in an unambiguous way. We do that by choosing a so called "*threshold module*". It represents the smallest of the big modules. Then, we just look at its number of revisions r and select the modules having $nor > r$. Using the threshold module SeaMonkeyMailNews(nor:22646, nob : 6644) we obtain two sets:

$$\begin{aligned} \text{BigRevMod} &= \{m \in \text{Modules} \mid nor(m) \geq nor(\text{SeaMonkeyMailNews})\} \\ &= \{\text{CalendarClient, RaptorDist, RaptorLayout,} \\ &\quad \text{SeaMonkeyCore,SeaMonkeyEditor, SeaMonkeyLayout,} \\ &\quad \text{SeaMonkeyMailNews}\} \qquad (6.6) \\ \text{SmallRevMod} &= \text{Modules} \setminus \text{BigRevMod} \qquad (6.7) \end{aligned}$$

The distinction is meaningful because while all the elements belonging to BigRevMod obey to Mozilla Empirical Law 1, this does not hold for the modules composing the SmallRevMod set (as shown in Figure 6.9).



(a) The BigRevMod set.



(b) The SmallRevMod set.

Figure 6.9: The *nob* growth with respect to *nor*.

As a second step in identifying the key modules, we look at the CVS Module View depicted in Figure 6.8(a). A first observation suggested by the view and proved by the graph depicted in Figure 6.10, is that the number of bugs does not monotonically increase with the number of products. The same holds even if the Module set is split in two subset using a threshold module.

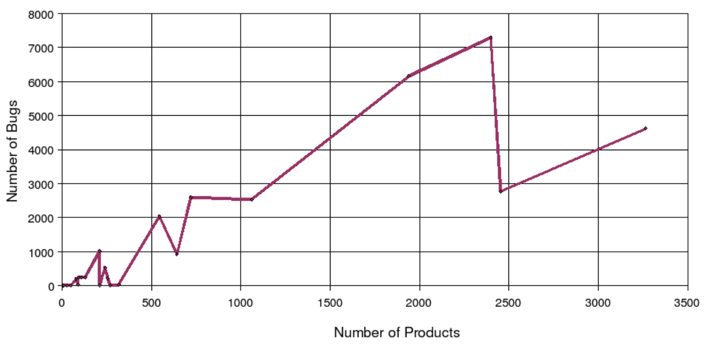


Figure 6.10: The *nob* growth with respect to *nop*.

As a consequence, we need a different method to partition the module set. Considering the ratio $\frac{nob}{nop}$, we can figure out which are the critical modules with respect to *nop* and *nob* criteria. These modules, which compose the

BigPrdMod set, are those with the highest number of bugs per product:

$$\begin{aligned}\text{BigPrdMod} &= \{m \in Modules \mid \frac{nob(m)}{nop(m)} > 10\} \\ &= \{\text{SeaMonkeyEditor (12.8), SeaMonkeyXPToolKit(10.36),} \\ &\quad \text{RaptorLayout(10.3), SeaMonkeyLayout(12.26),} \\ &\quad \text{SeaMonkeyMailNews (12.21)}\} \qquad (6.8)\end{aligned}$$

Up to now we have built two sets (BigRevMod and BigPrdMod) representing the key modules with respect to different criteria. The intersection of them gives the most important modules from both the points of view.

$$\begin{aligned}\text{MIMod} &= \text{BigRevMod} \cap \text{BigPrdMod} \\ &= \{RaptorLayout, SeaMonkeyEditor, SeaMonkeyLayout, \\ &\quad SeaMonkeyMailNews\} \qquad (6.9)\end{aligned}$$

All the results obtained are summarized in Table 6.3[7].

Name	***nop***	***nor***	***nob***	**BigRev Mod**	**BigPrd Mod**
SeaMonkeyLayout	2398	103669	29412	x	x
RaptorLayout	1938	73930	19971	x	x
SeaMonkeyEditor	721	31149	9235	x	x
SeaMonkeyMailNews	544	22646	6644	x	x
SeaMonkeyCore	3266	58134	16652	x	
RaptorDist	2455	42197	10345	x	
CalendarClient	1060	27584	7043	x	
SeaMonkeyXPToolKit	210	6997	2175		x

Table 6.3: The Mozilla modules classification.

6.2.2 Detecting design shortcomings using entities

Since we are not interesting in performing a complete and exhaustive analysis of Mozilla, we focus our attention on the SeaMonkeyLayout module only, which has the highest values of both *nor* and *nob* into the MIMod set.

First of all, we are interested in understanding the internal structure of the module and detecting the most important hierarchies. To do so, we make use of the CRITICAL DIRECTORY TREE VIEW, shown in Figure 6.11.

[7] The numbers of bugs in Table 6.3 are apparently in contradiction with the data shown in Figure 6.3 (in which the total number of bugs is 15355). The reason is that in Table 6.3 we consider the number of bug references. In Figure 6.3 a bug affecting two different products is considered as one bug, while in Table 6.3 it is considered as two bug references.

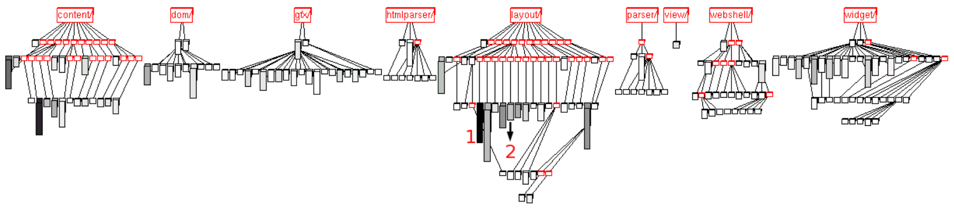


Figure 6.11: The Critical Directory Tree View applied to the Sea-MonkeyLayout module of Mozilla. As metrics we use the number of products for the figure height and the number of bugs for the figure color.

The module is composed of nine hierarchies, where **content** and **layout** seem to be the most critical. Moreover, we notice the structural similarity between the **parser** and the **htmlparser** hierarchies.
The next step in the analysis consists in inspecting the products belonging to the critical hierarchies. We focus on the **layout** hierarchy using a God Product View (depicted in Figure 6.12)

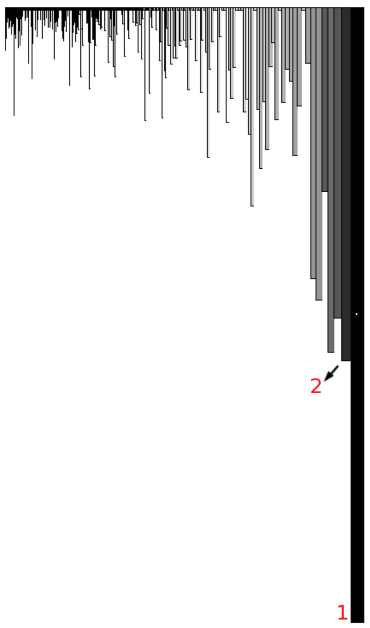
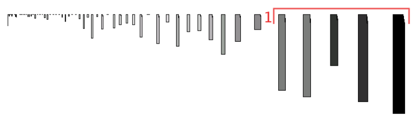


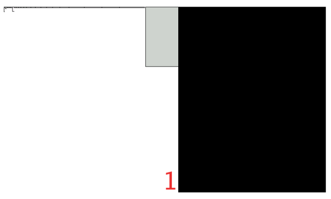
Figure 6.12: The God Product View applied to the **layout** hierarchy of Mozilla. As metrics we use the number of revisions for the width, the number of lines of code for the height and the number of bugs for the color.

Looking at the view, we find that the **layout** hierarchy contains huge products affected by a large number of bugs. As a consequence, this hierar-

chy is a reengineering candidate.
In particular the products marked as 1 (**nsCSSFrameConstructor.cpp**) and 2 (**nsPresShell.cpp**) in Figure 6.12 are likely to be God products, since they have 13683 and 7835 lines, 453 and 380 bugs respectively.
To better understand which roles these products have, we compare them with the products belonging to their container directories. The product **nsCSSFrameConstructor.cpp**, which is the most critical, is not contained in the most critical directory (marked as 1 in Figure 6.11), but it belongs to **layout/html/style/src** (marked as 2 in Figure 6.11). The product **nsPresShell.cpp** is contained in the directory **layout/html/base/src** (marked as 1 in Figure 6.11). Figure 6.13 shows the God Product Views applied to both the directories.



(a) The **layout/html/base/src** directory.



(b) The **layout/html/style/src** directory.

Figure 6.13: The God Product View applied to the directories **layout/html/base/src** and **layout/html/style/src**. As metrics we use the number of revisions for the width, the number of lines of code for the height and the number of bugs for the color.

Figure 6.13(b) confirms that **nsCSSFrameConstructor.cpp** is a God product. Its *nob* and *loc* metric measurements are huge with respect to those characterizing the other products belonging to the **layout/html/style/src** directory. Such hypothesis is also validated by the Product TimeLine View, depicted in Figure 6.14. As time goes by, the products belonging to the directory **layout/html/style/src** tend to die, giving their responsibilities to **nsCSSFrameConstructor.cpp** (marked as 1).

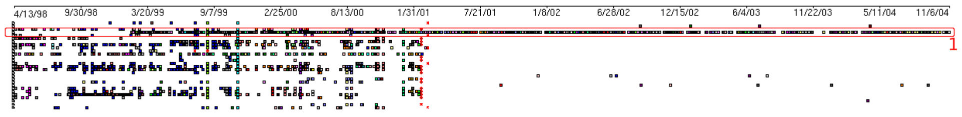


Figure 6.14: The Product TimeLine View applied to the **layout/html/style/src** directory.

