DISSERTATION

ArchView - Analyzing Evolutionary Aspects of Complex Software Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Leitung von

Univ.-Prof. Dipl.-Ing. Dr.techn. Harald C. Gall
Institut für Informatik
Universität Zürich
und
o.Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme
Technische Universität Wien

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Martin Pinzger

pinzger@infosys.tuwien.ac.at

Matrikelnummer: 9626545
Stuben 238, A-6542 Pfunds, Österreich

Wien, im Mai 2005

Kurzfassung

Grosse und komplexe Software Systeme sind laufenden Änderungen ausgesetzt die in allen Lebensabschnitten auftreten, wie in der Entwicklung, Wartung, Migration und Ausscheidung. Diese Änderungen sind einerseits notwendig, um den Erfolg eines Software Systems zu garantieren, aber andererseits wirken sie sich auf die Architektur und das Design eines Software Systems aus. Aus diesem Grunde ist eine laufende Überwachung und Analyse der Architektur und des Designs notwendig, um Fehler und Unzulänglichkeiten frühzeitig zu erkennen und zu beheben.

In dieser Dissertation stellen wir den ArchView Ansatz vor, der sich mit der Analyse und Bewertung von Software Modulen hinsichtlich ihrer strukturellen und evolutionären Eigenschaften befasst. Software Module sind architekturelle Elemente, die durch Source Files, Klassen oder Aggregationen von diesen implementiert werden. Das primäre Ziel unsere Arbeit ist die Identifikation jener Module und Strukturen, welche Bad Smells in der Implementierung, im Design und der Architektur darstellen und behoben werden müssen.

Als Grundlage für die Analyse und Bewertung der strukturellen und evolutionären Eigenschaften von Software Modulen verwendet ArchView Metriken und die Kopplungs-Beziehungen zwischen Modulen. Die Metriken bewerten die Grösse, Komplexität, Kopplungsgrad, Änderungs- und Fehlerhäufigkeit von Modulen. Zusammen mit den Kopplungsbeziehungen geben sie Aufschluss über Qualität der Implementierung. Bezüglich der Evolution führen wir diese Messungen für mehrere Releases durch und erhalten so Trends, welche uns auf Unzulänglichkeiten in der Implementierung, dem Design und der Architektur hinweisen.

Für die Darstellung der Ergebnisse verwendet ArchView Graphen in denen Knoten die Module und Kannten die Kopplungsbeziehungen darstellen. Zur Bewältigung der grossen Datenmenge stellen wir eine erweiterte Graph-Visualisierungs-Technik vor, die auf dem Prinzip des Measurement Mappings beruht. Diese Technik erlaubt die gleichzeitige Darstellung von Modulen mit vielfachen Metriken von mehreren Releases und deren Kopplungsbeziehung in einem Graph. Die so erstellten Graphen ermöglichen uns die visuelle Identifizierung von jenen Modulen und Kopplungsbeziehungen, die Bad Smells darstellen.

Wir demonstrieren und validieren den ArchView Ansatz anhand einer Fallstudie mit dem Open Source Projekt Mozilla. Die Resultate der Fallstudie zeigen die strukturellen und evolutionären Eigenschaften von Mozilla und weisen auf Bad Smells in der Architektur und im Design hin.

Abstract

Large and complex software systems are confronted with continuous changes during all stages in their life comprising development, maintenance, migration, and retirement. On the one side these changes are mandatory to guarantee the success of a software system but on the other side changes affect the architecture and design of a software system. Therefore, a continuous observation and analysis of the architecture and the design is mandatory to early identify errors and shortcomings and resolve them.

In this dissertation we propose the ArchView approach that focuses on the analysis and evaluation of software modules regarding their structural and evolutionary aspects. Software modules are architectural elements that are implemented in source files, classes or aggregations of them. The primary objective of our work is the identification of modules and structures that represent Bad Smells in the source code, the design, and the architecture to be resolved.

For the analysis and evaluation of the structural and evolutionary properties of software modules ArchView basically uses software metrics and coupling dependencies between modules. Metrics assess the size, complexity, coupling degree, modification and problem frequency of modules. In combination with coupling dependencies they provide information about the quality of an implementation. Regarding the evolution we perform these measurements for a number of releases to yield trend data that points us to shortcomings (Bad Smells) in the implementation, design, and architecture.

For the presentation of the results ArchView uses graphs in which nodes represent modules and edges represent the coupling relationships. To handle the huge amount of information we introduce an extended graph visualization technique that is based on the principle of measurement mapping. Our technique facilitates the representation of modules with multiple metric values of a number of releases, and their coupling relationships in one graph. The so created graphs allow us to visually identify those modules and coupling dependencies that indicate Bad Smells.

We demonstrate and validate the ArchView approach in a large case study with the Mozilla open source project. Results clearly show the structural and evolutionary properties of Mozilla and point to Bad Smells in the architecture and the design.

Acknowledgments

It is my pleasure to thank the many people who made this thesis possible.

First of all, I would like to thank my supervisors, Prof. Harald C. Gall and Prof. Mehdi Jazayeri, for opening the door to research, their guidance, and continuous support.

Special thanks goes to Michael Fischer for his discussions of and input on release history data and visualization techniques. I still remember the numerous extreme paper writing sessions we had together with Harald. Furthermore, I would like to thank Michele Lanza for his feedback and support on CodeCrawler and Beat Fluri as well as Eveline Suter for proof-reading this thesis.

I also would like to thank my colleagues of the Distributed Systems Group, Vienna where I started my research, and the Institute for Informatics, Zurich that has become my new working place. It was always fun to work together with you and still is.

Last but not least, I am grateful to my parents who allowed me to do what I liked to do. They always supported me in whatever I did and wherever it was possible. Many thanks go to my friends in Pfunds, Vienna, and Zurich. They pointed me to other activities that represented the perfect contrast program to my research.

Martin Pinzger
Zurich, Switzerland, May 2005

CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Large complex software systems undergo frequent changes during their life-cycle that are carried out as maintenance and evolution tasks. These tasks are concerned with fixing errors or adapting the system to new requirements. Attempts to estimate the costs of software maintenance and evolution yielded from 50% up to 75% of the total software project costs [Boe81], [Dav95], [Som00]. Much of this effort is given to program understanding ranging from understanding a system's architecture and design concerns (aspects) and its implementation.

The engineer needs to build mental models that show these concerns. Visualization of abstracted views on lower-level information has been accepted as a useful means to build these models [SFM99]. With regard to the computation of abstract views research projects concentrated on developing reverse engineering techniques and tools. According to Chikofsky and Cross reverse engineering refers to:

> The process of analyzing a subject system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction [CC90].

Reverse engineering tools aim at providing *(semi-)automatic* support for the extraction of higher-level representations. They are important instruments for maintaining and evolving software systems. Abstracted views are also mandatory to perform quality assessment of the implemented architecture and design.

In terms of extracting views on the software architecture of a system the reverse engineering technique is also called *architecture recovery* or *architecture reconstruction*. This thesis is focused on reverse engineering and, in particular, on architecture recovery to abstract architectural views that facilitate the analysis of *structural* and *evolutionary aspects* of software systems.

1.1 PROBLEM STATEMENT

The abstraction of views on the architecture of software systems has been subject to research over the past years and several commercial tools and research prototypes emerged. They focus on extracting data models from source code and execution traces. On top of the data models they perform metric measurements and provide facilities to navigate the data and abstract higher-level views. Graph-like representations are being used in which nodes represent source code or execution entities and edges relationships between them.

In the case study with the large open source project Mozilla, we applied a number of these tools and utilities. For instance, we used the commercial tool Imagix-4D[1] to parse the C/C++ source code, compute metrics, and navigate the source code. We further used the graph visualization tools CodeCrawler [Lan03], Rigi [MK88], [Won98] and SHriMP [SM95] to create the higher-level views. To summarize, our experiences with these tools yielded that they are useful to extract, browse and navigate the source code but fail in: 1) *creating and presenting higher-level views*; and 2) *analyzing evolutionary aspects*.

The first problem showed that existing approaches lack capabilities to create views able to highlight implementation, design, and architecture specific aspects. They use graphs with primitive glyphs for nodes and edges that are not sufficient to convey these aspects.

The second problem concerns software evolution analysis. Existing approaches concentrate on analyzing static and dynamic information of one particular software release but miss the aspect of evolution. For instance, the analysis of which entities were vulnerable to problems and had to be modified frequently is not supported. And, although recently there has been research done in this area several issues are left open. They concern: a) the integration of modification and problem report data to analyze fault and change proneness of implementation units; and b) trend analysis of implementation units and relationships over several software releases.

The approach presented in this thesis addresses the open issues. We base on existing extraction, analysis and visualization techniques and provide extended and new techniques to create novel high-level views on the implemented architecture and its evolution.

1.2 The Approach

The ArchView approach is an architecture recovery and analysis approach that addresses the extraction of higher-level views on a software system. Views facilitate the analysis of implementation and evolution specific aspects. With respect to implementation specific aspects our approach takes into account source code model data of several source code releases. Source code models are obtained per release by using existing static and dynamic fact extraction techniques. Concerning software evolution analysis ArchView processes modification and problem report data as obtained from the Concurrent Versions System (CVS)[2] and the bug reporting system Bugzilla[3].

ArchView integrates the different data models into one common model that allows for the navigation between source code models and corresponding release history data. Furthermore, the integrated data model is used to compute the change coupling relationships, abstract the

[1]http://www.imagix.com
[2]https://www.cvshome.org
[3]http://www.bugzilla.org

higher-level views and compute several new evolutionary metrics. Metrics refer to: 1) problem and modification report metrics; and 2) source code size, complexity, and coupling metrics tracked over a number of n releases. They characterize the problem and modification frequency of implementation units and their trend in size, complexity, and coupling.

For the visualization of abstracted views ArchView uses and extends the polymetric views technique introduced by Lanza and Ducasse [LD03]. Polymetric views follow the principle of *measurement mapping* in that larger metric values lead to larger glyphs in a graph [FP96]. This results in views that, for instance, point out change prone and hide idle implementation units. In order to visualize multivariate data of n releases ArchView extends polymetric views by Kiviat diagrams and graphs. They allow the user to study multiple aspects in parallel, such as the relation between complexity and modifications or metric trends. Concerning the latter aspect Kiviat diagrams highlight strong changes in the evolution indicating improvements or degradations of implementation units. Both visualization techniques are accompanied by a set of pre-defined views that are part of the ArchView approach.

1.3 Contributions

The contributions of this thesis comprise:

- The *E-FAMIX Meta Model* layouts the data model for integrating source code data of several releases with modification and problem report data.
- The *ArchView Integration Algorithm* that based on the name of entities integrates the different data models according to the E-FAMIX meta model. It further computes the change coupling dependencies.
- The *ArchView Abstraction Algorithm* that based on the integrated data model aggregates source code relationships and change coupling dependencies to visualize and analyze them on the level of software modules.
- The *Evolution Metrics* comprising metrics of modification and problem reports as well as source code metrics that are tracked over n releases. They are mandatory to perform software evolution analysis.
- The *Extended Polymetric Views Visualization Technique* used by ArchView to visualize multiple metric values of several releases as graphs.
- The set of *Module Evolution Views* focusing on identifying the change prone software modules and heavy coupling dependencies.
- The validation of the ArchView approach with the *Mozilla* open source project indicating the benefits as well as the open issues of the ArchView approach.

1.4 Thesis Outline

The thesis is structured as follows:

- In Chapter 2, we introduce the terms and basic concepts used in architecture recovery and software evolution analysis.
- In Chapter 3, we present related work in the areas of software evolution analysis, architecture recovery, and software visualization.
- In Chapter 4, we introduce the ArchView architecture recovery and analysis approach. The focus is on the ArchView process and its key concepts and features.
- In Chapter 5, we introduce the E-FAMIX meta model and present the techniques we use to extract and integrate the source code model, modification, and problem report data.
- In Chapter 6, we introduce the containment hierarchy model of ArchView according to which lower-level information is abstracted. Furthermore, we present the abstraction algorithm and the different metrics that are computed and used by ArchView.
- In Chapter 7, we present the polymetric views techniques and the extension of it to Kiviat diagrams and Kiviat graphs. They facilitate the visualization of multiple metric values of up to n releases in parallel. Based on both techniques we specify new system hotspots views that focus on the visualization of evolutionary and coupling aspects of software systems.
- In Chapter 8, we present the Mozilla case study we used to validate the ArchView approach. In the case study we concentrated on the analysis of the implementation and evolution aspects of Mozilla's content and layout module. For this we took into account data of seven recent releases and the data obtained from the Mozilla CVS[4] and Bugzilla[5] repositories.
- In Chapter 9, we draw the conclusions of the ArchView approach and discuss open issues and future work.
- In the Appendix, we provide the detailed E-FAMIX meta model description and the list of our publications that are related to the ArchView approach.

[4]http://www.mozilla.org
[5]https://bugzilla.mozilla.org

Chapter 2

Software Architecture and Evolution

This chapter provides the background information including the definition of terms used to describe and understand the ArchView approach. Most of the definitions are taken from recent work in the field of software architecture, architecture recovery, and software evolution analysis.

2.1 Software Architecture

Software Architecture is a term frequently used when discussing, designing and analyzing large complex software systems. However, there are different meanings and understandings of what software architecture actually is. According to this diversity there exists a number of quite similar definitions, such as collected by the Software Engineering Institute of the Carnegie Mellon University[1].

The IEEE 1471-2000 standard definition for a software architecture is:

> A *software architecture* is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [14700].

A system inhabits an environment, that can influence the system, and a system has one or more stakeholders. Each stakeholder has interests in, or relative to, that system.

> A *system stakeholder* is an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system [14700].
>
> *Concerns* are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability [14700].

[1]http://www.sei.cmu.edu/architecture/definitions.html

2.2 Architectural Views and Viewpoints

An architecture can be recorded by an architectural description that is organized into one or more constituents called architectural views.