The directory **layout/html/base/src** is more balanced. There are five products (marked as 1 in Figure 6.13(a)) which are likely to have a fundamental role, since their *nob* and *loc* values are large. This directory is a candidate for the starting point of a reengineering process.

6.2.3 Detecting design shortcomings using relationships

With the previous set of views, we have seen how, starting from a module, it is possible to detect design shortcomings. These can be found at any granularity level: Hierarchy, directory and single product. Up to now we have used only the information concerning the entities, without considering their relationships.
Now we want to understand which correlations hold among the directories belonging to the module under analysis. The Directory Bugs Correlation View, depicted in Figure 6.15, shows these dependencies, from the bugs shared point of view. In order to make the view easy to read, only the edges having a number of shared bugs greater than 100 are displayed. The standalone directories are also removed.

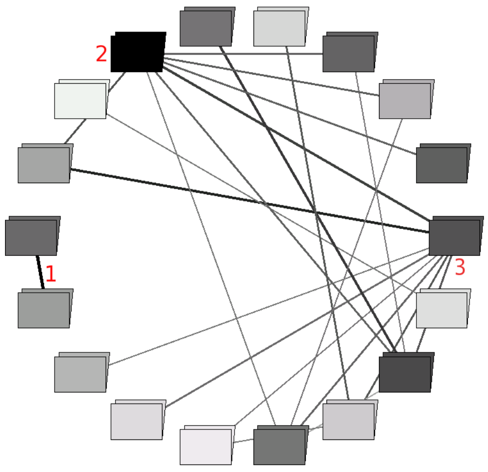


Figure 6.15: The Directory Bugs Correlation View applied to the SeaMonkeyLayout module of Mozilla. As metrics we use the number of revisions for the figure color and the number of shared bugs for both the edge color and width.

The directories marked as 2 (**layout/html/base/src**) and 3 (**content-/base/src**) are those which share bugs the most. This implies that they own a central role in the parent module context. The directory **content/base/src** represents a new reengineering candidate, while for the directory **layout/html/base/src** the need of a new design is confirmed (it is detected as reengineering candidate in the previous Section).

We focus our attention on the correlation marked as 1, which is the strongest in the view. This dependency holds between the directories **parser-/htmlparser/src/** and **htmlparser/src/**. For the hierarchies **htmlparser** and **parser** we have just noticed a structural similarity in the CRITICAL DIRECTORY TREE VIEW (Figure 6.11). We want to inspect whether this similarity is also present in the temporal dimension or not. In other words, we want to check whether the entity developments are somehow correlated or not. We do that using the DISCRETE TIME COMBO DIRECTORY TREE VIEW.

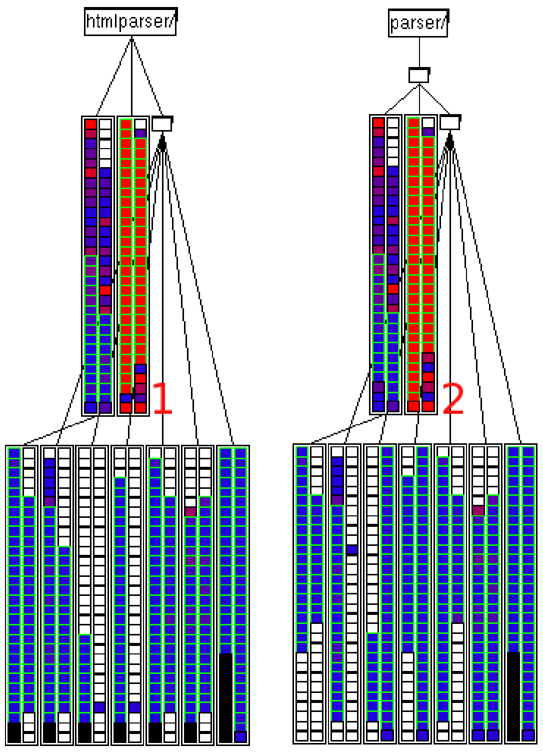
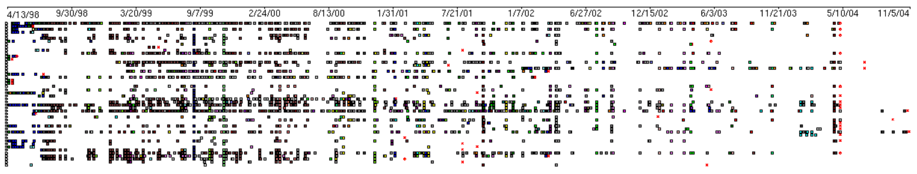


Figure 6.16: The DISCRETE TIME COMBO DIRECTORY TREE VIEW applied to the **htmlparser** and **parser** hierarchies of Mozilla.

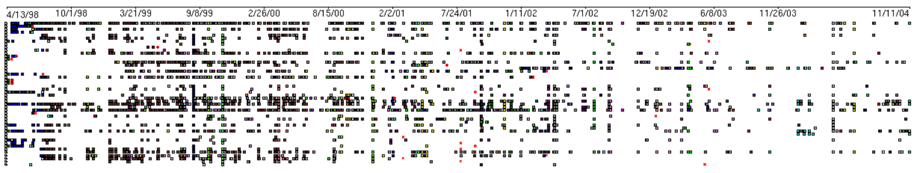
Figure 6.16 shows that the two hierarchies are similar from both the production of revisions and bugs points of view. Then, since the time dimension analysis seems to be the right way, we apply to the "hot directories"[8] (**htmlparser/src** marked as 1 and **parser/htmlparser/src** marked as 2) the PRODUCT TIMELINE VIEW.

Performing a cross-inspection of Figures 6.17(a) and 6.17(b) we find that the directories contain exactly the same products. Furthermore, up to

[8] The concept of "hot directories" is explained in Section 5.2.2.



(a) The **parser/htmlparser/src** directory.



(b) The **htmlparser/src** directory.

Figure 6.17: The Product TimeLine View applied to the directories **parser/htmlparser/src** and **htmlparser/src**.

April 18, 2004, all the commits of corresponding products were performed simultaneously. Then, from this date onwards, the products belonging to the **parser/htmlparser/src** directory tended to die.
Before drawing any conclusion, it would be better to focus on one product only (**nsParserModule.cpp** shown in Figure 6.18). In this way, we are in the position to analyze the situation at fine granularity.

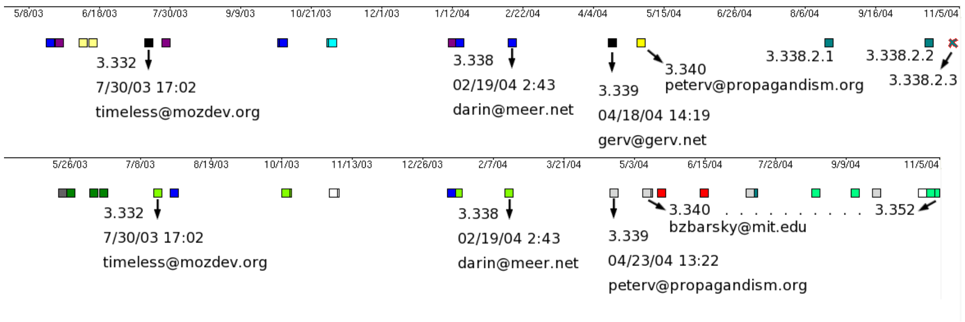


Figure 6.18: The Product TimeLine Views applied to both the versions of the **nsParserModule.cpp** product of Mozilla.

Up to the revision 3.338 the commits are identical from the revision number, author and time stamp points of view. Then, for the revision 3.339 both the author and the time stamp are different.

From this point onwards, the two products behave differently. In detail, the **parser/htmlparser/src/nsParserModule.cpp** file takes the branch 3.338.2.1 and, in the end, die (recognizable by the presence of the cross-shape figure). On the contrary, the **htmlparser/src/nsParserModule.cpp** product follows the main trunk until the revision 3.352.

Such situation does not represent a directory renaming, from **parser-/htmlparser/src** to **htmlparser/src**. In fact, in this case the directory **htmlparser/src** should have been created after the renaming, that is after the death of **parser/htmlparser/src**[9]. On the contrary, both the directories have existed since the beginning of the system.

We conclude that there was a duplicated part of the system, which was finally removed. The strong correlation found in Figure 6.15 is due to a directory duplication, that represents a bad design symptom.

6.2.4 Detecting logical couplings

Another design shortcoming is represented by logical couplings between modules, directories or products. Before concluding the presentation of the methodology, we would describe how our approach can uncover logical couplings not only between modules, but also between directories and products belonging to different modules. Such dependencies are much more significant than the logical couplings between directories belonging to the same module or products belonging to the same directory, because different parts of the system are involved.

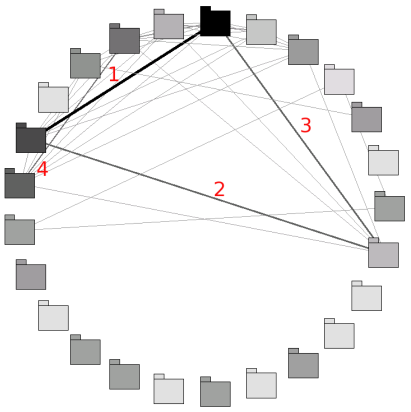
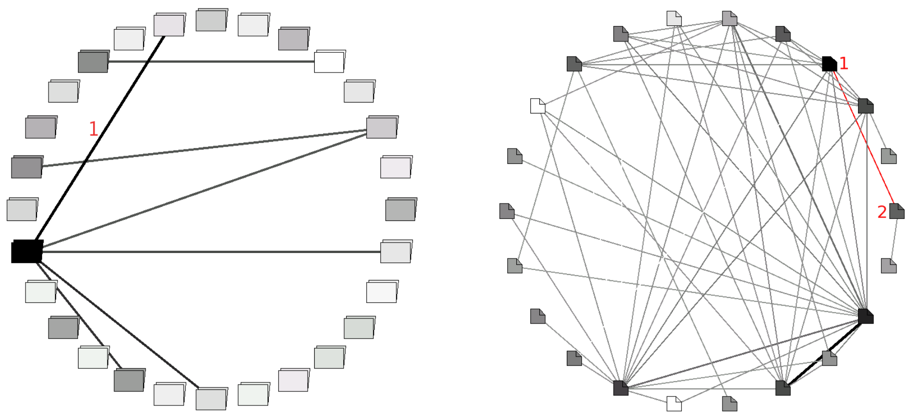


Figure 6.19: The Module Bugs Correlation View applied to the Mozilla system. As metrics we use the number of revisions for the figure color and the number of shared bugs for both the edge color and width.

[9] We can recognize the birth of a product by the presence of the so called *product first revision figure* (see Section 5.3.2).

The starting point of this analysis is the Module Bugs Correlation View, which is depicted in Figure 6.19. As we have seen in Section 4.5.3, the strongest dependency, after filtering the false correlations, is that between the CalendarClient and SeaMonkeyLayout modules (marked as 3). Once the logical coupling between modules is detected, we have to decrease the analysis granularity to the directory level. We do so using a modified version of the Directory Bugs Correlation View. It displays only the edges linking directories belonging to different modules. The view is shown in Figure 6.20(a).



(a) The Calendar and SeaMonkeyLayout modules.

(b) The **view/src/** and **layout/html/base/src** directories.

Figure 6.20: The Directory Bugs Correlation View and the Product Bugs Correlation Views. As metrics we use the number of revisions for the figure color and the number of shared bugs for both the edge color and width.

All the edges in Figure 6.20(a) are likely to represent design shortcomings, since they involve directories belonging to different parts of the system. These directories are reengineering candidates.
We continue the analysis focusing on a pair of directories, in order to uncover correlations between products also. We choose the strongest dependency, that is marked as 1 in Figure 6.20(a). It links the directories **layout/html/base/src** and **view/src/** with a number of shared bugs equal to 123.

Figure 6.20(b) depicts the Product Bugs Correlation View applied to the directories mentioned above.
By construction, the view has the following property: Given two products p_1 and p_2, if p_1 and p_2 belong to different directories, then they belong to different modules too.

With respect to Figure 6.20(b), the edge colored in red is the only correlation holding between products belonging to different directories. Such products are: **layout/html/base/src/nsPresShell.cpp** and **view/src/nsView-Manager.cpp**. As a consequence, the logical coupling marked as 1 in Figure 6.20(a) is entirely mapped to the dependency between **nsPresShell.cpp** and **nsViewManager.cpp**. These two products replace the directories **lay-out/html/base/src** and **view/src/** as reengineering candidates.

We conclude that using the PRODUCT BUGS CORRELATION VIEW, we can filter the coarse-grained information gained analyzing the DIRECTORY BUGS CORRELATION VIEW, focusing on more precise targets.

6.3 The Mozilla Project Evolution: Relations discovered

One of the features of our tool consists in its capability to design and apply "on-the-fly" new polymetric views.
During the analysis of Mozilla, we hypothesized that some relations between entity metrics could hold. In this Section we empirically verify the validity of these relations. To do so, we use a cluster of views designed ad-hoc for either confirming or invalidating the supposed relations. All of them share the following characteristics:

- They make use of the correlation layout, in which the two position (vertical and horizontal) are mapped on metric measurements.
- They use neither the size nor the color for mapping metric measurements.
- They use only rectangle figures.
- For all of them figures represent products.

6.3.1 Number of Bugs - Fractal Value

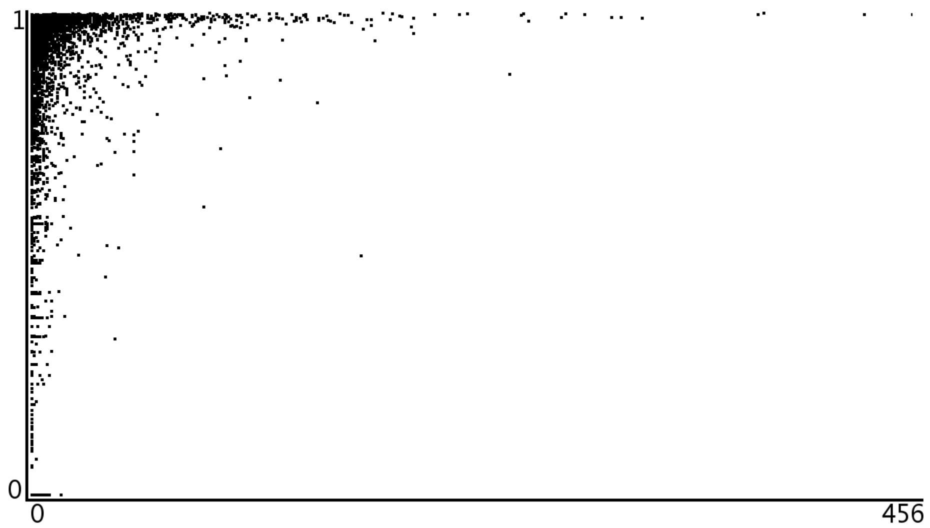


Figure 6.21: The Number of Bugs - Fractal Value Correlation View. As metrics we use the number of bugs for the horizontal position and the Fractal Value for the vertical position.

With respect to Figure 6.21, the horizontal position represents the number of bugs, while the vertical position is related to the Fractal Value[10].

Taking into account that, given a product:

- If it was developed by one author only then the Fractal Value is equal to 0.
- The Fractal Value $\rightarrow 1$ if the development was equally distributed among a lot of authors.

We postulate the following empirical law:

Mozilla Empirical Law 2 *the more the development of a product is distributed among different authors, the greater the number of bugs affecting it is.*

6.3.2 Number of Bugs - Number of fellows

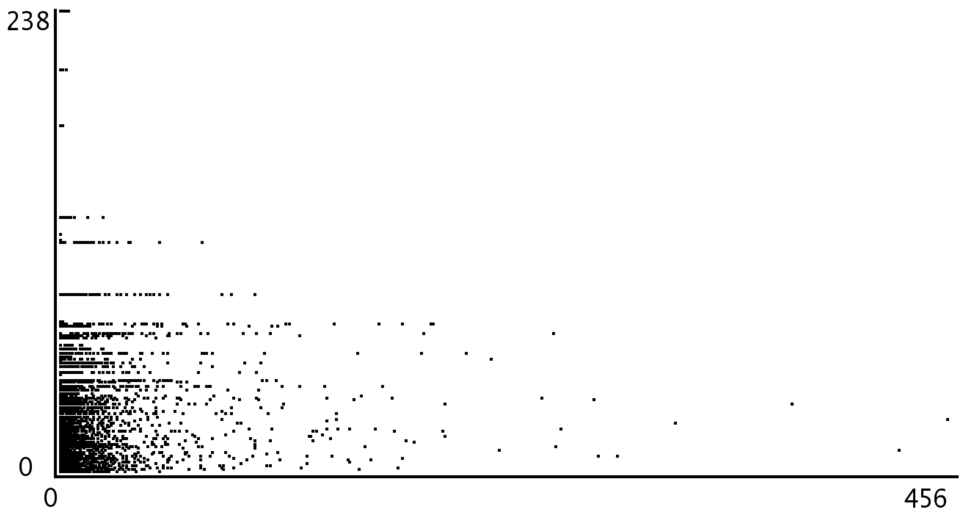


Figure 6.22: The Number of Bugs - Number of Fellows Correlation View. As metrics we use the number of bugs for the horizontal position and the number of Fellows for the vertical position.

For each rectangle contained in Figure 6.22, the horizontal position is related to the number of bugs, while the vertical position is proportional to the number of fellows.

Given two products p_1 and p_2, we define that p_1 is a fellow of p_2 if they belong to the same directory. Therefore, the number of fellows is equal to

[10]The concept of Fractal Value is presented in Section 4.3.2.

the directory size, in terms of number of products. Looking at Figure 6.22, we introduce the following empirical law:

Mozilla Empirical Law 3 *given a product p which belongs to a directory d, the greater the number of products belonging to d is, the lower the number of bugs affecting p is.*

6.3.3 Number of Bugs - Average Growth rate

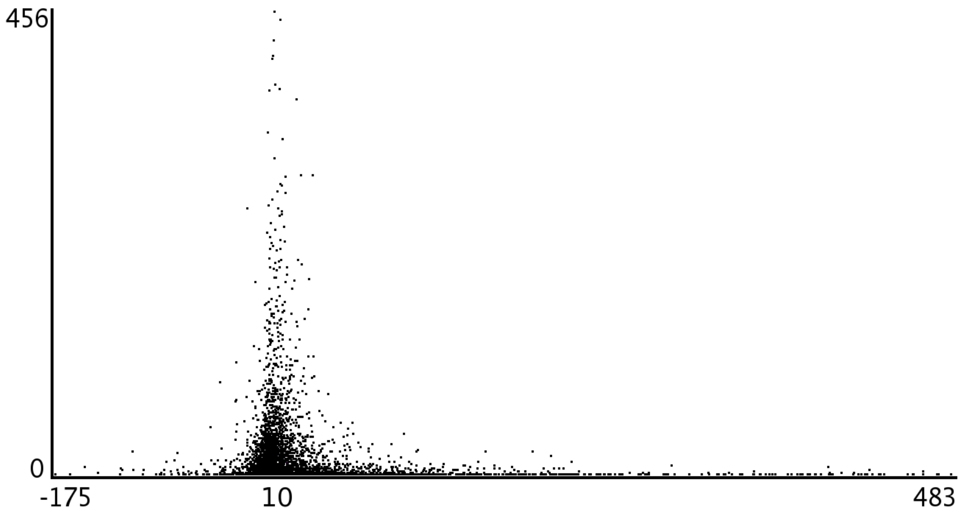


Figure 6.23: The Number of Bugs - Average Growth Rate Correlation View. As metrics we use the number of bugs for the vertical position and the Average Growth Rate for the horizontal position.