> A *view* is a representation of a whole system from the perspective of a related set of concerns [14700].

Each view conforms to a viewpoint and addresses one or more of the concerns of the stakeholders.

> A *viewpoint* is a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [14700].

The viewpoint determines the languages to be used to describe the view and any associated modeling methods or analysis techniques to be applied to these representations of the view. A number of books and articles exist that address the issues. Regarding the description of a software architecture they share the view that different views and viewpoints are mandatory. For instance, Kruchten proposed the “4+1” view model [Kru95] that distinguishes: logical view, process view, development view, physical view, and use case view. The latter view ties together the other four views. Similarly, Hofmeister *et al.* [HNS00] proposed to use four different views: conceptual architecture, module interconnection architecture, execution architecture, and code architecture.

Regarding viewpoints Bass *et al.* [BCK03] presented a categorization of architectural views into three different view types:

- *Module Views.* Elements of these views are modules which are units of implementation with a well-defined interface providing a coherent unit of functionality. With module views the decomposition of a software system into its implementation units is described. Consequently, module views include a description of the functionality assigned to each module, possible generalization of modules, and uses-relationships between modules.

- *Component-and-Connector Views.* Here, the elements are runtime components and connectors. Components are the principal computational units such as clients, servers, processes and databases. Connectors represent the communication vehicles among components, such as remote procedure calls, named pipes, and sockets. Basically, views of this type describe the executing components, their interactions and run-time behavior within the running application.

- *Allocation Views.* Views of this type are concerned with the mapping of the various modules, components and connectors to the development environment and runtime environment respectively. Consequently, elements of this view type are software modules, components and connectors, and elements of the environment ranging from hardware resources (computer, processor, etc.) to human resources (designers, programmers, testers, etc.) to which software elements are allocated to.

The ArchView approach focuses on the module views because they represent aspects (concerns) of the implementation of a software system. The other two view types are out of the scope of this thesis. Regarding module views the primary subject architectural elements are *software modules* and their relationships.

> A module is an implementation unit of software that provides a coherent unit of functionality [CBB$^+$02].

Programming language units, such as Smalltalk, Java, C++ or Modula modules, are examples of software modules. Modules can both be aggregated and decomposed. The relationships that are used to design module views are:

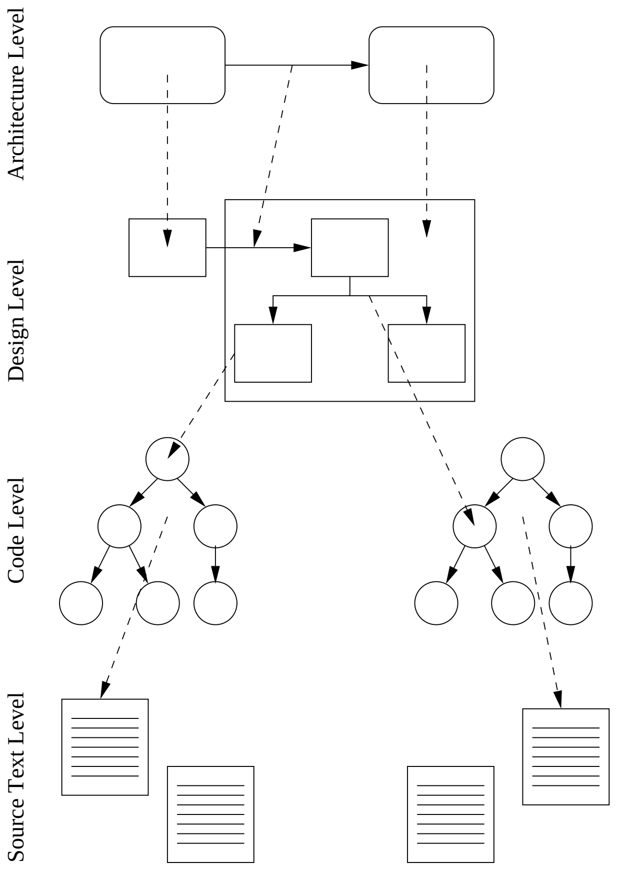
- *Is-part-of*: Defines a part/whole relationship between the submodule A – the part – and the aggregate module B – the whole, or parent. In this thesis, we also use this relationship to express the mapping between modules and their corresponding elements in the design level (*i.e.,* package, class, directory, file) that implement each module. In Figure 2.1, the is-part-of relationship is represented by the *contains* relationship.
- *Depends-on*: Defines a dependency relationship between the modules A and B. This relation is typically used early in the design process when the precise form of the dependency has yet not be decided. From the perspective of architecture recovery the type of dependency relationship is already defined in the code. Instead of depends-on we also use the design level relationships, such as class aggregation, function call, or variable access relationships.
- *Is-a*: Defines a generalization relationship between a more specific module – the child A – and a more general module – the parent B. Because the focus of ArchView is on object-oriented systems we use the class inheritance for the is-a relationship.

The objective of reverse engineering and, in particular, architecture recovery is to extract module views that reflect the concerns used to implement a software system. Basically, these concerns can reside on different abstraction levels which is be described in the next section.

2.3 Abstraction Levels

Views on the implementation of a software system can be of different levels of abstraction depending on the concerns to represent. Figure 2.1 shows the different abstraction levels, the entities and relationships used to create the views, and the mapping (hierarchy) between the levels. The latter is used to abstract higher-level views from lower-level source code information and vice versa to trace higher-level views down to source code. The model is derived from the models proposed by Riva [Riv04] and Kazman *et al.* [KWC98].

The four abstraction levels are:



Entities:	module
Relations:	uses, generalization
Hierarchy:	contains
Entities:	directory, file, package, class, method, function, attribute, variable,
Relations	inherits, aggregates invokes, accesses
Hierarchy:	contains,
Entities:	programming language constructs
Relations:	syntactic nesting, links

Figure 2.1: Abstraction levels and corresponding entity and relationship types used to create views on the implementation. Dotted arcs indicate the mapping between entities of different abstraction level.

- *Architecture*: At the top, there are the architectural viewpoints that describe the architecturally relevant aspects of the system. In this thesis we focus on the viewpoints that concern software modules, their decomposition, uses, and generalization.

- *Design*: The design viewpoints capture the design aspects of software systems, such as the object-oriented design aspects. The representation of the aspects is with source code models – they are close to the source code but do not contain all the details. In this thesis we use the E-FAMIX meta model for a language-independent representation of source code of object-oriented programming languages and release history data.

- *Code*: The code view points give a detailed representation of the syntactic structure of classes and source files. Typically, this is modeled with the Abstract Syntax Tree (AST).

- *Source Text*: The implementation of the system consists of a set of documents, such as source files, configuration files, build commands, log files, etc..

Regarding this thesis the important levels are Design and Architecture. The lower-two levels are handled by lexical and syntactical analysis tools that parse the source text, and extract a full or partial abstract syntax tree representation from which the design level entities and relationships are obtained.

2.4 SOFTWARE EVOLUTION

Software evolution encompasses all stages in the life of a software system: development, maintenance, migration, and retirement. A software system has to react to the changing forces of the environment or it will become obsolete. It is this change over time that is the focus of software evolution research. In this context, Parnas stated:

> Software, like people, gets old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable [Par94].

Analyzing software evolution refers to analyzing the changes, their causes, and effects. Concerning the causes and effects Lehman *et al.* identified a set of laws called the "Laws of software evolution" [LPR$^+$97]. They present general observations, for example:

- *Continuing Change*: Systems must be continually adapted else they become progressively less satisfactory;
- *Increasing Complexity*: As a system evolves its complexity increases unless work is done to maintain or reduce it; and
- *Continuing Growth*: The functional content of a system must be continually increased to maintain user satisfaction over their lifetime.

During the evolution of software systems changes occur on all levels of abstraction. The architecture of a software system as well as its design and implementation evolve.

A major problem with evolution is that software systems suffer from signs of aging as they are adapted to changing requirements. Signs, for instance, refer to architectural erosion, architectural drift, or architectural mismatch. *Architectural erosion* is defined as violations in the architecture that lead to increased system problems and brittleness" [PW92]. Perry and Wolf also defined the term *architectural drift* as "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of architecture". *Architectural mismatch* indicates the gap that exists between the designer's architectural descriptions and the actual realizations in the code [GAO95].

Causes for these signs of software aging are, for instance, poor design decisions and changes that damage the architecture or the lack of conformance between implementation and intended architecture. The effect of the signs is that software productivity and quality continue to fall short. The costs for fixing problems and adapting the system to changing and new requirements explode and the 'life" of a software system is shortened.

2.5 Controlling Software Evolution

To overcome or avoid the negative effects of software aging is by placing change in the center of the software development and maintenance process. We must advance beyond the engineering metaphor of current software development and provide more and better support for software change and evolution. This includes support for analyzing software systems and its evolution and support for controlling and executing changes. The focus of this thesis is on the analysis and control of software evolution. The aspect of support for the execution of changes is out of scope.

Architecture recovery is a reverse engineering technique that addresses the analysis issues providing the foundation for controlling evolution. According to Jazayeri *et al.*

> *Architecture recovery* refers to the techniques and processes used to uncover a system's architecture from available information [JRvdL00].

Architecture recovery addresses the extraction of architectural views that represent the system from the perspective of *structural* and *evolutionary* concerns. Questions to be answered are, for instance:

- Which are the modules that constitute the system? What are the relationships between these modules? Which architectural styles and patterns have been used?
- Are there structures in the implementation that indicate architectural mismatch, architectural erosion, or architectural drift?
- How does the architecture, design, and the implementation evolve over time? Which are the change prone modules?
- Are there hidden dependencies in the implementation that cause change propagation?
- Are there trends of decay in the architecture, design, and implementation?

The answers point the engineers to the locations in the implementation – modules and relationships that cause the erosion, mismatch, drift, and decay. Based on this knowledge, the system developers and architects can plan and execute restructuring that resolve the shortcomings. This results in a number of benefits that are gained through architecture recovery and software evolution analysis, such as:

- *Architectural analysis and evaluation*: Breaking down a large software system into smaller, more manageable units facilitates understanding of the behavior and quality attributes.
- *Maintenance and evolution*: A better understanding of the architecture facilitates to asses, plan, and execute changes to the system more effectively.
- *Evolution traceability*: By extracting and analyzing architectural information from different releases it is possible to trace the system evolution.

- *Conformance-checking*: Resulting architectural views can be used to check whether the "as-designed" architecture conforms to the "as-built" architecture.
- *Reuse*: Higher-level views on architectural elements (*e.g.,* modules) show their interfaces and facilitate the identification of elements that can be re-used when developing a new or re-engineering an existing software system.
- *Product line architecting*: The idea of a product line or product family is in maximizing re-use of software artifacts and minimizing costs for developing single products. Architecture recovery helps to recover the architectures of single products and product families to design and maintain the reference architecture, isolate the variable parts, and to generalize software components [PGG$^+$03].

In order to extract the architectural views and perform the architecture and software evolution analysis we have to cope with a number of challenges, such as:

- What are the signs of architectural erosion, mismatch, drift, and decay, and how can they be tracked down to the implementation?
- How can hidden dependencies in a system that complicate and hinder its evolution be discovered? How can existing analysis methods be adapted, revised, or enhanced to enable this?
- How can the plethora of software data (several source code releases, modification and bug data, release data) be filtered and visualized? Which are the most suitable effective visualization models and techniques for that?

In this thesis, we address these challenges with a focus on the hidden dependencies, the extraction and integration of software data, the abstraction of these data to higher-level views, and the visualization and analysis of these views.

CHAPTER 3

RELATED WORK

In this chapter, we review the state-of-the-art in the research field of software evolution. Numerous research groups have started research projects in software evolution: the whole established community of reverse engineering, re-engineering, and program understanding has acknowledged that *evolution* is indeed the umbrella of their research activities.

3.1 SOFTWARE EVOLUTION

Lehman and Belady's *Laws of Software Evolution* [LB85] establish that as systems evolve, they become more complex, and consequently more resources are needed to preserve and simplify their structure. They also establish that *successful* systems (*i.e.,* used in a real-world environment) *must change*, or become progressively less useful in that environment. Lehmann, Perry and Ramil explored the implication of evolution metrics on software maintenance [LPR98], [LPR$^+$97]. They used the number of modules to describe the size of a version and defined evolutionary measurements which take into account differences between consecutive versions.

Various well-known techniques exist to make systems more flexible in the face of change. Many *design patterns*, in particular all of those in the original Design Patterns book [GHJV95], are intended to increase flexibility, however at the cost of increased complexity. Software architectures [SG96] establish rules that govern how a system grows and evolves. Unfortunately, certain kinds of unanticipated change can break the assumptions of an architectural style (for example, pipeline architectures intended for batch processing can be hard to migrate to a fully interactive setting).

There are several approaches that analyze the influence of changes in an evolving software system: Burd and Munro analyzed the influence of changes on the maintainability of software systems by defining a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls [BM99]. Gold and Mohan defined a framework to understand the conceptual changes in an evolving system [GM03]. Based on measuring the detected concepts, they could differentiate between different maintenance activities. In terms of change effects, impact analysis approaches (*e.g.,* [Arn96], [LR03], [CFV99]) attempt to determine, given a point in the source code involved in a modification task, all other possible points in the code that are transitively dependent upon this seed point. Many of these approaches are based on static slicing (*e.g.,* [GL91]) or dynamic slicing (*e.g.,* [AH90]).

Zimmermann *et al.* placed their analysis at the level of entities in a meta-model [ZWDZ04]. Their focus was to provide a mechanism to warn developers that: "Programmers who changed these functions also changed ...". Further, Ying *et al.* applied data mining techniques to the change history of the code base to identify change patterns to recommend potentially relevant source code for a particular modification task [YMNCC04].