Concerning Figure 6.23, the vertical position represents the number of bugs, while the horizontal position is proportional to the Average Growth Rate.
Given a product p composed of a set of revisions $R = \{r_i\}$, the Average Growth Rate of p is defined as:

$$\text{Av. Growth Rate}(p) = \frac{\sum_{r_i \in R} \text{Lines Add}(r_i) - \sum_{r_i \in R} \text{Lines Rem}(r_i)}{|R|} \tag{6.10}$$

Looking at Figure 6.23, we postulate that:

Mozilla Empirical Law 4 *given a product, the more its Average Growth Rate approximates the value 10, the greater the number of bugs affecting the product is.*

6.4 Related Work

Lehman *et al.* have built the largest and best known body of research on the evolution of large, long-lived software systems [LB85, LPR$^+$97, LPR98, Tur96]. Lehman's laws of software evolution [LPR$^+$97], which are based on his case-studies of several large software systems, suggest that as systems grow in size, it becomes increasingly difficult to add new code unless explicit steps are taken to reorganize the overall design. Turski's statistical analysis of these case-studies suggests that system growth (measured in terms of numbers of source modules and number of modules changed) is usually sub-linear, slowing down as the system gets larger and more complex [LPR98, Tur96].

In [GT00] Godfrey and Tu examine the growth of Linux [Lin] over its six year lifespan using several metrics. They find that, at the system level, its growth is super-linear which is a violation of Lehman's fifth law.

Mozilla has been already addressed by Mockus *et al.* in a case-study about Open Source Software Projects [MFH02]. However, in contrast to our work, they focus on the overall community and development process. Their goal consists in comparing the development process of both open source (or free) and non-free software. They are interested in understanding how the work is distributed over the developers, while we focus on the time dimension.

6.5 Conclusion

In the previous two Chapters we have addressed the problems of studying a large software system "in the Large" and "in the Small".
Now, what we need is a way to combine the two approaches in order to perform a complete analysis of the system, from the highest to the lowest levels of abstraction.

6.5.1 Summary

In this Chapter we have presented a complete top-down methodology which leads the entire analysis of a subject software system. The proposed approach consists in a set of guidelines suggesting how to combine fine-grained and coarse-grained polymetric views.

The explanation of the methodology has been done together with the analysis of a case-study (Mozilla). In this way, we have had the possibility to evaluate how the approach works in practice.

First of all, we have studied the evolution of Mozilla. We have compared it with both the existing software evolution theory [Tur96, Leh96, LPR$^+$97, LPR98] and with another Open Source system [GT00] (the Linux Kernel). In this context, we have introduced a new evolutionary perspective: the number of bugs growth point of view, finding that it is super-linear for the Mozilla system.

We have then shown how to detect hidden dependencies among modules, directories and products. The last feature of our methodology consists in the detection of design shortcomings and reengineering candidates.

In the end, we have postulated four empirical laws to which Mozilla seems to obey.

Benefits

The main benefits of our approach are the following:

- *Completeness.* Using our methodology, we can analyze both the evolution and the structure of a system. Furthermore, all the possible granularity levels are considered in the structural inspection.
- *Customizability.* The proposed methodology has the capability to adapt itself to the system under analysis. In fact, as we have seen in Section 6.3, it can be enriched with new polymetric views designed ad-hoc for the subject system.
- *Simplicity.* We have seen how simple it is to find out design shortcomings in a large software system like Mozilla.
- *Interactivity.* We have always the opportunity to interact with the figures composing a view. We can not only inspect and query them, but also we can apply on the figures new polymetric views "on the fly".

Limits

Our approach is limited in the following way:

- *Computation time.* It is a limit of the coarse-grained analysis (see Section 4.7.3).
- *Choice of the view.* It is not always clear which is the most effective view to apply.
- *Need of a validation phase.* In most of the cases, we are not able to draw conclusions, but we can only formulate hypothesis. We need to perform a following validation phase (for example code reading).

Chapter 7

Conclusion

In this Chapter we summarize the contributions made in this thesis, discuss the benefits of our approach, and point to directions for future work.

7.1 Contributions

In this thesis we have presented a new approach to study both the structure and the evolution of large software systems. We claim that our method is complete, in the sense that it covers all the stages required to analyze a software project: From the initial collecting of information to the data presentation and interpretation. In detail, the proposed approach is composed of the following phases:

1. **The population of a release history database.** It combines the version information with bug tracking report data. The information is retrieved from the data sources (CVS repository and Bugzilla), parsed and finally stored in a database. The release history database contains the history of the system and therefore it represents the starting point for the evolution analysis. The main benefits provided are the following:
   - Scattered information, like that presents into the CVS log files, is aggregated in a well defined structure.
   - The populating process has the capability of coupling two data sources (CVS and Bugzilla) having a weak relationship (the informal link in the CVS description field). This is important because bug reports tell us a lot of the history of a system.
   - The populating process is completely automatic.
2. **The coarse-grained and the fine-grained analysis.** We have discussed four clusters of polymetric views, aimed at addressing the problem of the system understanding from both the structural and the

evolutionary perspectives. The presented views make us able to study a software system at every level of abstraction. The coarse-grained analysis is focused on the entire system, while the scope of the fine-grained is the internal structure of a module. The two approaches provide the following benefits:

- *Reduction of complexity.* We can understand the structure of huge and complex software projects without having to read source code at all.
- *Scalability.* Each polymetric view is able to give a great amount of information in a condensed way. Moreover, all the proposed views scale up to large software systems, *e.g.*, more than 2 *Mloc*.
- *Applicability.* Both the approaches are language independent. They are applicable to not only any programming language, but also most of the software paradigms.
- *Use of bugs as cross entities.* Analyzing how the bugs are shared by different modules, directories and products makes it possible to uncover otherwise hidden dependencies.
- *Customizability.* The polymetric views are highly customizable by changing the metrics, the layouts, the figures and the figure parameters. These changes are important, because every software system has its particularities to which the view must be adapted to.
- *Different points of view.* The entities are shown from different points of view. In this way, we can either confirm or invalidate an hypothesis drawn with a view, using another view.
- *Reengineering candidates detection.* It makes our approach useful in a reengineering context.

3. **The top-down methodology.** It consists in a set of guidelines suggesting how to combine fine-grained and coarse-grained polymetric views, in order to lead the entire analysis of a software system. The proposed methodology has been explained together with the analysis of the Mozilla case-study. The main benefits provided are the following:

   - *Completeness.* We can study both the evolution and the structure of a system. Furthermore, all the possible granularity levels are considered in the structural inspection.
   - *Customizability.* Our methodology has the capability to adapt itself to the system under analysis. It can be enriched with new polymetric views designed ad-hoc for the subject system.

- *Simplicity.* The proposed approach makes it easy to find out design shortcomings into large software systems.
- *Interactivity.* We have always the opportunity to interact with the figures composing a view. We can not only inspect and query them, but also we can apply on them new polymetric views "on the fly".

7.2 Future Work

In the future we plan to investigate the following ideas:

- *Generalize the Release History Database.*
  When we designed the Rhdb structure and construction method, we had in mind how we intended to use it. However, we believe that such an instrument could be useful for different evolutionary approaches. Thus, our future work will be concentrated on yielding the Rhdb as general as possible, minimizing the coupling with our application. This should be done by removing the Rhdb simplification assumptions; in detail:

  – Develop an algorithm to detect merge points (like the one proposed by Gall *et al.* in [FPG03b]).
  – Develop a porting for Subversion [Sub] which is replacing CVS.
  – Develop bridges for other version control systems like ClearCase [Cle] and SourceSafe [Sou].
  – Consider the entire CVS repository. Having all the revisions available at the same time, that is the entire CVS repository[1], could open new evolution perspectives. We are interested in exploring this new direction, keeping in mind the database "size problem".

- *Computational efficiency.*
  We could use caching mechanism as a possible solution for decreasing the time needed to render a view from the second time on.

- *New entities and relationships.*
  We would introduce new entities which do not have an equivalent in the software, such as:

  – Bug clusters sharing a particular property.
  – As proposed by Gall *et al.* [FPG03a], software features.

  In order to use these entities to generate new polymetric views, it should be possible to design and create them "on-the-fly".

[1]The entire CVS repository can be downloaded with `rsync`.

- *Source code analysis.*
  We would improve our approach by providing a set of parsers able to analyze source code written with the most common programming languages. In this way, we can introduce new entities like classes, methods, attributes and new relationships like inheritance and use (in an Object Oriented context).

- *Quantitative hypothesis reliability.*
  During the analysis of a view, we formulate some hypothesis. With the introduction of a "score" we can get a quantitative idea of the hypothesis reliability. For example, a movement of responsibilities can be detected by the presence of different patterns into different views. By assigning a score to each pattern, we can compute the reliability of the hypothesis. It is given by the sum of the pattern scores.

- *Automatic patterns detection.*
  Given a pattern, if we define its formal definition, then we can develop an algorithm able to automatically detect it. We believe that the most complex task of this approach will be the formal definitions of patterns.

- *Perform other case-studies.*
  The analysis of other software systems gives us the possibility to:

  - Evaluate our methodology on other systems.
  - Check whether the Mozilla empirical laws still be valid for other software systems or not.
  - Taking into account that the growth of Mozilla is sub-linear, while the growth of the Linux Kernel is super-linear [GT00], it would be interesting to study the model of the growth of other Open Source software systems.

7.3 Epilogue

In this thesis we have proposed an approach to tackle the problem of **understanding the evolution of software systems**, based on software metrics and software visualization. We claim that its main benefit consists in combining the formalism of the former with the simplicity and interactivity of the latter.

The wide applicability of the approach (there are hundreds of Open Source software projects developed using CVS as versioning system and Bugzilla as bug tracking system) will allow us not only to study many software systems, but also to extend, improve and specialize our technique on the basis of the gained experience.

Appendix A

Formal Definitions of Figures

This Appendix provides some formal definitions concerning the figures introduced in this thesis.

A.1 Discrete Time Figure

We present the formal definition of the Discrete Time Figure Phases introduced in Section 4.3.1.

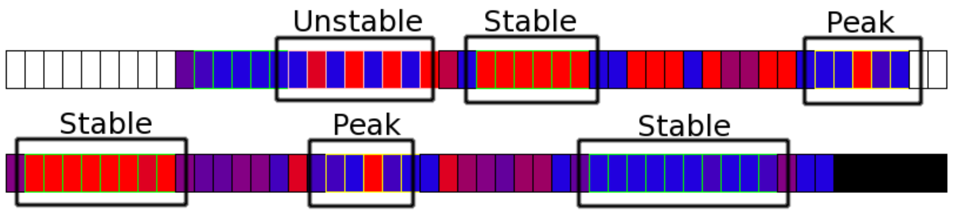


Figure A.1: A Discrete Time Figure with Phases

A.1.1 Stable Phase

Let $R = \{r_1, r_2, \ldots, r_n\}$ be the set of rectangles composing a Discrete Time Figure like one of the two shown in Figure A.1. We define $S = \{r_1, r_2, \ldots, r_s\}$ as a Stable Phase if the following holds:

$$|S| \geq 6 \wedge \Big(\forall i \in \mathbb{N} : 1 < i \leq |S| \Rightarrow (r_i = \text{next}(r_{i-1}) \wedge \big|\text{Red}(r_i) - \text{Red}(r_{i-1})\big| < d_{\max})\Big) \tag{A.1}$$

where $\text{next}(r_i)$ returns the rectangle which follows directly r_i and $\text{Red}(r_i)$ returns the **R** value (in the RGB notation) of the internal color of r_i[1].

[1]The R value for pure red is 255 while it is 0 for pure blue.

A.1.2 Peak Phase

Given the set R as previously and the set $P = \{r_1, r_2, \ldots, r_p\}$, if the following holds then P is defined as a Peak Phase:

$$\begin{aligned}
&\Big(\Big(\forall t \in \mathbb{N} : 1 < t \leq |P| \Rightarrow r_t = \text{next}(r_{t-1})\Big) \wedge \Big(\exists i \in \mathbb{N} : 2 < i < p-1 \Rightarrow \\
&\big(\forall j \in \mathbb{N} : 1 < j < i \Rightarrow \big|\text{Red}(r_j) - \text{Red}(r_{j-1})\big| < d_{\text{max2}} \wedge \\
&\forall k \in \mathbb{N} : i < k < p \Rightarrow \big|\text{Red}(r_k) - \text{Red}(r_{k+1})\big| < d_{\text{max2}} \wedge \\
&\forall j, k \in \mathbb{N} : 1 \leq j < i, i < k \leq p \Rightarrow \big|\text{Red}(r_k) - \text{Red}(r_j)\big| < d_{\text{max2}} \wedge \\
&\forall z \in \mathbb{N} : (1 \leq z \leq p \wedge z \neq i) \Rightarrow \big|\text{Red}(r_z) - \text{Red}(r_i)\big| > d_{min}\big)\Big)\Big) \\
&\wedge |P| > 5 \qquad (A.2)
\end{aligned}$$

A.1.3 Unstable Phase

Given the set R as usual, a set $U = \{r_1, r_2, \ldots, r_u\}$ and a time interval length equal to l, U is defined as Unstable Phase if the following holds:

$$\begin{aligned}
&|U| = k \times l, k \in \mathbb{N} \wedge k \geq 4 \wedge \Big(\forall i, j : \Big(i \in \mathbb{N}, j \in \mathbb{N} : 1 \leq i, j \leq k \times l \wedge \\
&((\left\lfloor \frac{i}{l} \right\rfloor = 2n_i + 1, n_i \in \mathbb{N} \wedge \left\lfloor \frac{j}{l} \right\rfloor = 2n_j + 1, n_j \in \mathbb{N}) \vee (\left\lfloor \frac{i}{l} \right\rfloor = 2n_i, n_i \in \mathbb{N} \\
&\wedge \left\lfloor \frac{j}{l} \right\rfloor = 2n_j, n_j \in \mathbb{N})) \wedge i \neq j\Big) \Rightarrow |\text{Red}(r_i) - \text{Red}(r_j)| < d_{\text{min2}}\Big) \wedge \\
&\Big(\forall i, j : \Big(i \in \mathbb{N}, j \in \mathbb{N} : 1 \leq i, j \leq k \times l \wedge \big(\left\lfloor \frac{i}{l} \right\rfloor = 2n_i + 1, n_i \in \mathbb{N} \wedge \\
&\left\lfloor \frac{j}{l} \right\rfloor = 2n_j, n_j \in \mathbb{N}\big) \wedge i \neq j\Big) \Rightarrow |\text{Red}(r_i) - \text{Red}(r_j)| > d_{\text{max3}}\Big) \qquad (A.3)
\end{aligned}$$

Analyzing the Mozilla case-study we found that the values $d_{\text{max}} = 20\%, d_{\text{max2}} = 20\%, d_{\text{max3}} = 50\%, d_{\text{min}} = 65\%, d_{\text{min2}} = 20\%$ give the best results in highlighting discrete time phases.

A.2 System Production Trend Phases

We present the formal definition of the Phases detectable in a System Production Trend View like the one depicted in Figure A.2. These Phases are informally introduced in Section 4.4.2.

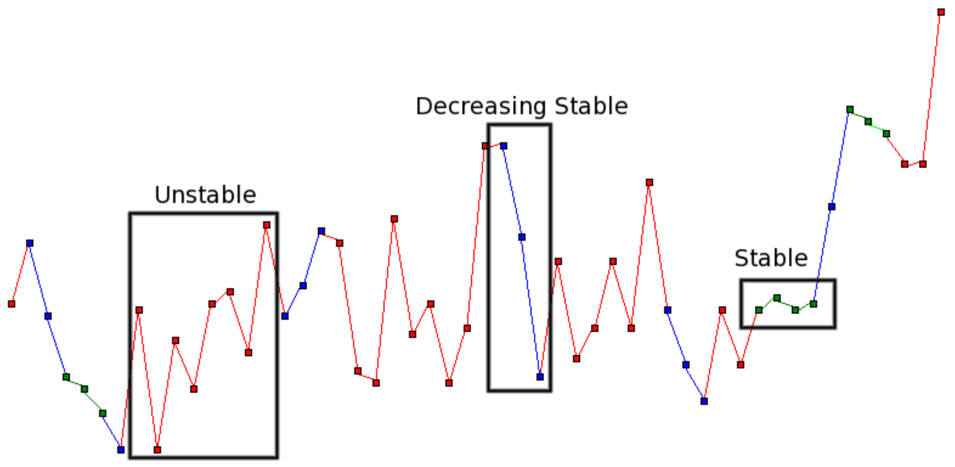


Figure A.2: The System Production Trend View applied to Mozilla. The time period is from Jan 1, 2000 to Jan 1, 2001. The detected phases are highlighted.

A.2.1 Stable Phase

The formal definition of Stable Phase is:

$$\begin{gathered} S = \{s_1, s_2, \ldots, s_n\} \quad \text{is a stable phase if} \\ n \geq 3 \wedge \forall i \in \mathbb{N} : 1 < i \leq n \Rightarrow |y(s_i) - y(s_{i-1})| < d_{\max} \end{gathered} \tag{A.4}$$

where $y(s_i)$ is the Y value of the s_i time interval production node.

A.2.2 Increasing/Decreasing Stable

The formal definition of Increasing/Decreasing Stable Phase follows:

$$\begin{gathered} S = \{s_1, s_2, \ldots, s_n\} \quad \text{is an inc/dec stable phase if} \\ n \geq 3 \wedge \forall i \in \mathbb{N} : 2 < i \leq n \Rightarrow \\ (1-\alpha)\big(y(s_{i-1}) - y(s_{i-2})\big) \leq \big(y(s_i) - y(s_{i-1})\big) \leq (1+\alpha)\big(y(s_{i-1}) - y(s_{i-2})\big) \\ \wedge \quad \operatorname{sign}\big(y(s_{i-1}) - y(s_{i-2})\big) = \operatorname{sign}\big(y(s_i) - y(s_{i-1})\big) \end{gathered} \tag{A.5}$$

Examining the Mozilla case-study we found that the values $d_{\max} = 25$, $\alpha = 50\%$ give the best results in detecting the Phases.

Appendix B

BugCrawler: Implementation

In this Chapter we describe the implementation of BugCrawler, the tool we have implemented and which made possible all the visualizations presented in this thesis.

B.1 BugCrawler

BugCrawler is an extension of CodeCrawler, a software visualization tool written in Smalltalk by M. Lanza [Lan03a]. BugCrawler supports software evolution through the combination of software metrics and software visualization. In Figure B.1 we can see a screen-shot of BugCrawler.

The discussion of the implementation of BugCrawler is structured according to the following topics:

1. **The Overall Structure**, *i.e.*, the way the tool is structured in terms of subsystems (Section B.2).