Cubranic and Murphy introduced the Hipikat [CM03] approach. Hipikat uses project information to provide recommendations for a modification task. Project information comprises a number of different sources, including the source code versions, modification task reports, newsgroup messages, email messages, and documentation. The focus of Hipikat is on providing recommendations for relevant project artifacts to developers who are evolving a system whereas the focus of ArchView is on software architecture and evolution analysis.

Fenton and Ohlsson reported on an experiment with two commercial software systems in which they tested a range of basic software engineering hypotheses relating to: The Pareto principle of distribution of faults and failures; the use of early fault data to predict later fault and failure data; metrics for fault prediction; and benchmarking fault data [FO00]. They found no evidence to support the hypothesis that size and complexity of modules are good predictors of either fault-prone or failure-prone modules. Their results showed that those modules which are the most fault-prone prerelease are among the least fault-prone postrelease, while conversely, the modules which are most fault-prone postrelease are among the least fault-prone prerelease. ArchView also addresses the relation between size and complexity to fault and change proneness but focuses more on the techniques to obtain, integrate, and visualize views on the implementation and its evolution.

Taking into account different releases of a system Lanza *et al.* introduced the Evolution Matrix [Lan01] that represents the history of classes. Based on size metrics tracked over a number of releases they defined a specific vocabulary to categorize classes (*e.g.,* Pulsar, Supernova, White Dwarf, etc.). Similarly, Girba *et al.* described an approach that based on summarizing source code metric values of several releases facilitates identifies change prone classes [GDL04].

Gall *et al.* analyzed the history of changes in software systems to detect the hidden dependencies between modules [GHJ98]. Their analysis was at the file level, rather than dealing with the real code and considered release and version information of software units (modules, files, etc.) as well as modification reports [GJKT97]. In [GJR99] Gall *et al.* described a visualization approach to allow an engineer to quickly grasp the evolution "nature" of modules, differentiating stable from more volatile ones with respect to change and growth trends.

Fischer *et al.* extended the concept of logical coupling and defined a filtering mechanism and a data scheme for such an integration in [FPG03b]. The data scheme is the initial version of the Release History Database (RHDB) that we adapted for the ArchView approach. In [FPG03a] Fischer *et al.* analyzed the evolution relation to bug reports to track the hidden dependencies between system features. By instrumenting the code the authors showed how features are scattered over the project tree and how features are logically coupled over releases. An extension of this approach with a number of specific visualization techniques is described in [FG04]. This approach allows an engineer to uncover hidden dependencies among different features over many releases.

Based on CVS data Krajewski *et al.* discovered change couplings: developers checking in and out files within certain periods of time and the relationship of these files discovered dependencies that are difficult to detect by other means and pointed to several bad code smells [FBB$^+$99] by means of visualizations using JGraph [GJK03].

In terms of re-engineering activities, Demeyer *et al.* [DDN02] propose practical assumptions to identify where to start a re-engineering effort: working on the most buggy part first or focusing on the client's most important requirements.

3.2 Architecture Recovery

Research on architecture recovery spans a wide area of activities: approaches, such as Bookshelf [FHK$^+$97], Dali [KC99], Bauhaus[1] or Rigi [MK88], [Won98] follow the Extract-Abstract-View Metaphor described in [EKRW02]. They focus on the creation of condensed high-level views to facilitate program understanding. Most tools differ in the underlying fact extraction technique, in the methods and details of fact representation, and in the analysis and visualization techniques.

Murphy and Notkin proposed a reconstruction technique based on reflexion models [MNS01]. The user starts with a structural high-level view model that is mapped against the source code. The result of the mapping is a reflexion model that shows the differences between the developer's high-level and the recovered model. Koschke and Simon have extended the original reflexion models to hierarchical architecture models [KS03].

Similar to reflexion models Robillard and Murphy proposed an approach using Concern Graphs that abstract the implementation specific details of a concern and makes explicit the relationships between different parts of the concern. They implemented their approach in the Feature Exploration and Analysis Tool (FEAT) that allows a developer to manipulate a concern representation extracted from a Java system, and to analyze the relationships of that concern to the code base.

Cremer *et al.* [CMW02] described a graph-based approach for re-engineering COBOL programs. Since the focus of their work is on source code transformation, their visualizations are very detailed but do no support abstractions to higher levels.

In [EKRW02] Ebert *et al.* introduced GUPRO which is an integrated workbench that supports program understanding of heterogeneous systems on arbitrary levels of granularity. However, it does not concentrate on the abstraction of higher-level views from source code. Moreover, GUPRO supports program understanding via textual information, but it does not include graphical representations to depict its findings.

[1]http://www.iste.uni-stuttgart.de/ps/bauhaus

Extracting architectural properties from large open source systems such as the Mozilla system has been addressed by Godfrey *et al.* [GL00]. Their work relied on PBS [FHK$^+$97] which is a reverse engineering workbench containing the Relational Algebra tool Grok [FH00]. The new version of the PBS workbench is SWAGkit[2]. This toolkit concentrates on extracting higher-level views from C/C++ source code. But, both PBS and SWAGkit do not consider the visualization of metrics to characterize abstracted entities and relationships leading to more condensed and comprehensible views.

The SAR method described by Krikhaar [Kri99] concentrates on creating higher-level views on the architecture. The approach is based on Relational Partition Algebra [FKvO98] and defines a process for selecting the information sources from which higher-level views are abstracted. The architecture recovery approach of ArchView is similar to the SAR method but also takes into account evolution.

Riva proposed a view-based architecture reconstruction approach named NIMETA [Riv04]. Similar to Krikhaar the approach is based on relational algebra. NIMETA emphasizes the scrupulous selection of architectural concepts and architecturally significant views that are reflecting the stakeholders' interests.

Other works concentrate on diverse coupling metrics: in [BDW99a] Briand *et al.* discuss a unified framework for coupling measurement in object-oriented systems based on source model entities. Based on these metrics they verified in [BDW99b] the coupling measurements on file level using statistical methods and change coupling information based on "ripple effects" [YCM78]. In [ABF04] Arisholm *et al.* describe how coupling can be defined and measured based on dynamic analysis of systems. This recent study shows that some dynamic coupling measures are significant indicators of change proneness and that they complement existing coupling measures that are based on static analysis.

In terms of analysis of evolution history data, Zimmermann *et al.* inspected release history data of several software systems for change coupling between source code entities [ZDZ03]. They conclude that augmentation of architectural data with evolutionary information could reveal new otherwise hidden dependencies between source code entities. Even though a number of other work used release history data as well, a detailed evaluation of the correlation between source model entities and the properties of change coupling is still missing.

3.3 Information Visualization

Information visualization is defined as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition." [CMS99]. It derives from several communities. Starting with Playfair (1786), the classical methods of plotting data were developed. In 1967, Jacques Bertin, a French cartographer, published his theory in *the semiology of graphics* [Ber74]. This theory identifies the basic elements of diagrams and describes a framework for their design. Edward Tufte published a theory of data graphics that emphasized maximizing the density of

[2]http://swag.uwaterloo.ca/swagkit

useful information [Tuf90], [Tuf97]. Both Bertin's and Tufte's theories have been influential in the various communities that led to the development of information visualization. They mainly addressed issues of how certain types of data could best be visually rendered on paper or on screen.

The goal of information visualization is to *visualize any kind of data*. It must be emphasized that most information visualization systems involve using computer graphics which render the data using 2D- and/or 3D-views of the data. Applications in information visualization are so frequent and common, that most people do not notice them: examples include meteorology (weather maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc.

According to Ware [War00], visualization is the preferred way of getting acquainted with and navigating large data pools. In the 1980s, thanks to increased performance of computers, researchers started to develop tools and methodologies to interactively display large amounts of information on the screen. Stasko *et al.* [SDBP98] give an excellent overview over this pioneering work.

In the more specific field of reverse engineering, visualization soon proved to be an effective technique, yielding many tools such as Rigi [MK88], [Won98] and SHriMP (Creole) [SM95].

Visualization has also proven to be a key technique of research in software evolution, mainly due to the huge amounts of information that need to be processed and understood. Riva *et al.* analyzed the stability of the architecture [GJR99], [Jaz02] by using colors to depict the changes over a period of releases. Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [VRD04]. Similar to [GJR99], Wu *et al.* describe an Evolution Spectrograph [WSHH04] that visualizes a historical sequence of software releases.

Grosser, Sahraoui and Valtchev applied Case-Based Reasoning on the history of object-oriented system as a solution to a complementary problem to ours: to predict the preservation of the class interfaces [GSV02]. They also considered the interfaces of a class to be the relevant indicator of the stability of a class. Sahraoui *et al.* employed machine learning combined with a fuzzy approach to understand the stability of the class interfaces [SBLE00].

Source Viewer 3D (sv3D) is a tool that uses a 3D metaphor to represent software systems and analysis data [MMF03]. The 3D representation is based on the SeeSoft pixel metaphor [BE96] and extends it by rendering the visualizations in a 3D space.

We emphasize the fact that many researchers view information visualization as a mere way to present their data, while we are convinced that an (interactive) visualization itself is a central part of evolution research, due to the very large amounts of information. In his thesis Lanza followed this principle and developed the CodeCrawler tool [Lan03]. The tool integrates a number of pre-defined views that facilitate coarse-grained, fine-grained, and evolutionary software visualization. The ArchView approach also follows these principles and introduces additional and new views on the implementation of a software system and its evolution.

CHAPTER 4

THE ARCHVIEW APPROACH

This chapter introduces the ArchView architecture recovery and analysis approach with a focus on the ArchView process and its key principles and features.

4.1 INTRODUCTION

The idea to re-construct the architecture of software systems from available information sources is not new and there exists a number of approaches that concentrate on this issue. However, our experience with the existing approaches showed that actually there is no sufficient solution to this problem.

The major challenge of architecture recovery is in abstracting reasonable higher-level views from lower-level information. We claim reasonable to be *condensed* higher-level views that highlight the interesting elements and relationships and hide information of minor interest. For instance, when analyzing the coupling between modules we want to see or point out the strong coupling relationships and hide the weak ones. To create such views we need techniques to:

- Extract facts from available data source;
- Aggregate and abstract information;
- Filter the interesting information; and
- Visualize the results in a way that facilitates reasoning about the architecture and its evolution.

With respect to these techniques existing approaches lack: 1) data to analyze evolutionary aspects of a software system; and 2) techniques to visualize multivariate data of n software releases.

Addressing these issues we introduce the ArchView approach, which in addition to source code, considers modification and problem report data. Data about these reports is available from versions and bug reporting systems, such as CVS[1] and Bugzilla[2]. The primary idea of ArchView is to integrate this data with the data models extracted from n source code releases. Based on the integrated data model several new evolution metrics are computed. They concern: 1) problem and modification report metrics; and 2) source code size, complexity, and coupling metrics tracked over a number of n releases.

The first set of metrics characterizes the frequency of problems and modifications of implementation units (*e.g.,* software modules, source files). The second set of metrics facilitates the visualization of the *trend* of measured metric values. With the trend information the user is able to spot changes that, for instance, led to an improved or degraded design and implementation. For example, from release x to release y the two modules A and B have been decoupled by removing the cyclic dependency relationship.

Concerning the visualization ArchView uses the polymetric views techniques [LD03] to map the measured metric values to graphical attributes. Furthermore, ArchView provides an extension to the polymetric views that facilitates the visualization of multiple metrics of up to n releases. Using these two techniques our approach comes up with a number of new polymetric views that provide insights into the implementation and evolution of a software system.

In the following sections we present the process and the key principles of the ArchView approach. The detailed descriptions of the different techniques used by ArchView are provided in the subsequent chapters.

4.2 Change-Prone Modules

A primary objective of ArchView is to point out the change-prone modules and heavy coupling dependencies. To find these change-prone entities we first have to define: What is a change-prone module?

A *software module* is an implementation unit that stems from the decomposition of a software system into manageable units [CBB+02], [BCK03]. Depending on the granularity level of the system decomposition a software module is implemented by a single source file or class but may also be implemented by a set of source files or classes. For instance, the Mozilla source code is organized in more than 90 software modules whereby each module refers to a set of files grouped in several source code directories. Packages are similar to source code directories and also are used to group classes to an implementation unit that represents a software module. The focus of this thesis is on software modules as groups of source files and on single source files.

The quality of an implementation of a software module can be measured by software metrics. They quantify the size, computational complexity, modification and problem frequency. For the size of modules we primarily use the number of lines of code (LOC) and the number of functions

[1]https://www.cvshome.org
[2]https://bugzilla.mozilla.org

(NFM) metrics. The computational complexity of a module is determined by the McCabe cyclomatic complexity [McC76] and the Halstead metrics [Hal77]. These metrics provide indications for the maintainability of software modules [Sab01, CALO94].

In addition to the size and complexity metrics ArchView computes evolution metrics that concern the number of modifications and reported problems during a specified observation period (*e.g.,* between two releases). Based on these metrics we refer to a *change-prone module* as:

> A software module that relative to the other modules is large in size, computational complex, and has more modification and problem reports assigned to it.

Change prone modules are, therefore, implementation units that compared to the other modules were involved in maintenance and evolution activities more often. In the refactoring community change prone modules and coupling relationships are referred to as *Bad Smells* as described by Fowler *et al.* [FBB+99] and Kerievsky [Ker05].

Another aspect of decomposing a system concerns the dependency and in particular the usage relationships between modules. Typically, the behavior that belongs together is put into one module (*e.g.,* class) and can be accessed by other modules through its interface [Par72]. If other modules depend on the services provided by this module then the modules are coupled.

On the source code level these *uses* relationships are function calls, variable accesses, and also type references between modules. Heavy coupling between two modules occurs if relative to other coupling relationships the number of source code relationships between two modules is high or exceeds a specified threshold. The effect is that when modifying one module the coupled modules also have to be modified. Heavy coupling dependencies contribute to Bad Smells in the code, the design, and the architecture.

In addition to source code coupling ArchView takes into account change coupling relationships. A change coupling relationship between two modules originates from changes in both modules that were committed by the same developer in the same commit transaction [GJK03]. We indicate heavy change couplings by the number of such pairwise commits that relative to other change couplings is high or exceeds a specified threshold.

The approach followed by ArchView is to identify and highlight these change prone modules and heavy coupling dependencies by using graphical visualizations.

4.3 ArchView Process

The ArchView approach follows an iterative and interactive architecture recovery process that is depicted by Figure 4.1. The figure shows the different process phases/steps of ArchView together with the information sources taken into account and the flow of information. The different process phases of ArchView are:

1. Fact Extraction.
   The fact extraction phase is concerned with extracting the facts from available information

sources. ArchView takes into account different source code releases (R1, R2, ..., Rn) and modification (CVS data), as well as problem report data (Bugzilla data). The latter data sources enrich the source code model data by release history data. The various data sources are retrieved, processed and extracted facts are stored in the ArchView repository.

2. Data Integration.
   The task of the data integration step is to create and maintain a consistent data repository. For instance, the different tools and techniques used to extract the facts from source code and configuration management data produce separate data files. Similarly, the iterations of the abstraction step as well as the iterations in the architecture analysis result in different data files. The ArchView data integration tool processes these separate data files and integrates them into the ArchView repository [PFG05], [APGP05].

3. View Abstraction.
   In the view abstraction phase, higher-level views are abstracted from lower-level information. The ArchView abstraction algorithm facilitates information abstraction onto different levels. In this thesis we focus on the level of software modules and source files. Resulting views are expressed in terms of high-level entities, abstracted coupling relationships, and metrics that are computed during abstraction. Resulting views together with the links to abstracted lower-level information are stored in the ArchView repository. From there the views are retrieved for further abstraction iterations or analysis and visualization [PFG05], [PFJG04].

4. Visualization & Analysis.
   In the visualization phase abstracted views and measured metric values are presented to the user. Computed views are the basis for the analysis of the current implementation and its evolution. To facilitate analysis the ArchView visualization techniques concentrate on highlighting the change prone modules and heavy coupling relationships [PGFL05].

The ArchView process is iterative and interactive. The user is in the driving seat and controls the abstraction of architectural views, their visualization and analysis. Support is provided by ArchView in the form of tools and predefined views. Tools aid in automating the extraction, integration, and abstraction of data models. Furthermore, there are tools that facilitate information filtering and composition of views. The set of pre-defined views facilitates the analysis of implementation and evolution specific aspects. This set can be re-used in other analysis projects and extended by additional views.

4.4 Key Features

The key features of the ArchView approach and the major benefits gained by a feature are:

- Considering source code model data of several releases.
  Size, complexity and coupling metrics are computed of different releases. This enables the

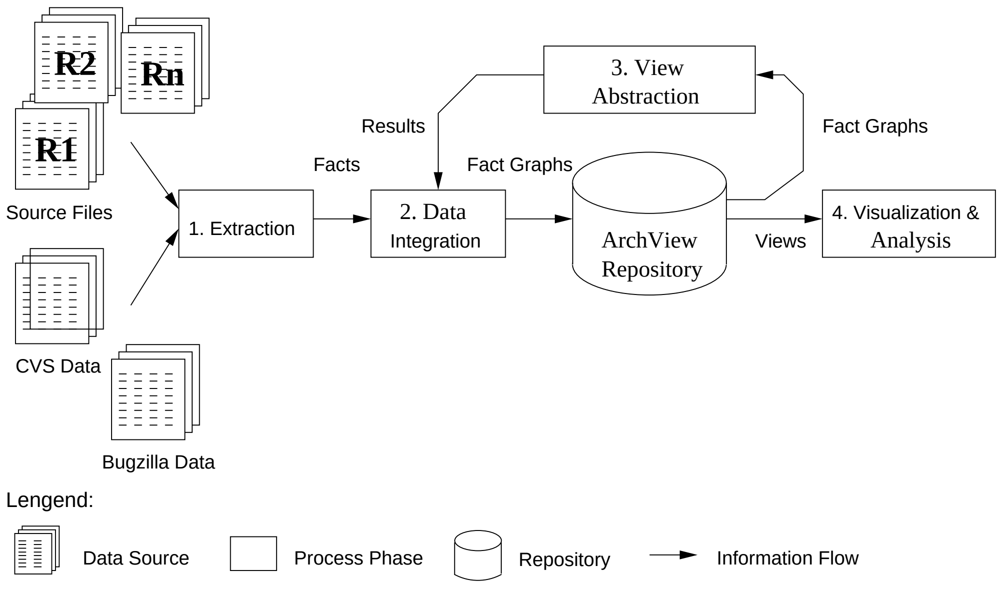


Figure 4.1: ArchView Process with data sources (on the left), the ArchView repository (in the center) and the different process phases.

visualization and analysis of metric trends of implementation units and coupling dependencies.

- Modification and problem report data.
  Evolution metrics are measured whereby different observation periods can be selected. This allows the analysis of modification and problem behavior during different periods. The user can concentrate on a particular observation period and compare it with other observation periods.

- Change coupling dependencies.
  In addition to source code ArchView computes change couplings between source files and software modules. They indicate the propagation of changes between modules in the past caused by shortcomings in the design.

- Integration of data models.
  The integration tool links the different data models to one model that facilitates the navigation between them. For example, ArchView allows the user to navigate between the models of the different source code releases and the modification and problem report data.

- Abstraction of data models.
  The abstraction of data models condenses lower-level source code entities and relationships to higher-level implementation units (*e.g.,* software modules) and relationships. Lower-level relationships, such as function calls are aggregated and established between the higher-level implementation units.

- Extended polymetric views visualization.
  For the coarse grained analysis of the implementation of one release we apply polymetric views. Based on this technique we specify a set of additional views that take into account coupling and evolution metrics. We extend the polymetric views technique to visualize measured values of multiple metrics of several releases in one view. They enable the user to analyze the trends of size, complexity, coupling, and evolution metrics and spot strong changes that indicate improvements or degradations in the implementation and design.

The techniques, algorithms and methods applied to realize each key feature are described in the following chapters.

4.5 Module View Example

The output of the ArchView process are graphical views in which nodes represent the modules and edges represent the coupling dependencies between modules. Figure 4.2 represents an example of a graph generated with ArchView. In this example the focus was on analyzing the inheritance structure of Mozilla's content and layout implementation and its evolution.

The nodes depict the seven software modules of Mozilla that implement Mozilla's content and layout handling. The edges of the graph represent the inheritance relationships between modules and show which module inherits behavior from which other modules. The width of the edges indicates the number of aggregated inheritance relationships. The thicker the edge the more and the stronger the inheritance relationship is between two modules. Each node is drawn as a diagram that shows 9 metric values computed from 7 Mozilla releases. In this example, the metrics indicate the size in number of classes (NOC) and the fan-in (IHNAR-in, IHFan-in, IHNCE-in, IHNR-in) and fan-out (IHNAR-out, IHNCE-out, IHFan-out, IHNR-out) metrics of module inheritance. Basically, the fan-in metrics denote the number of classes that other modules inherit from a module. The fan-out metrics denote the number of classes that a module inherits from other modules. An explanation of these metrics is presented in Chapter 6.

In the following we provide an interpretation of this view and point out a number of key findings that we can get from such views:
First of all, we gain information about the inheritance structure of the content and layout modules. We see that, except the `MathML`, all modules inherit behavior from the `DOM` module. Hence, `DOM` clearly is a super module. The other module from which behavior is inherited is `NewLayoutEngine`.

Regarding the edges the graph shows that `NewLayoutEngine` and `XPToolkit` inherit more behavior from `DOM` than do other modules. Furthermore, the graph highlights a "Bad Smell" in the inheritance structure. This is due to the cyclic inheritance between
`NewLayoutEngine` and `DOM`. Apparently, there is a number of classes contained by the
`NewLayoutEngine` module that should be moved to the `DOM` module.

In addition to the structure, the graph represents detailed measurements of size, inheritance fan-in fan-out metrics of several releases. For instance, it points out the `DOM` module as the largest

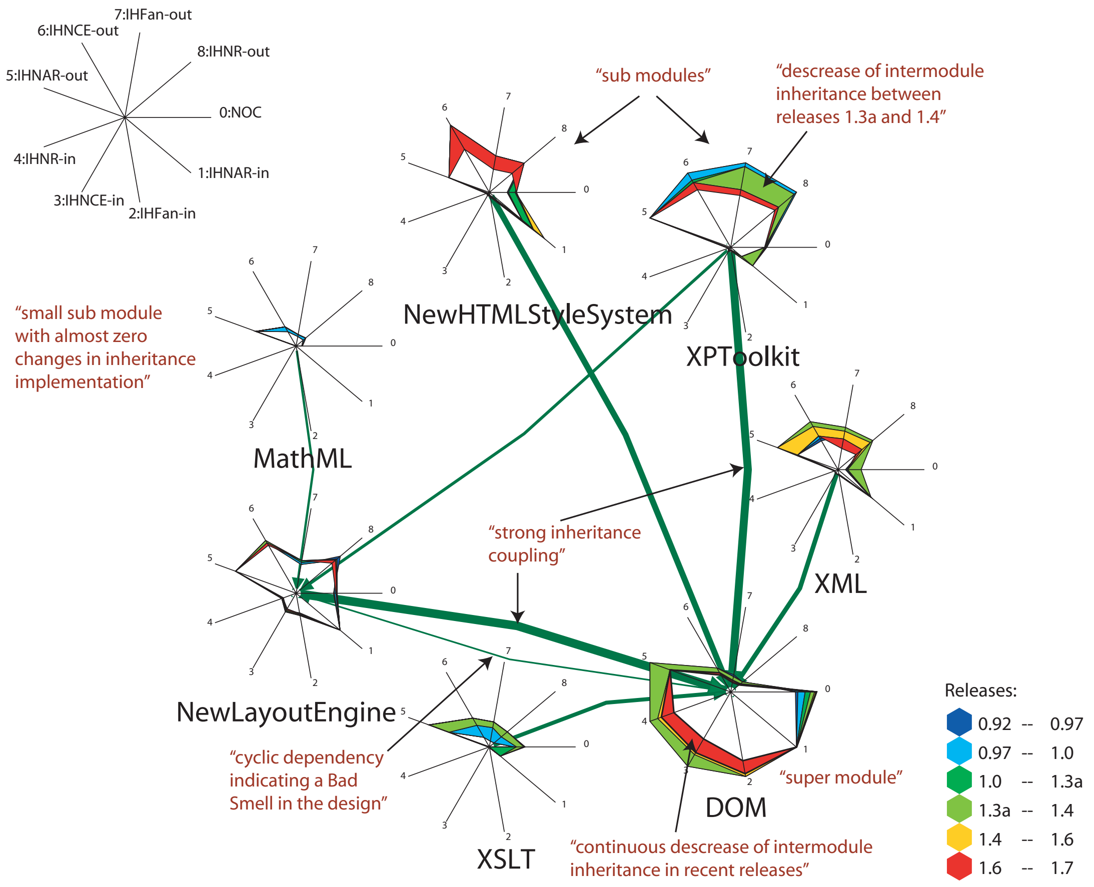

7:IHFan-out
6:IHNCE-out
8:IHNR-out
5:IHNAR-out
0:NOC
4:IHNR-in
1:IHNAR-in
3:IHNCE-in
2:IHFan-in
"sub modules"
"descrease of intermodule inheritance between releases 1.3a and 1.4"
NewHTMLStyleSystem
XPToolkit
"small sub module with almost zero changes in inheritance implementation"
MathML
XML
"strong inheritance coupling"
NewLayoutEngine
"cyclic dependency indicating a Bad Smell in the design"
XSLT
DOM
"super module"
"continuous descrease of intermodule inheritance in recent releases"
Releases:
0.92 -- 0.97
0.97 -- 1.0
1.0 -- 1.3a
1.3a -- 1.4
1.4 -- 1.6
1.6 -- 1.7

Figure 4.2: Source code coupling evolution view on Mozilla content and layout modules with metric values of in-coming and out-going inheritance relationships. Values are of 7 releases from 0.92 to 1.7. Edges denote aggregated inherits relationships taken from release 1.7 filtered using a threshold of 5 for RNAIH.

module with the highest number of classes and `MathML` as the smallest module with a rather small number of classes. With regard to the fan-in and fan-out metric values it depicts the `DOM` module as a super module. This is indicated by the high fan-in values and low fan-out values. Diagrams of other modules, such as of the `NewHTMLStyleSystem` and `XPToolkit` nodes classify these modules as sub modules. They show high fan-out values but low fan-in values because they solely inherit behavior from other modules (*i.e.,* `DOM` and `NewLayoutEngine`).

Furthermore, the graph shows metric values of several releases that allows us to retrieve information about the progression of metric values and spot the strong changes in past releases that affected the implementation. ArchView indicates strong changes of metric values by large polygons. For instance, the size values represented by the `DOM` diagram show a large module to which continuously new classes where added. In relation to the `DOM` module the `MathML` module did not change a lot. Furthermore, the relative large, green polygon in the `XPToolkit` diagram clearly highlights a big change between the releases 1.3a and 1.4. In this observation period the number of class inheritances of the `XPToolkit` was reduced from 53 to 40 inheritance relationships. This metric values further decreased from release 1.6 to 1.7 as indicated by the red polygon. Apparently, there is a trend of reducing the intermodule inheritance. Another interesting trend is depicted by the green and red polygons of the `DOM` module. They show a continuous decrease in the inheritance of `DOM` behavior in the recent releases. In contrast, the metric values shown by the `MathML` node indicate a small sub module with almost zero changes during the seven releases.

This is a representative number of key findings that we can gain from the higher-level views created by ArchView. Further views are presented and interpreted in Chapter 8.

4.6 Summary

The chapter introduced the ArchView architecture recovery and analysis approach including the process, the key concepts and features. Basically, ArchView follows the Extract-Abstract-View Metaphor [EKRW02]. In contrast to existing approaches, ArchView considers modification and problem report data that are integrated with extracted source code models. Based on this data model ArchView computes abstracted views as well as source code, modification, and problem report metrics.