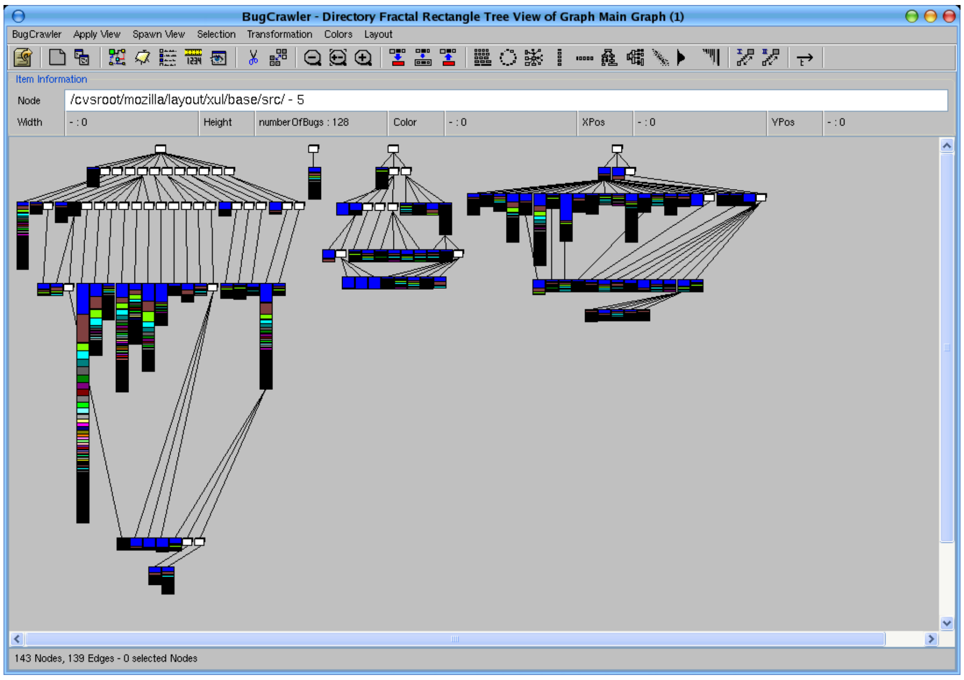
2. **The Rhdb Bridge**, *i.e.*, the subsystem responsible for retrieving data from the Release History Database and mapping it in Smalltalk objects (Section B.3).

3. **The Internal Architecture of CodeCrawler**, *i.e.*, the design of the core domain model (Section B.4).

4. **The CodeCrawler Bridge**, *i.e.*, the subsystem responsible for linking Release History Database objects with CodeCrawler entities (Section B.5).

5. **The Visualization Engine**, *i.e.*, the subsystem responsible for visualizing polymetric views (Section B.6).

6. **Interactive Facilities**, *i.e.*, the direct-manipulation possibilities that are offered to the user (Section B.7).



Node /cvsroot/mozilla/layout/xul/base/src/ - 5

Width - : 0 Height numberOfBugs : 128 Color - : 0 XPos - : 0 YPos - : 0

143 Nodes, 139 Edges - 0 selected Nodes

Figure B.1: A screen-shot of BugCrawler's main window. The visualized view is a Fractal Directory Tree View applied to the RaptorLayout module of Mozilla.

7. **Extensions implemented**, *i.e.*, a summary of the extensions implemented in BugCrawler, with respect to the original tool CodeCrawler (Section B.8).

B.2 The Overall Structure

The general architecture of BugCrawler dictates on one hand how much and which kind of functionality it provides, on the other hand it also defines how it can be extended in case of changing or new requirements.

BugCrawler adopts the *double-bridges* architecture shown in Figure B.2, which is composed of:

- *The Rhdb Bridge.* It provides flexibility by decoupling the Rhdb from the other subsystems. In this way, only the Rhdb Bridge must be changed when the Rhdb changes, while the remaining part of the system is not affected by the modification.
- *The BC-CC Bridge and the Code Model.* They act as a bridge between the visualization engine and the Rhdb entities. In order to keep a certain flexibility BugCrawler uses the facade pattern [GHJV95] which

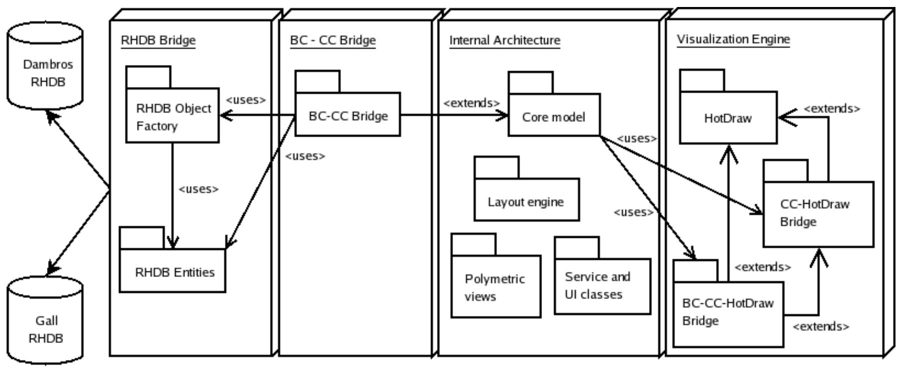


Figure B.2: The general architecture of BugCrawler, composed of four main subsystems: The RHDB Bridge, the BC-CC Bridge, the Internal architecture and the Visualization Engine.

hides both the visualization engine and the Rhdb entities from the core. In this way, only the facade classes (which implement the facade pattern) must be changed when the visualization engine or the Rhdb entities change.

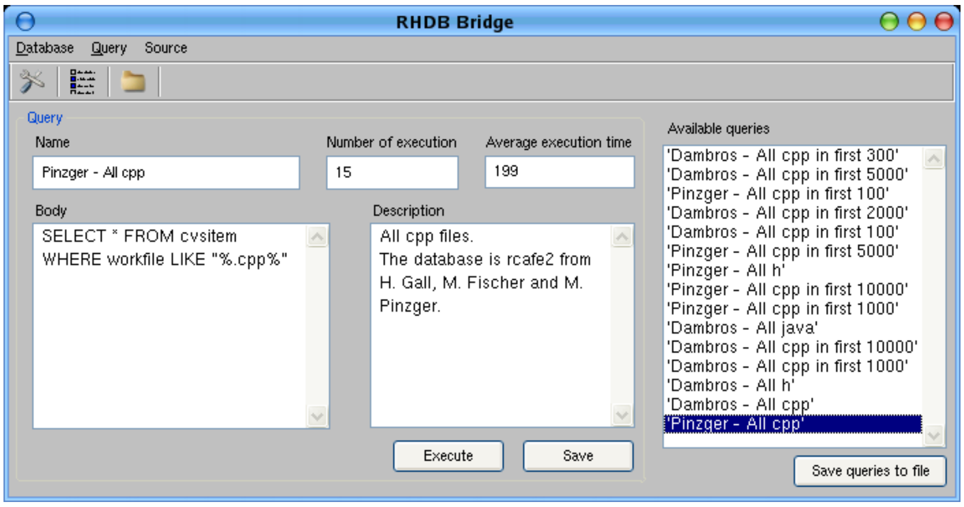
B.3 The Rhdb Bridge

The Rhdb is the BugCrawler subsystem responsible for managing the connection with the Release History Database. It provides to the user the opportunity of performing queries to the database, using the window depicted in Figure B.3.

The queries can be composed, modified and saved in XML format. Moreover, since the execution of some queries can take a lot of time, the number of executions and the average execution time are computed and stored for each query. When the user performs a query, the Rhdb Bridge retrieves the data from the database, mapping it in Smalltalk objects (Rhdb entities) according to the class hierarchy shown in Figure B.4.

As we can see, the main Rhdb entities are designed as follows:

- **Module**, *i.e.*, the highest level entity which can contain one or more directories.
- **Directory**. It implements the composite pattern [GHJV95]. A directory can include subdirectories and products.
- **Product**. It represents a CVS product and contains at least one revision.



Name: Pinzger - All cpp
Number of execution: 15
Average execution time: 199

Body:

```
SELECT * FROM cvsitem
WHERE workfile LIKE "%.cpp%"
```

Description: All cpp files. The database is rcafe2 from H. Gall, M. Fischer and M. Pinzger.

Available queries:

```
'Dambros - All cpp in first 300'
'Dambros - All cpp in first 5000'
'Pinzger - All cpp in first 100'
'Dambros - All cpp in first 2000'
'Dambros - All cpp in first 100'
'Pinzger - All cpp in first 5000'
'Pinzger - All h'
'Pinzger - All cpp in first 10000'
'Pinzger - All cpp in first 1000'
'Dambros - All java'
'Dambros - All cpp in first 10000'
'Dambros - All cpp in first 1000'
'Dambros - All h'
'Dambros - All cpp'
'Pinzger - All cpp'
```

Figure B.3: The BugCrawler's RHDB Bridge interface.

- **Revision and Bug**, *i.e.*, the CVS revision and the Bugzilla Bug. The n-to-n relationship represents the fact that a revision can be affected by many bugs, while a bug can affect many revisions.

For all the classes mentioned above, we implemented the corresponding proxies (proxy patter [GHJV95]). In this way, the original classes represent accurately the information stored in the database, while the proxy classes provide complex and composite data concerning the corresponding entities.