The polymetric views visualization technique is based on mapping metric values to graph attributes. ArchView extends an existing technique to provide more detailed views that highlight the change prone modules and heavy coupling relationships.

The integration of modification and problem report data as well as the extension of the visualization techniques present the major improvements to existing architecture recovery approaches.

CHAPTER 5

BUILDING THE ARCHVIEW REPOSITORY

This chapter describes the pre-processing of information sources including source code and release history data to build the ArchView Repository.

5.1 INTRODUCTION

For the reconstruction of higher-level views ArchView takes into account the source code of several releases as well as data from versions and bug reporting systems.

The extraction phase is concerned with pre-processing the different data sources to obtain data models. They represent a structured view on the data sources containing the entities and relationships that are subject to analysis. For instance, the source code model contains the source code entities, such as classes, methods, attributes and the relationships between them, such as class inheritance, method calls, attribute accesses.

The integration phase links the different data models to one integrated model that contains the relevant information about the implementation, problems (or bugs), and modifications. The integrated data model is stored in the ArchView repository that serves as basic input to the subsequent ArchView abstraction, visualization, and analysis steps.

5.2 THE E-FAMIX META MODEL

The ArchView repository is the central data storage of our approach that holds the different extracted and integrated data models as well as analysis results. We implemented it with a relational database management systems (*i.e.,* MySQL[1]) that stores:

- source code models of different source code releases;

[1]http://www.mysql.com

- configuration management data (modification and problem reports);
- abstracted views; and
- measured metric values of source code entities and relationships.

For the representation of source code different meta models exist, such as FAMIX of the University of Bern [Sof99], Datrix of Bell Canada Inc. [Bel99], or Bauhaus resource graphs of the University of Stuttgart [Uni05]. All these meta models model the same kind of data – source code – however in slightly different ways. For instance, they use different type identifiers to represent source code entities and relationships. Recent work focused on integrating the different source code meta models into the Dagstuhl meta model [LTP04]. But this is on-going work. With respect to ArchView non of these models takes into account the representation of releases, modifications, and problems.

For the specification of the E-FAMIX meta model we used the FAMIX meta model. FAMIX is a meta model for a language-independent representation of source code of object-oriented programming languages. It provides extension points that can be used to include programming language specific features, such as C++ templates. We used FAMIX for representing the source code of each release and added the types and relationships that are mandatory to represent release, modification, and problem report data. Figure 5.1 depicts an overview of E-FAMIX meta model.

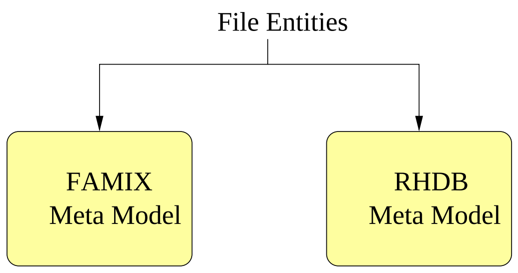


Figure 5.1: E-FAMIX meta model consisting of the FAMIX and RHDB meta model linked by file-entities (overview).

The E-FAMIX meta model consists of two meta models that are linked together by file-entities that are common to both meta models:

- The FAMIX meta model for representing the source code models; and
- The RHDB meta model for representing the configuration management data.

The most important extension to FAMIX is by the RHDB meta model. The latter meta model is used to represent modification report (MR) and problem report (PR) data as obtained from versions systems, such as CVS[2] and bug reporting systems, such as Bugzilla[3].

[2]https://www.cvshome.org
[3]http://www.bugzilla.org

The RHDB meta model is linked to the FAMIX model by using the *file-entity* defined by both models. The principle is straight forward: source files are entities that exist in both data models. On the one hand source files contain the programming language specific entities that are subject to the source code model extraction (FAMIX). On the other hand, source files present the items that are managed by configuration management systems and therefore are also entities of the release history database (RHDB). For instance, modification reports about changes committed to the source code repository are assigned to source files.

A detailed description of the E-FAMIX meta model can be found in the Appendix. The next sections describe the extraction of the different data models.

5.3 Source Code Fact Extraction

The source code fact extraction step is concerned with pre-processing the selected source code releases to FAMIX compliant source code models.

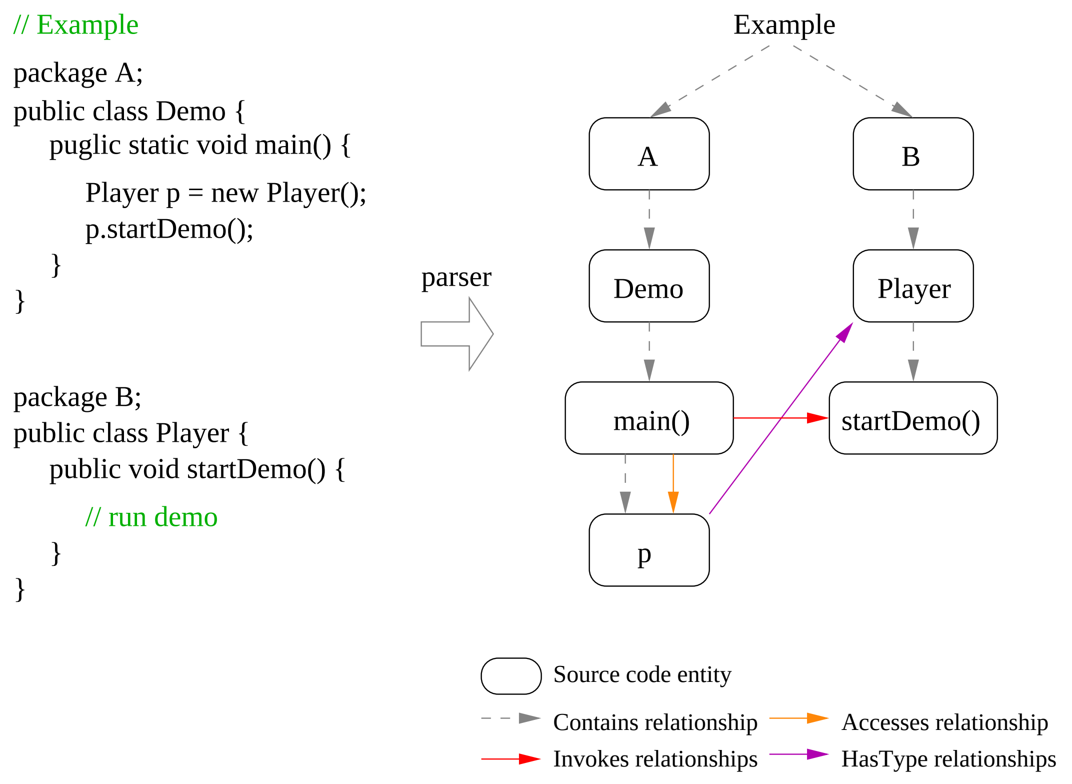
For the extraction of information from source code ArchView primarily applies syntactical analysis tools which are source code parsers. Parsers are programming language dependent and produce an intermediate representation of source code (Abstract Syntax Tree, AST). In the context of reverse engineering the AST is traversed to output the information about the basic source code entities and their relationships. The set of extracted source code facts is referred to as a source code model (SCM). Figure 5.2 shows an example of a source code model extracted from a Java source code snippet.

The parser processes the project containing the Java classes of the Example application. It parses the source code of each source file and extracts facts about contained source code entities and relationships. Facts concerning the example above are: there is the package `A` that contains class `Demo`. `Demo` contains the `main()` method which contains the local variable `p`. Variable `p` references an object of class `Player`. `p` is accessed by the `main()` method to invoke the `startDemo()` method. Furthermore, there is the package `B` that contains class `Player` which contains the `startDemo()` method.

Each source code release of a system constitutes such a parsing-project and results in a source code model. For instance, for the Mozilla case study we checked out the major source code releases from the Mozilla repository and parsed each release with the Imagix-4D tool[4]. For each release we created a project containing the parser configuration and the repository holding the extracted source code model of a release. With our Imagix-4D plug-in we accessed each repository and exported the E-FAMIX compliant source code model.

For the representation of source code models different file formats exist ranging from pure ASCII files to relational databases. Common to most of these representations is the use of directed attributed graphs. Nodes in the graph represent source code entities and edges the relationships. Both, nodes and edges are assigned a set of attributes, such as the name of the entity, the size of a source file, the accessibility specification of methods and attributes, etc.

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



```
// Example
package A;
public class Demo {
    puglic static void main() {
        Player p = new Player();
        p.startDemo();
    }
}

package B;
public class Player {
    public void startDemo() {
        // run demo
    }
}
```

Figure 5.2: Source code model extracted from a Java code snippet.

Output formats used by current reverse engineering tools are, for example, the Rigi Standard Format (RSF) [Won98] or its successor the Graph eXchange Language (GXL) [HWS00]. The latter comes with an XML representation of graph data. Furthermore, there exists a converter from RSF to GXL and vice versa. ArchView uses RSF as an intermediate format to facilitate the application of other reverse engineering tools on extracted source code models.

RSF uses an easy to use tuple format `<relation A B>` to represent nodes, edges, and attributes of graphs. Figure 5.3 depicts the RSF file that represents the source code model extracted from the Java example shown in Figure 5.2. The unstructured RSF file contains the definitions of the existing source code entities (the Root entity is only given for syntactic reasons). Each entity is assigned its entity type as defined by the E-FAMIX meta model (see `type` tuples). The `contains` tuples define the containment (hierarchy) of entities. For instance package `A` contains class `Demo`. The last three tuples define the relationships between source code entities, such as the access to the local variable `p` or the invocation of method `startDemo()` by `main()`.

The results of the source code fact extraction step is a set of source code models (one per release) that are stored in the ArchView repository.

```
type Root Syntactic
type A Directory
type B Directory
type Demo.java File
type Player.java File
#
type A Package
type B Package
type Demo Class
type Player Class
type main() Method
type startDemo() Method
type p LocalVariable
#
contains A Demo.java
contains B Player.java
contains Demo.java Demo
contains Player.java Player
contains A Demo
contains B Player
contains Demo main()
contains Player startDemo()
contains main() p
#
accesses main() p
hasType p Player
invokes main() startDemo()
```

Figure 5.3: RSF file representing the extracted E-FAMIX conform source code model of the Java source code snippet.

5.4 Release History Data

In addition to source code, ArchView takes into account configuration management data with focus on version data, modification and problem reports. The primary data sources for this kind of information are the repositories managed by version control systems such as the Concurrent Versions System (CVS[5]) and bug tracking systems, such as Bugzilla[6]. Currently, ArchView concentrates on CVS and Bugzilla but future work is concerned with providing support for other versions and bug tracking systems, such as Subversion[7] the successor of CVS, IBM Rational's

[5]https://www.cvshome.org
[6]https://bugzilla.mozilla.org
[7]http://subversion.tigris.org

ClearCase[8], or Microsoft's Visual SourceSafe[9].

5.4.1 CVS

CVS is designed to handle revisions of textual information by storing delta's between subsequent revisions in the repository. Binary files can be stored in the repository as well, but they are not considered by ArchView.

Revision numbers

Typically, version control systems distinguish between version numbers of files and software products. Concerning files these numbers are called *revision numbers* and indicate different versions of a file. In terms of software products they are called *release numbers* and indicate the releases of a software product.

Each new version of a file stored in the CVS repository receives a unique revision number. After an update of a file and a commit of the changes to the CVS repository the revision number of each affected file is increased by one. Because some files are more affected by changes than others these files have different revision numbers in the CVS repository.

A release represents a snapshot on the CVS repository comprising all files realizing a software system whereby the files can have individual revision numbers. Whenever a new version of the software system is released a symbolic name (*i.e.,* tag) indicating the release is assigned to the revision numbers of current files. The relation *symbolic name - revision number* is stored in the header section of every tagged file and appears also in the header section of CVS log files.

Revision numbers and CVS tags are the information that are needed to checkout the different source code releases. For more information on CVS internals we refer the reader to the CVS manual [Fre03].

Version Control Data

For each working file in the repository CVS generates version control data stored in log files. From there log file information can be retrieved by issuing the `cvs log` command. The specification of additional parameters allow for the retrieval of information about a particular file or a complete directory. Figure 5.4 depicts an example log file taken from the Mozilla project showing version data of the source file `nsCSSFrameConstructor.cpp` as it is stored by CVS. Basically, a log file consists of several sections, each describing the version history of an artifact (*i.e.,* file) of the source tree. Sections are separated by a line of '=' characters. For the population of the release history database (RHDB) we take the following properties into account:

RCS file - The path information in this field identifies the artifact in the CVS repository.

[8] http://www-306.ibm.com/software/awdtools/clearcase

[9] http://www.microsoft.com/ssafe

```
RCS file: /cvsroot/mozilla/layout/html/style/src/nsCSSFrameConstructor.cpp,v
Working file: nsCSSFrameConstructor.cpp
head: 1.804
branch:
locks: strict
access list:
symbolic names:
        MOZILLA_1_3a_RELEASE: 1.800
        NETSCAPE_7_01_RTM_RELEASE: 1.727.2.17
        PHOENIX_0_5_RELEASE: 1.800
        ...
        RDF_19990305_BASE: 1.46
        RDF_19990305_BRANCH: 1.46.0.2
keyword substitution: kv
total revisions: 976;   selected revisions: 976
description:
----------------------------
revision 1.804
date: 2002/12/13 20:13:16;  author: doe@netscape.com;  state: Exp;  lines: +15 -47
Don't set NS_BLOCK_SPACE_MGR and NS_BLOCK_WRAP_SIZE on ...
----------------------------
...
----------------------------
revision 1.638
date: 2001/09/29 02:20:52;  author: doe@netscape.com;  state: Exp;  lines: +14 -4
branches:  1.638.4;
bug 94341 keep a separate pseudo frame list for a new pseudo block or inline frame ...
----------------------------
....
=============================================================================

RCS file: /cvsroot/mozilla/layout/html/style/src/nsCSSFrameConstructor.h,v
```

Figure 5.4: Example CVS log-file containing the modification reports of the source file nsCSS-FrameConstructor.cpp.

symbolic names - Lists the assignment of revision numbers to tag names. This assignment is individual for each artifact since revision numbers may differ.

description - Lists the *modification reports* describing the change history of the artifact starting from its initial check-in till the current release. Besides the modifications made in the main trunk all changes which happened in the branches are also recorded there. Reports (*i.e.*, revisions) are separated by a number of '-' characters.