To create Rhdb objects we use the small hierarchy having the abstract class RhdbObjectFactory on the top. The two concrete subclasses Rhdb-DambrosObjectFactory and RhdbGallObjectFactory make it possible to create instances from both our Release History Database and the one developed

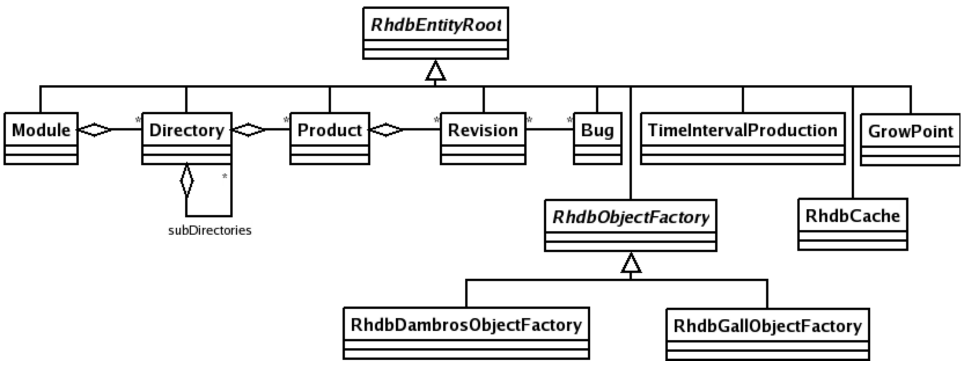


Figure B.4: The class diagram of the Rhdb Bridge.

by Gall *et al.* [FPG03b]. Each subclass encapsulates the particularities of the corresponding Database.

We designed the RhdbCache class in order to improve the performance of the data retrieval. This class implements a cache mechanism which avoids to retrieve data from the database if it is already present in the system.

B.4 The Internal Architecture of CodeCrawler

In this Section we describe the internal architecture of CodeCrawler, as designed by M. Lanza [Lan03a] and slightly extended by us. The internal architecture can be divided into four parts: (1) the core model, (2) the polymetric views subsystem, (3) the layout engine and (4) the user interface and service classes.

1. **The Core Model**. We can see a simple class diagram of CodeCrawler's core model in Figure B.5.

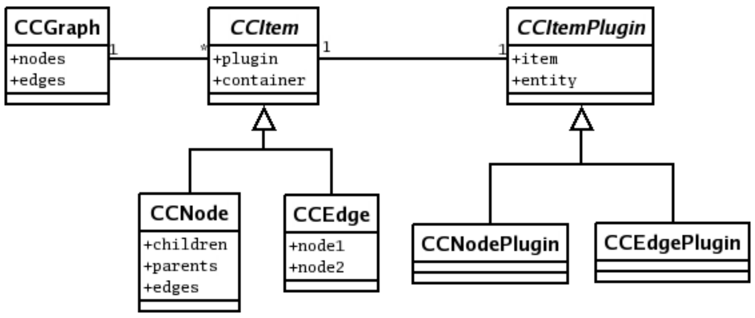


Figure B.5: The core model of CodeCrawler.

CodeCrawler uses nodes to represent entities (modules, directories, products, etc.) and edges to represent relationships (bugs sharing, bugs containing, etc.) between the entities. The nodes and edges are contained within a class that represents a graph in the mathematical sense. Both the node class (CCNode) and the edge class (CCEdge) inherit from an abstract superclass which represents a general item (CCItem). CCItem serves as bridge between the visualization part (it contains an attribute named figure which points to a figure class). It is also a bridge to a parallel plugin hierarchy (it contains an attribute named plugin which points to a plugin class). The classes in the plugin hierarchy provide most of the functionality of the nodes and edges. This functionality is separated into an own hierarchy in order to obtain more flexibility and a higher degree of extensibility.

2. **The Polymetric Views**. All information regarding a certain visualization (what is to be visualized, how, where, which metrics, etc.) is

stored by means of a view specification class (CCViewSpec). When it comes to display a view of a software system, a view builder (CCView-Builder) interprets an instance of a specification class and builds the needed visualization.

3. **The Layout Engine**. In CodeCrawler all layouts (at this time ca. 15) inherit from a common abstract superclass (CCAbstractLayout). A layout class takes as input a collection of node figures and assigns a position to each of them.

   We extended the CodeCrawler layout engine by adding 6 layouts (Black Holes, Time Based Evolution Matrix, etc.). Furthermore, we designed a new type of layout (applicable to composite figures) which assigns positions to composite figures according to the properties of their inner figures.

4. **The Service and UI Classes**. Besides the classes mentioned above, CodeCrawler contains many more classes which provide for various services, for example storing constants and color mappings. Other classes are pure user interface classes (Dialogs, Panels, etc.). Since these classes do not have any features that are particularly important for software visualization tools, their discussion is omitted.

   We added our service classes creating them from scratch for services concerning only BugCrawler, while we inherit from CodeCrawler in the other cases (for example User Interface).

B.5 The CodeCrawler Bridge

The bridge linking BugCrawler and CodeCrawler is implemented by means of the plugin hierarchy shown in Figure B.6. As we have seen in Section B.4, this hierarchy belongs to the core model of CodeCrawler.

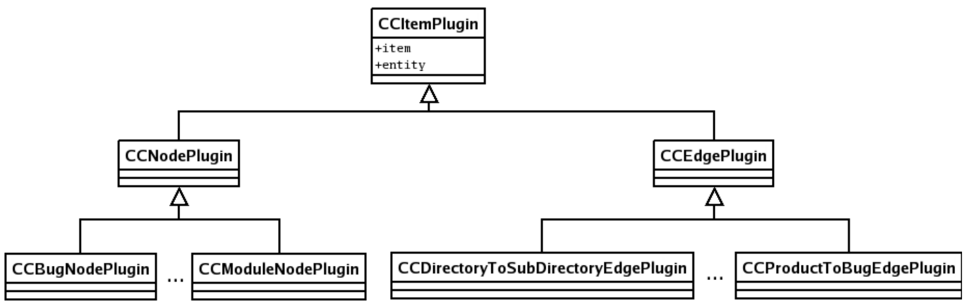


Figure B.6: The plugin class hierarchy. It implements the bridge between BugCrawler and CodeCrawler.

The abstract superclass CCItemPlugin defines an attribute named entity which points to the needed class, *e.g.*, in our case a Rhdb entity. To protect against changes in the representation of the Rhdb entities we use facade classes, *i.e.*, in BugCrawler we implemented a hierarchy of plugins which have counterparts in the Rhdb entities. To make an example, to represent a Bug class, BugCrawler implements a CCBugNodePlugin class which interfaces with the Bug class. The same structure is used for all the other Rhdb entities.

B.6 The Visualization Engine

The original tool CodeCrawler uses as visualization engine the HotDraw framework [Bra95]. We implemented the BugCrawler visualization engine inheriting from both BugCrawler (in most cases) and HotDraw.

B.6.1 The CodeCrawler Visualization Engine

The HotDraw framework is a lightweight 2D editor written in Smalltalk, consisting of ca. 150 classes. It provides for basic graphical functionalities like zooming, scaling, elision and comes with a collection of simple figures (rectangles, lines, ellipses, composite figures, etc.) that can be easily reused and extended through subclassing, as CodeCrawler does indeed: The subclasses include CCDrawing, which represents the drawing surface on which the visualization is displayed, and several figures classes (CCRectangleFigure, CCLineFigure, etc.) which add functionality to the quite simple HotDraw figure classes. However, these subclasses do not offer protection against changes in HotDraw, since the subclasses would be affected too. Therefore in CodeCrawler three classes (CCItemFigureModel, CCNodeFigureModel and CCEdgeFigureModel), organized in a small hierarchy, serve as facade classes for the figure classes that subclass HotDraw's classes. This allows us to replace on-the-fly the graphical representation, *e.g.*, the figure of a node or an edge. Furthermore, the facade classes implement several operations that we want to effect on figures (graphical operations, geometric transformations, etc.) and delegate them to the appropriate concrete figures on the screen.

B.6.2 The BugCrawler Visualization Engine

We extended the CodeCrawler visualization engine by means of the strategy pattern [GHJV95] depicted in Figure B.7. Its structure is based on the combination of two hierarchies: The first, having BCCompositePolygonFigure as superclass, is composed of classes mapping Rhdb entities.

The second hierarchy, having the abstract class BCFigureSpec on the top, implements a family of algorithms for drawing figures. In this way, the

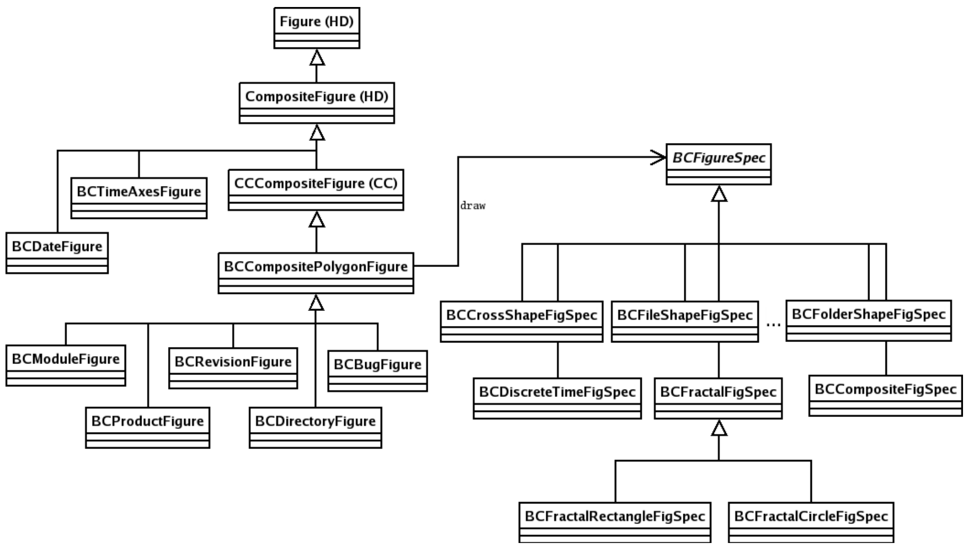


Figure B.7: The class hierarchy implementing the BugCrawler visualization engine.

figure appearances can be dynamically assigned. For example, in the Discrete Time Directory Tree View the class BCDirectoryFigure is associated with either a BCDiscreteTimeFigSpec or a BCFolderShapeFigSpec class depending on the number of products the directory contains (zero or more). In the same way, a BCRevisionFigure can have a rectangle shape or a cross shape in the Product TimeLine View, depending on the dead state of the revision.