- The *revision* number identifies the source code revision (main trunk, branch) which has been modified.
- Date and time of the check-in are recorded in the *date* field.
- The *author* field identifies the person who did the check-in.
- The value of the *state* field determines the state of the artifact and usually takes one of the following values: "Exp" means experimental and "dead" means that the file has been removed.

- The *lines* fields counts the lines added and deleted of the newly checked in revision compared with the previous version of a file.
- If the current revision is also a branch point, a list of branches derived from this revision is listed in the *branches* field (*e.g.,* 1.638.4).
- The following *free text* field contains informal data entered by the *author* during the check-in process.

ArchView uses the RHDB Populator tool [FPG03b] to access the CVS repository and retrieve the modification reports. Basically, the tool traverses through the source tree structure to retrieve the modification reports from the CVS repository on directory basis. Modification reports about "unused" files that have an entry in the CVS repository but are not part of the current checked out version are also captured (*i.e.*, deleted files or files belonging to different products). Each modification report is parsed for the facts mentioned above. Extracted facts are stored in the RHDB.

5.4.2 Bugzilla

In addition to the CVS data, the Populator tool also provides facilities to access the bug reporting system Bugzilla for retrieving the problem reports. Problem reports describe "bugs" that have been identified during execution of the system (*e.g.*, testing) and reported to the developers.

Problem Reports

Problem reports as stored by Bugzilla contain administrative information, such as contact information, mailing addresses, discussion, and information that describes the reported problem. Listing 5.1 lists an example of a problem report derived from the Bugzilla repository of Mozilla.

Listing 5.1: Example of a Bugzilla entry (*i.e.*, problem report) of the Mozilla project.

```
<bug_id>100069</bug_id>
<bug_status>VERIFIED</bug_status>
<product>Browser</product>
<priority>--</priority>
<version>other</version>
<rep_platform>All</rep_platform>
<assigned_to>doe@mozilla.org</assigned_to>
<delta_ts>20020116205154</delta_ts>
<component>Printing: Xprint</component>
<reporter>doe@mozilla.org</reporter>
<target_milestone>mozilla0.9.6</target_milestone>
<bug_severity>enhancement</bug_severity>
<creation_ts>2001-09-17 08:56</creation_ts>
<qa_contact>doe@mozilla.org</qa_contact>
```

```
<op_sys>Linux</op_sys>
<resolution>FIXED</resolution>
<short_desc>Need infrastructure for new print
     dialog</short_desc>
<keywords>patch, review</keywords>
<dependson>106372</dependson>
<blocks>84947</blocks>
<long_desc>
 <who>doe@mozilla.org</who>
 <bug_when>2001−09−17 08:56:29</bug_when>
 <thetext></thetext>
</long_desc>
```

Key information extracted from problem reports include:

- *bug_id*: This ID is referenced in the modification report. Since the IDs are stored as free text in the CVS repository, the information can not be reliably recovered from the change report database.
- *bug_status* (status white-board): Describes the current state of the bug and can be *unconfirmed*, *assigned*, *resolved*, etc.
- *resolution*: Indicates what happened to a bug and can be: empty (“”), has been fixed (*fixed*), is no valid bug (*invalid*), will never be fixed (*wontfix*), etc.
- *product*: Determines the product which is affected by a bug. Examples in Mozilla are Browser, MailNews, NSPR, Phoenix, Chimera, etc.
- *component*: Determines which component is affected by a bug. Examples for components in Mozilla are Java, JavaScript, Networking, Layout, etc.
- *dependson*: Declares which other bugs have to be fixed first, before this bug can be fixed.
- *blocks*: List of bugs which are blocked by this bug.
- *priority*: This field describes the importance and order in which a bug should be fixed. This field is utilized by the programmers/engineers to prioritize their work to be done. The available priorities range from **P1** (most important) to **P5** (least important.)
- *bug_severity*: This field is used to further classify problem reports into blocker, critical, major, minor, trivial, enhancement bugs.
- *target_milestone*: Possible target version when changes should be merged into the main trunk.

Linking CVS and Bugzilla Data

Retrieved modification and problem reports build two separate data sources because CVS and Bugzilla do not provide a built-in mechanism to reference a bug with the modifications to source files that have been carried out in order to fix the bug. These links are mandatory to reflect fixed bugs to changes in source files and vice versa. Therefore, the RHDB Populator tool includes an algorithm to establish the links between modification and problem reports.

The algorithm is based on bug report numbers that manually have been entered in modification reports by the software developers when committing changes to source files. Basically, a link is established whenever a reference (*i.e.,* bug report number) to a problem report is found in a modification report. Problem report numbers in modification reports are searched by regular expressions, *e.g.,* bug #145342. Because these numbers are entered as free text results contain false positive matches as well. To improve data quality all matched numbers are validated using information available with PRs and secondary information such as patches. Details about the RHDB Populator tool and an evaluation of the quality of extracted and linked modification and problem reports are given in [FPG03a] in which the tool has been applied to the CVS and Bugzilla data of the Mozilla project.

5.4.3 Change Couplings

Change coupling between two files A and B originates from changes made to the two files that were committed by the same developer in a single commit transaction. Transactions can span on an arbitrary number of files and can be of arbitrary length. Typically, commit operations take several seconds or minutes.

CVS does not explicitly store the information about transactions, however it indicates them by storing the same log message (*i.e.,* MR) for the involved files. Consequently, change coupling relationships can be reconstructed by analyzing the MRs. Underlying data coming from the reports is retrieved from the repository and represents the input to the algorithm described in Figure 5.5.

The algorithm is based on a sliding time window with two parameters: the maximum length of time that a transaction can last τ_{max} and the maximum distance in time δ_{max} between two subsequent MRs. According to the abstraction algorithm, a modification report mr is included in a transaction T if:

1. The log message lm_2 or the author a_2 differs from the previous MR; and

2. The check-in time t_2 is at most δ_{max} apart from the check-in time t_1 of the previous report; and

3. The check-in time t_2 is at most τ_{max} apart from the start time t_{start} of the transaction.

Otherwise a new transaction T is created and the modification report is assigned to it.

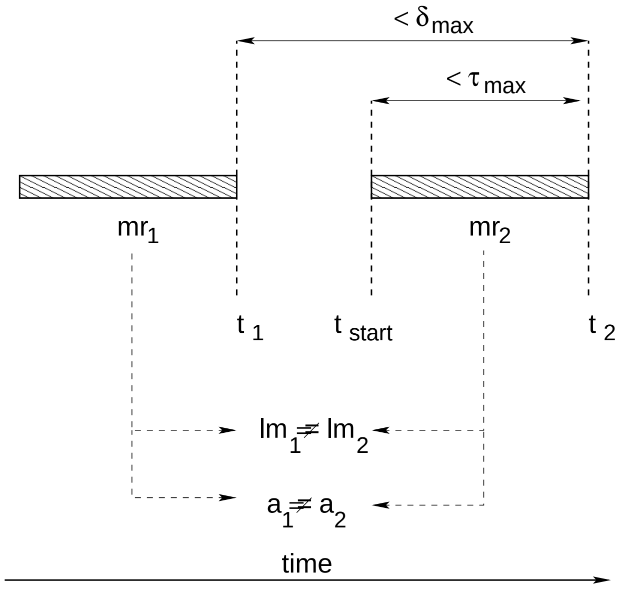


Figure 5.5: When two modification reports (MRs) belong to the same transaction.

Heuristics obtained from our experiences with the Mozilla CVS repository range from 45 to 60 seconds for δ_{max} and 15 to 20 minutes for τ_{max}. Smaller values tend to split transactions, and larger values tend to combine transactions into one. Similar experiences have been reported by related approaches, such as German *et al.* [GHJ04] or Zimmermann *et al.* [ZWDZ04].

Based on the transactions the change coupling relationships between source files are computed. A change coupling relationship is established between two files whenever there exists MRs for the two files that belong to the same transaction. Finally, all the extracted relationships are added to the repository.

The results of presented extraction techniques are one source code model per release and the release history database. Source code models provide the implementation specific facts and the release history database provides facts about problems and modifications of source files.

5.5 Data Integration

The objective of the data integration step is to establish the links between the entities of the different source code and the release history data models. Integration is needed to have a common view on the different data models facilitating the navigation between the different models and the analysis of them.

Currently, the integration of the data models is done on the level of source files. Files are entities that exist in all extracted ArchView models: source files contain the extracted source code entities and furthermore they are the entities managed by configuration management systems. With respect to source files differences between the data models exist because of:

```
$listMR$ := list of modification reports
sort $listMR$ by author, checkinTime
$\tau_{max}$ = 15 min.
$\delta_{max}$ = 45 sec.
$a_1$ := null
$t_1$ := 0
$t_{start}$ := 0
$lm_1$ := null
$listT$ := {}
**foreach** $mr$ **of** $listMR$ **do**
    $a_2$ := author of $mr$
    $t_2$ := checkin time of $mr$
    $lm_2$ := log message of $mr$
    **if** $a_1 \neq a_2$ **or** $lm_1 \neq lm_2$ **or** $t_2 > (t_1 + \delta_{max})$ **or** $t_2 > (t_{start} + \tau_{max})$ **then**
        $T$ := new transaction
        add $T$ to $listT$
        $t_{start}$ := $t_2$
    **end**
    assign $mr$ to $T$
**end**
```

Figure 5.6: Algorithm to reconstruct transactions from CVS modification reports.

- Deleted and added files.
  During maintenance and evolution of a software system source files are added to or deleted from the source base.

- Moved files and subdirectories.
  Source files are moved to a different subdirectory due to restructuring of the source code base. The file entity is the same but the global name of the file has changed.

- Generated files.
  Certain source files are input to compilers that generate a number of corresponding other source files. For instance, the configuration management system contains *.idl files from which the idl compiler generates the corresponding header files that are input to our parsing tool.

The main task of the data integration step is concerned with checking if the source file in one data model exists in the other data model. For these checks ArchView uses the fully qualified name (global file name) of source files and applies the Algorithm 5.7. The algorithm uses three heuristics to map the file identifiers of two data models by: 1) the fully qualified file name; 2) the short file name; and 3) the file name with the file extension replaced. The latter heuristic

```
$scmA$ := load model A
$scmB$ := load model B
$mapping$ := new Mapping
**foreach** $eB$ **of** $scmB$ **do**
    $nameB$ := $eB$.fullName
    $eA$ := $scmA$.queryEntity($nameB$)
    **if** $eA = null$ **then**
        $nameB$ := $nameB$.chopDirectories()
        $eA$ := $scmA$.queryEntity($nameB$)
        **if** $eA = null$ **and** $nameB$.*endsWith(“.h”)* **then**
            $nameB$ := $nameB$.replaceFileExtension(“idl”)
            $eA$ := $scmA$.queryEntity($nameB$)
        **end**
    **end**
    **if** $eA \neq null$ **then**
        $mapping$.insert($eA$, $eB$)
    **end**
**end**
```

Figure 5.7: Algorithm for mapping RHDB and SCM data models by the fully qualified file name.

accounts for files that are generated by a pre-compiler. For example, in the Mozilla case study idl files are contained in the RHDB instead of the generated C/C++ header files that are contained in extracted source code models.

Experiences gained in the case study showed that using these heuristics we establish around 96–99.8% of the links between file entities of the different data models. Details about the precision of the data integration algorithm are presented in Chapter 8.

The result comprises records of mapped identifier pairs that are inserted into the mapping table of the ArchView repository.

5.6 Summary

The ArchView repository is the core data source of the ArchView architecture recovery and analysis approach. It integrates source code and release history data into a common data model. For the representation of this data model we introduced the E-FAMIX meta model and presented corresponding extraction techniques. The E-FAMIX meta model is an extension of the FAMIX meta model by entity and relationship types to represent release history data.

Concerning the extraction we presented the techniques and formats to derive the different source code and release history data models. Based on the latter data model we described the algorithm for the linkage of modification and problem report data. They allow for the navigation of problems down to source code modifications and vice versa. Furthermore, we presented the algorithm used to compute the change coupling relationships between source files.

The integration of the different data models is based on the fully qualified name of source files. Based on this name we presented an algorithm that handles deleted, added, moved, and generated files. The result is a repository that contains an E-FAMIX conform data model of several source code releases, modification, and problem report data. The repository serves the ArchView abstraction, visualization, and analysis phases that are described next.

CHAPTER 6

ARCHITECTURAL VIEW ABSTRACTION

To obtain higher-level views low-level information has to be condensed and abstracted. This chapter introduces the ArchView containment hierarchy model that defines paths for information abstraction and the ArchView view abstraction algorithm.

6.1 INTRODUCTION

The amount of information obtained by the fact extraction phase is high especially when analyzing large and complex software systems. For example, the case study we present in Chapter 8 is of a large open source software system that comprises source code model data of more than 10.000 C/C++source files of seven releases, more than 450.000 modification reports and 250.000 bug reports.

Having a single user to understand all the details is difficult if not impossible but also not always mandatory. In general, browsing and navigating through such large information sources is difficult and time consuming. Users do not have reasonable points (*e.g.*, source code entities) from which to start investigations and analysis because starting points often are not visible in the huge amount of information. Information abstraction is needed that aggregates lower-level information about source code entities and their relationships and reflects them on higher-levels of abstraction.

As mentioned in the related work past and recent research has been concerned with abstracting high-level views from lower-level information, such as source code. Most important in the context of this thesis are the work from Holt [Hol98] and Feijs *et al.* [FKvO98]. They both used relational algebra to aggregate and abstract architectural information. Therefore, the algebraic concepts used by our abstraction algorithm is not new per se. ArchView uses these concepts and extends them in two ways:

- *Metrics*: The ArchView abstraction algorithm computes source code metrics about abstracted entities and relationships. Measured metric values express, for instance, the size of entities (*e.g.,* in terms of the number of contained low-level entities) and the weight of relationships (*e.g.,* in terms of the number of aggregated relationships). They are mandatory to highlight interesting entities as well as to filter information.

- *Information sources*: ArchView takes into account source code data of *several* releases, for instance, to facilitate the analysis of metric trends of source code entities. Furthermore, ArchView takes into account release history data. This data enriches extracted source code model data and allows for analysis of the evolutionary aspects of a software system's implementation (*e.g.,* change coupling between source files).

The next Section introduces the ArchView containment hierarchy model that specifies the paths along which low-level source code and release history data can be abstracted.

6.2 Source Code Containment Hierarchy

The ArchView containment hierarchy model specifies the hierarchy of implementation units and source code entities. The hierarchy stems from decomposing the system into manageable implementation units according to the object-oriented design paradigm. Our hierarchy model is based on the abstraction levels described in Section 2.3 and specified by entities of the E-FAMIX meta model and its relationships that express the containment of entities. Figure 6.1 depicts the model.

According to the decomposition of a system we go top-down and describe the following hierarchical levels:

- *Architectural level:* From the point of view of the implementation the architecture of a system is specified by subsystems and software modules and the relationships between them. A system is decomposed into software modules. According to Clements *et al.* [CBB+02] we refer to a software module as an implementation unit of software that provides a coherent unit of functionality. Modules present a code-based way of considering the system [BCK03]. Composition of subsystems is indicated by the self-arc of the Module entity hence we do not need to include a separate subsystem entity in our model.

- *Design level*: The design level contains the entities that are used to specify a detailed model of the implementation. One of the most frequent used design paradigms to specify these models is the object-oriented design. The entities used by this paradigm and contained in this abstraction level are Package and Class. Additionally, this level contains also the entities Directory and File. Latter two are not directly used in object-oriented design models but are used by programming languages such as Java and C++ to manage the source code in files and directories. Often there is a direct mapping between Class and File, and Package and Directory. For instance, in Java the package structure corresponds to the directory structure. Also in Java the implementation of a class typically is contained in one file. The three entities Directory, Package, and Class can have sub-directories, sub-packages, and sub-classes respectively which in Figure 6.1 is indicated by the self-arcs.

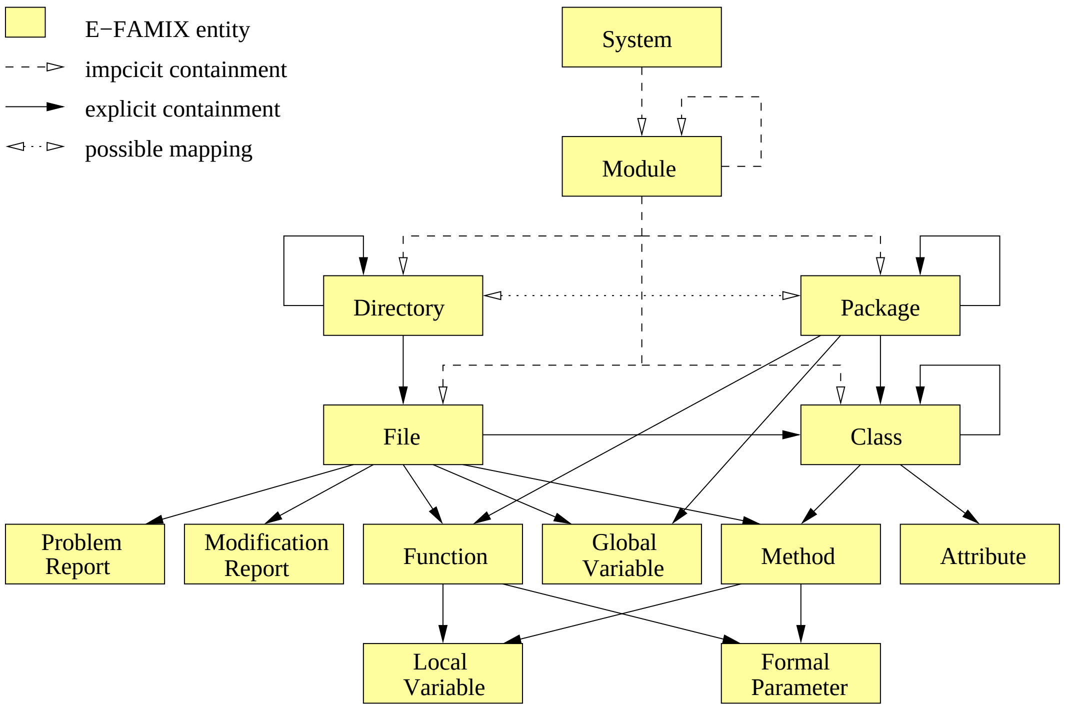


Figure 6.1: Containment hierarchy of source code models including modification and problem reports.

- *Code level*: They comprise all principal entities provided by a programming language to implement the system. Basically, these entities are Method, Attribute, Local Variable and Formal Parameter. A class contains methods and attributes. A method contains its parameters and local variables. Additionally, for handling the C part of C++ applications we added global functions and variables that are contained by a source file. Another add-on to the hierarchy model is the Modification Report and Problem Report entities. They indicate the facility provided by the E-FAMIX model to also abstract modification and problem report data along the File entity.

6.2.1 Establishing the links between the hierarchical levels

Whereas the links between the entities of design and the code level are explicit the links between the entities of the architectural and the design level are not. The basic reason for this is that software architectures are abstract concepts and so are software modules. Although recent software development processes provide support for such a mapping of abstract concepts to the implementation in practice it is not always carried out by the developers.

Typically, the system is decomposed into software modules. The implementation of each software module is broken down to one or more classes, packages or files or directories respectively. Information about such refinements often is contained in design documents. For instance, the design documents of Mozilla provide this information on the module owners website[1]. There the architects of Mozilla listed all software modules and for each module specified its owner, links to design documents, and the source code directories containing the implementation. The latter represent the links between the architectural and the design level entities that can be directly integrated into extracted source code models and then are available for information abstraction.

If no information about the mapping of abstract concepts to source code is available the concepts and their mapping has to be determined by the user. With respect to software modules such a determination is concerned with grouping the design elements (*i.e.,* classes, files, packages, directories) together that implement a coherent set of functionality. Straight forward techniques for such a grouping are based, for example on the:

- Package or the directory structure: Each package or directory is assigned to a software module
- Naming conventions: Prefixes or postfixes of class, package, file, or directory names that indicate the affiliation of an entity to a particular software module.

Hints about naming conventions can be obtained from design and code documents or developers. Both techniques also have been used in recent architecture recovery approaches, such as the Software Reflexion Models described by Murphy *et al.* [MNS01].

Another technique that provides support for relating entities of the architectural level with entities of the design and implementation levels is clustering. Clustering tools, such as Bunch [MMCG99], provide algorithms to (semi-)automatically aggregate tightly coupled source code entities (*e.g.,* classes, files) to modules and subsystems. However, to obtain reasonable results with clustering knowledge about the design is necessary, for instance, to configure the algorithm with appropriate main seeds.

Having determined the source code organizational units that implement software modules the remaining hierarchy and thus the links between the different abstraction levels is due to the containment relationships depicted by Figure 6.1. A module consists of one or more source files that contain the implementation of classes, functions, and the definitions of global variables. Classes contain attributes and methods which further contain parameter and local variable definitions that represent the lowest-level entities in our hierarchy model.

6.3 Software Metrics

Software metrics are a key input to our analysis and visualization approach. Metrics used by ArchView stem from source code and release history data and are computed during fact extraction and information abstraction. They range from metrics that assess the *size* (*e.g.,* number of methods in a class), *program complexity* (*e.g.,* cyclomatic complexity) of source code entities and *evolutionary* metrics (*e.g.,* number of modifications of a source file).

[1]http://www.mozilla.org/owners.html

ArchView concentrates on module and coupling dependency metrics on the design and architectural level. Modules refer to classes, packages, files, directories and software modules. Coupling dependencies refer to source code and change coupling relationships. Source code relationships are file includes, class inherits and aggregations, type definitions, function calls, and variable accesses. Metrics assessing lower-level source code entities, such as methods or attributes are also considered but not subject of this thesis.

6.3.1 Module Metrics

Table 6.1 lists the set of metrics used by ArchView to assess the size of software modules.

Metric	Description
NOD	Number of associated directories
NOF	Number of associated files
NOP	Number of associated packages
NOC	Number of associated classes
NGF	Number of global functions
NOM	Number of methods
NFM	Number of global functions and methods (NGF + NOM)
NGV	Number of global variables
NOA	Number of instance and class attributes
NOV	Number of global variables and attributes (NGV + NOA)
LOC	Length in number of lines

Table 6.1: Size metrics of software modules.

For assessing the complexity of a module's implementation ArchView uses the McCabe cyclomatic complexity [McC76] and Halstead complexity metrics [Hal77] listed in Table 6.2.

Metric	Description
CCMPLX	Accumulated McCabe cyclomatic complexity
HALCONT	Halstead Intelligent Content – language-independent measure of the amount of content (complexity) of a module
HALEFF	Halstead Mental Effort – number of elemental mental discriminations necessary to create, or understand a module
HALDIFF	Halstead Program Difficulty – measure of how compactly a module implements its algorithms

Table 6.2: McCabe and Halstead complexity metrics of software modules.

Table 6.3 lists the evolution metrics that ArchView computes for software modules.

Regarding problem reports several metrics are computed that address the different problem report categories as offered by the Bugzilla bug reporting system[2]. The four main categories are

[2]https://bugzilla.mozilla.org

Metric	Description
NMR	Accumulated number of modification reports assigned to a module during a specified observation period
NPR	Accumulated number of problem reports assigned to a module during a specified observation period
NPR-x	Accumulated number of problem reports of category x assigned to a module during a specified observation period
ENT	Accumulated entropy of modification reports (sum of lines added + lines deleted) of a module during a specified observation period

Table 6.3: Modification and problem report metrics of software modules.

Status, *Priority*, *Resolution* and *Severity*. Each category has several sub-categories that specify the different values of a main category. For instance, the status of a problem report in Mozilla can be unconfirmed, new, assigned, reopened, resolved, or verified. Basically, the values of the four main categories are unique hence we use them to build the names for the different problem report metrics. For instance, `NPR-verified` denotes the number of problem reports with status verified.

Table 6.4 lists the set of metrics to assess the coupling between modules. Basically, they describe the number of relationships of a certain type that a module has with other modules. Related to the E-FAMIX meta model these are file includes, class inherits and aggregates, method invokes, variable accesses, and type references. The list depicts the coupling metrics according to the coupling via class inheritance and method invocation. We use different prefixes to distinguish these metrics, such as “IH” for class inheritance and “I” for method invocations metrics.

In addition to the source code coupling metrics, ArchView takes into account change coupling metrics. Basically, they denote the number of pairwise modifications that occurred for a module during a given observation period. Table 6.5 lists these metrics.

6.3.2 Relationship Metrics

In addition to metrics computed for software modules, ArchView computes metrics for abstracted relationships. With regard to the E-FAMIX meta model these relationships are: `includes`, `inherits`, `aggregates`, `invokes`, `accesses`, `hasType`, and `couples`. Basically, measured metric values denote the number of aggregated lower-level relationships and involved source and target entities. Table 6.6 lists the set of metrics used in this thesis.

For more object-oriented source code metrics we refer the reader to the publications of Lorenz and Kidd [LK94], Henderson-Sellers [HS95], and Fenton and Pfleeger [FP96]. The metrics presented there can also be used by ArchView.

Metric	Description
IHFan-in	Fan-in of class inheritance – number of classes of other modules that inherit behavior from the module
IHNCE-in	Number of contained classes that are inherited by classes of other modules
IHNR-in	Number of in-coming inheritance relationships
IHNAR-in	Number of abstracted in-coming inheritance relationships
IHFan-out	Fan-out of class inheritance – number of classes of other modules from which a module inherits behavior
IHNCE-out	Number of contained classes that inherit behavior from other modules
IHNR-out	Number of out-going inheritance relationships
IHNAR-out	Number of abstracted out-going inheritance relationships
IFan-in	Fan-in of function/method calls – number of functions/methods of other modules that call the module's functions/methods
INCE-in	Number of contained functions/methods that are called by functions/methods of other modules
INR-in	Number of in-coming call relationships
INAR-in	Number of abstracted in-coming call relationships
IFan-out	Fan-out of function/method calls – number of functions/methods of other modules that are called by the module's functions/methods
INCE-out	Number of contained functions/methods that call functions/methods of other modules
INR-out	Number of out-going call relationships
INAR-out	Number of abstracted out-going call relationships

Table 6.4: Source code coupling metrics (in-coming and out-going class inheritance and function calls) of software modules.

6.4 Abstraction Algorithm

The goal of the ArchView abstraction algorithm is to reflect lower-level source code and release history information on higher levels of abstraction, such as onto the level of software modules. The input to our abstraction algorithm is an integrated source code and release history data model. Consider the following simplified integrated E-FAMIX conform data model depicted by Figure 6.2.

The Abstraction-Example has been decomposed into two software modules PA and PB whereby each module is implemented by one package with corresponding name. Package PA contains the class $CA1$ with the methods $a()$, $b()$, and $c()$ and further class $CA2$ with the methods $m()$ and $n()$. Package PB contains the class $CB1$ with the methods $x()$ and $y()$. Higher-level entities are packages (*i.e.,* modules) and classes. The containment relationship between entities are depicted as dashed (gray) arcs. Call relationships between methods are depicted as solid (red) arcs.