The use of the strategy pattern in this context provides the following benefits:

- Each class added to the BCCompositePolygonFigure hierarchy can use all the drawing algorithms just implemented in the BCFigureSpec hierarchy.
- We can implement new drawing algorithms without modifying any class belonging to the BCCompositePolygonFigure hierarchy. The new algorithms will be applicable to all the figures, giving a high degree of extensibility.
- The classes belonging to the BCCompositePolygonFigure hierarchy can define behaviors which are independent from the figure appearances defined by FigureSpec classes. For example, each figure class implements a context-based menu (described in detail in Section B.7) which is identical for all the figure appearances, even for those which will be added later on.

The classes BCDateFigure and BCTimeAxesFigure are not associated with any Rhdb entity or relationship, because they represent information concerning entire views. For example, the BCTimeAxesFigure represents the temporal dimension in the DISCRETE TIME COMBO MODULE and PRODUCT TIMELINE views. These classes don't need the CCItemFigureModel facade functionality provided by CodeCrawler, since they don't have the item counterparts. That is the reason why we chose to inherit them directly from HotDraw instead of CodeCrawler.

B.7 Interactive Facilities

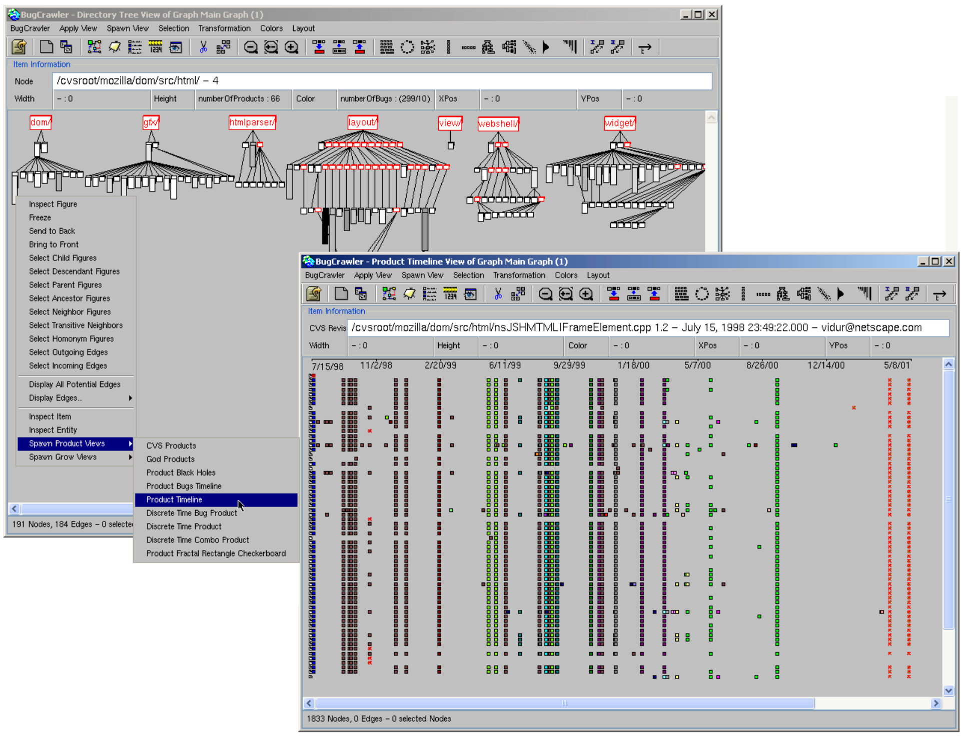
Once the visualization is rendered on the screen, the user not only wants to look at it, he also wants to interact with it. According to Storey *et al.* [SFM99] this helps to reduce the cognitive overhead of any visualization. In CodeCrawler the HotDraw framework provides for direct manipulation at a purely graphical level, *i.e.*, the user can click, drag, double-click, delete, zoom out/in, spawn child windows, etc. CodeCrawler uses that functionality by providing context-based (popup) menus for each node and edge. Note that, depending on the type of the node, different choices are offered to the user. CodeCrawler offers also a so called macro navigation, a facility that enables the user to go back/forth from one view to another.

We implemented the interactive facilities of BugCrawler extending both context menus and macro navigation. For example, the BCDirectoryFigure defines a context menu which allows the user to apply views at the product granularity, having the selected directory as target. Figure B.8 shows the original view (CRITICAL DIRECTORY TREE) with the context menu in the left window, and the chosen view (PRODUCT TIMELINE) applied to the selected directory in the right window.

B.8 BugCrawler: Extensions implemented

The main extensions implemented in BugCrawler, with respect to the original tool CodeCrawler, can be summarized as:

- A bridge responsible for querying MySql databases and importing the data as Smalltalk objects. The bridge provides also a cache mechanism to improve its efficiency.
- A class hierarchy representing Rhdb entities and factory classes to create them.
- A bridge linking Rhdb entities with CodeCrawler plugins.
- New entities, relationships, metrics and layouts.



Node /cvsroot/mozilla/dom/src/html/ – 4
Width – : 0 Height numberOfProducts : 66 Color numberOfBugs : (299/10) XPos – : 0 YPos – : 0

191 Nodes, 184 Edges – 0 selecte

CVS Revis /cvsroot/mozilla/dom/src/html/nsJSHMTMLIFrameElement.cpp 1.2 – July 15, 1998 23:49:22.000 – vidur@netscape.com
Width – : 0 Height – : 0 Color – : 0 XPos – : 0 YPos – : 0

1833 Nodes, 0 Edges – 0 selected Nodes

Figure B.8: BugCrawler at work. The context menus are dynamically built depending on the entity or relationship selected.

- New figure classes hierarchically organized and a family of algorithms for drawing these figures.
- New context-based menus and macro navigation for Rhdb entities.

Bibliography

[Apa] Apache http server home page. http://httpd.apache.org.

[BAHS97] Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[BE96] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[BM99] Elizabeth Burd and Malcolm Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering, WCRE '99*, pages 168–174, 1999.

[Bra95] John Brant. Hotdraw. Master's thesis, University of Illinois at Urbana-Chanpaign, 1995.

[Buga] A bug's life cycle. http://bugzilla.remotesensing.org/bug_status.html.

[Bugb] Bugzilla home page. http://www.bugzilla.org/.

[Bugc] Bugzilla installation list. http://www.bugzilla.org/installation-list/.

[Cle] Clearcase home page. http://www-306.ibm.com/software/awdtools/clearcase/.

[CVS] Cvs home page. http://www.cvshome.org/.

[Dav95] Alan Mark Davis. *201 Principles of Software Development.* McGraw-Hill, 1995.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[DLS00] Stéphane Ducasse, Michele Lanza, and Lukas Steiger. Supporting evolution recovery: a query-based approach. In *ECOOP '2000 International Workshop of Architecture Evolution*, 2000.

[Eic01] Stephen G. Eick. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 6(1):1–12, 2001.

[FPG03a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, November 2003.

[FPG03b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, September 2003.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (International Conference on Software Maintenance)*, 2004.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Mass., 1995.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, 2003.

[GJKT97] Harald Gall, Mehdi Jazayeri, René R. Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance 1997 (ICSM '97)*, pages 160–166, 1997.

[GL04] Tudor Gîrba and Michele Lanza. Using visualization to understand the evolution of class hierarchies. In *11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, 2004.

[Gno] Gnome home page. http://www.gnome.org/.

[GSV02] David Grosser, Houari A. Sahraoui, and Petko Valtchev. Predicting software stability using case-based reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engienering (ASE '02)*, pages 295–298, 2002.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, San Jose, California, 2000.

[HP96] R. C. Holt and J. Pak. GASE: Visualizing software evolution-in-the-large. In *Proceedings of WCRE '96*, pages 163–167, 1996.

[HS98] J. W. Harris and H. Stocker. *Handbook of Mathematics and Computational Science*. Springer-Verlag, New York, 1998.

[Jaz02] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[JGR99] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99 Proceedings (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society, 1999.

[KDE] Kde home page. http://www.kde.org/.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[Lan03a] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[Lan03b] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[LB85] Manny M. Lehman and Les Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[Leh96] Manny M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[Lin] The linux kernel home page. http://www.kernel.org/.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, 1994.

[LPR$^+$97] M.M. Lehman, D. E. Perry, J. F. Ramil, W. M. Turski, and P. D. Wernick. Metrics and laws of software evolution - the nineties view. In *Metrics '97, IEEE*, pages 20 – 32, 1997.

[LPR98] M. M. Lehman, Dewayne E. Perry, and Juan F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 208–217, 1998.

[MFH02] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[Moz] Mozilla home page. http://www.mozilla.org/.

[MyS] Mysql database. http://www.mysql.com.

[Red] Redhat home page. http://www.redhat.com/.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics.* Addison Wesley, 1996.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience.* The MIT Press, 1998.

[SFM99] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

[Som00] Ian Sommerville. *Software Engineering.* Addison Wesley, sixth edition, 2000.

[Sou] Sourcesafe home page. http://msdn.microsoft.com/vstudio/previous/ssafe/.

[Sub] Subversion home page. http://subversion.tigris.org.

[Sus] Suse home page. http://www.novell.com/linux/suse.

[Tur96] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599–600, 1996.