Source code model data of this form is obtained from the ArchView repository and input to the abstraction algorithm. The algorithm consists of following four basic steps:

Metric	Description
CFan-inout	Fan-in and fan-out of change coupling – number of files of other modules that a module is change coupled with
CNCE	Number of contained files that are change coupled with files of other modules
CNAC	Number of abstract change coupling relationships
CNMR	Number of modification reports involved in the change coupling
CENT	Entropy of modification reports involved in the change coupling

Table 6.5: Change coupling metrics of software modules.

Metric	Description
RNMR	Number of modification reports involved in the change coupling
RNMRPR	Number of modification reports involved in the change coupling for which a link to a problem report exist
RENT	Entropy of modification reports involved in the change coupling
RNAC	Number of abstracted change coupling relationships
RNCR	Number of calling functions/methods
RNCE	Number of called functions/methods
RNAI	Number of abstracted function/method call relationships
RNSUB	Number of contained sub classes
RNSUP	Number of inherited super classes of other modules
RNAIH	Number of abstracted class inheritance relationships

Table 6.6: Metrics of abstracted change coupling, invokes, and inherits relationships.

1. *Select entities and relationships to be processed*: The user selects a set of software modules to be analyzed. In our example we parameterize the algorithm with the software modules (packages) PA and PB, and the $invokes$ relationship type. Additionally, we specify the source and target entity type of the relationships ($Method$).

2. *Compute contained source entities for each selected entity pair*: Depending on the relationship type to be analyzed the algorithm computes sets of lower-level entities contained by each element of the pair. For example, analyzing the $invokes$ relationship between the modules PA and PB the algorithm computes the two sets $setA$ and $setB$. $setA$ holds the methods $a()$, $b()$, $c()$, $m()$, and $n()$ that are contained by package PA. $setB$ comprises the methods $x()$, and $y()$ that are contained by package PB.

3. *Query relationships between sets of entities*: Based on the source code model graph the algorithm computes the direct relationships from entities of $setA$ to entities of $setB$. Referring to our example the result of the query comprises the direct invokes relationships $b() \rightarrow x()$, $m() \rightarrow x()$, $n() \rightarrow x()$, and $n() \rightarrow y()$. Whenever the query retrieves at least one lower-level relationship the algorithm aggregates these relationships to an abstract relationship and computes the coupling metrics presented by Table 6.6.

4. *Output abstracted relationships and metrics*: The result comprises aggregated (*i.e.*, ab-

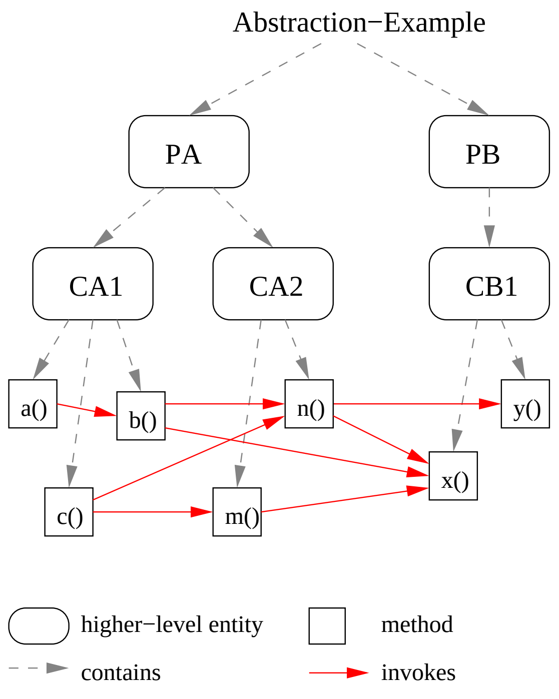


Figure 6.2: Source code model with 2 packages, 3 classes, 7 methods, contains and invokes relationships.

stracted) relationships between selected higher-level entities and computed metric values assigned to them. They are stored in the source code model.

The enriched source code model of our example is depicted by Figure 6.3. It contains a new invokes arc between package PA and PB with three measured values that indicate the number of aggregated lower-level invokes relationships (RNAI), the number of callers in PA (RNCR) and the number of callee's in PB (RNCE).

Listing 6.1 outlines the implementation of the ArchView abstraction algorithm in Java using the structured query language (SQL) for information retrieval.

Listing 6.1: Algorithm to abstract direct relationships of type $rType$ between the higher-level entities eA and eB.

```
public class Abstractor extends Query {
    private Entity eA, eB;
    private String fromType, toType;
    private String relType;

    protected arcFactory = new ArcFactory();

    public Abstractor(Entity eA, Entity eB, String fromType,
                String toType, String relType) {
```

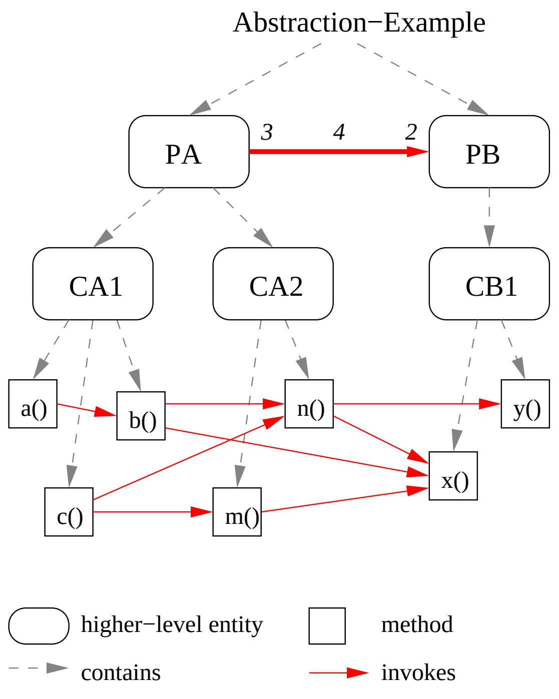


Figure 6.3: Source code model graph enriched by abstracted *invokes* relationship between PA and PB and computed coupling metrics.

```
        this(eA, eB, fromType, toType, relType);
    }

    public Arc abstractRel() {
        Arc arc = null;

        // get entities of eA and entities of eB
        Vector setA = eA.getContainedEntities(fromType);
        Vector setB = eB.getContainedEntities(toType)

        // select relationships between entities of setA and setB
        String sql = "select count(*) nr, "
            + "count(distinct r.from) nrFrom, "
            + "count(distinct r.to) nrTo "
            + "FROM " + relType + "r "
            + "WHERE r.nodeFrom IN (" + buildIDs(setA) + ") "
            + "AND r.nodeTo IN (" + buildIDs(setB) + ")";

        Statement sqlStm = connection.createStatement();
        ResultSet rs = sqlStm.executeQuery(sql);
        if (rs.next()) {
            // create new abstract arc and add measures
            arc = arcFactory.create(relType, eA.getID(), eB.getID());
            arc.addAttribute("nr", rs.getInt("nr"));
            arc.addAttribute("nrFrom", rs.getInt("nrFrom"));
            arc.addAttribute("nrTo", rs.getInt("nrTo"));
        }

        return arc;
    }
}
```

The algorithm abstracts direct relationships of type $rType$ between two entities eA and eB. Step 1 accords to the initialization of the algorithm that is done in the constructor. The algorithm itself is implemented by the $abstractRel()$ method. According to step 2 it first computes the contained entities of eA and eB stored in the vectors $setA$ and $setB$. The identifiers of entities of both sets are used in the SQL-query that is composed next (step 3). It is executed on the database table $relType$ and retrieves relationships with a source (from) entity that is contained by $setA$ and with a target (to) entity that is contained by $setB$. The number of matched relationships (nr) and number of source ($nrFrom$) and target ($nrTo$) entities are returned. If relationships have been found then a new arc object of type $relType$ between eA and eB is created. Retrieved measures are assigned to the arc object that is returned to the calling method.

6.5 Summary

The ArchView abstraction algorithm aggregates relationships between lower-level source code entities along the containment hierarchy up to higher-level entities, such as classes, files, directories, packages, and respectively software modules and subsystems. A relationship between two higher-level entities is established whenever there is at least one relationship between their contained source code entities.

In addition to abstracted relationships, the algorithm derives measures for each higher-level entity and abstracted relationship. They are indicators for the size of entities and the strength of abstracted relationships. Established relationships, as well as computed measures are added to each source code model.

The abstraction step is applied to data models of selected releases. Enriched models are used in the analysis and visualization phase to analyze and present higher-level views on the implementation and its evolution.

CHAPTER 7

VISUALIZATION & ANALYSIS

This chapter describes the ideas and techniques used by ArchView to compute different views on the integrated data model. Each view highlights a particular aspect and aids an engineer in understanding the current implementation and its evolution.

7.1 INTRODUCTION

Visualization has been accepted as a useful means to understand complex data, because visual displays allow the human brain to study multiple aspects of complex problems – like reverse engineering – in parallel [SDBP98]. However, often the visualizations themselves are hard to interpret, and in the case of evolutionary data, they often succeed in obscuring the relevant information.

Visualization has to focus on the interesting information and hide/filter information of minor interest. With regard to architecture recovery the degree of interestingness depends on: 1) the architectural aspect/property the user wants to analyze and 2) the viewpoint from which the user looks at the extracted, integrated, and abstracted data models.

The aspect focused on by ArchView concerns the implementation and the evolution of software systems. They comprise aspects of the:

- Logical structure of the implementation:
  - Which are the main building blocks (*i.e.,* implementation units) of the software system?
  - Which units are coupled with each other and how strong are these coupling dependencies?
  - Are there entities and relationships that indicate *Bad Smells,* such as cyclic coupling dependencies or *God Modules*?

- Evolution:
  - How did the software modules and the coupling dependencies evolve – can we identify change prone modules and *Bad Smells*, such as *Divergent Change* or *Shotgun Surgery*?
  - Which units were most vulnerable to problems and modified most frequently?
  - Are there change couplings between modules and how strong are they?

To answer these questions ArchView provides a number of different views on the implementation and its evolution. In the following we present these views and describe the visualization techniques that we use to compute them.

7.2 Feature Vectors and Evolution Matrices

The data input to the visualization and analysis techniques of ArchView is obtained from the integrated data model stored in the ArchView repository. It contains the software module information, the link to the source code entities that implement a software module, the aggregated source code and change coupling dependencies between software modules and lower-level source code entities, and measured values of module and relationship metrics.

Metric values of each module and relationship are represented by an i-dimensional feature vector $M = \{m_1, m_2, ..., m_i\}$. To denote the evolution of a module or relationship the values of the same set of metrics are tracked over n releases. The results are per module or relationship a set of n feature vectors. The release number is added to the feature vector leading to $M^n = \{m_1^n, m_2^n, ..., m_i^n\}$. Based on these vectors the evolution of a module or relationship is expressed by the following evolution matrix E containing the n vectors with measured values of i metrics:

$$E_{i \times n} = \begin{pmatrix} m_1' & m_1'' & .. & m_1^n \\ m_2' & m_2'' & .. & m_2^n \\ . & . & ... & . \\ . & . & ... & . \\ m_i' & m_i'' & .. & m_i^n \end{pmatrix}$$

Evolution matrices are computed for each selected module and relationship. The number of metrics to be considered depend on the aspect to analyze or the view to visualize respectively. ArchView provides a predefined set of views and metric configurations but also supports the user to compose his/her own views.

Table 7.1 provides a tabular representation of an evolution matrix. It lists an excerpt of metric values used to characterize the size of a software module. In this example, the software module is Mozilla's Document Object Model (DOM) module and metric values are of seven selected Mozilla releases. The abbreviations of the metrics are listed in the first column. Remaining columns are headed by the release number and contain the measured values.

Metric	0.92	0.97	1.0	1.2	1.4	1.6	1.7
NOD	44	45	50	50	50	50	49
NOF	397	405	443	464	477	485	492
NOC	459	476	528	566	595	607	609
NOM	9.802	9.395	10.346	10.823	11.104	11.130	11.068
NGF	333	880	288	325	341	334	330
NFM	10.135	10.275	10.634	11.148	11.445	11.464	11.398
NOA	906	988	1.118	1.236	1.292	1.316	1.293
NGV	219	227	234	262	250	237	229
NOV	1.125	1.215	1.352	1.498	1.542	1.553	1.522

Table 7.1: Example of an evolution matrix $E_{9\times 7}$ containing measured values of 9 module size metrics of Mozilla's DOM module of 7 releases.

The size values listed in the table clearly indicate a growing module. For instance, the number of files (NFM) that implement the DOM module increased from 397 to 492 source files. A similar trend can be observed for the number of global functions and methods (NGF) that increased from 10.135 to 11.398 functions/methods.

In this form evolution matrices are input to the ArchView analysis and visualization techniques to, for instance, generate views on one or several releases.

7.3 Higher-Level Views on a Release

The basic principle of the ArchView visualization technique is the mapping of module and relationship metrics to graphical attributes. A recent approach that concentrated on such a mapping are the polymetric views introduced by Lanza *et al.* [LD03]. Basically, the model data is visualized with graphs whereby nodes represent the source code entities and edges the relationships between them. The extension to traditional graph visualization approaches is that nodes and edges of graphs are rendered with metric values as demonstrated by Figure 7.1.

With the polymetric view technique up to 5 different metrics in nodes and 3 different metrics in relationships can be visualized. The five node metrics are for the width, height, color, x-, and y-position of a node in a graph. The three edge metrics are for the width, length, and color of an edge.

The width, height, length, and color metrics of nodes and edges can be used in different graph layouts. The x- and y-position metrics are not applicable in graph layouts in which the position is computed by the layout algorithm (*e.g.,* tree, spring, etc.).

Fenton and Pfleeger call this rendering technique *measurement mapping* [FP96]. Their condition is: "A measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations". This condition is satisfied by the polymetric views technique in that larger values lead to larger glyphs.

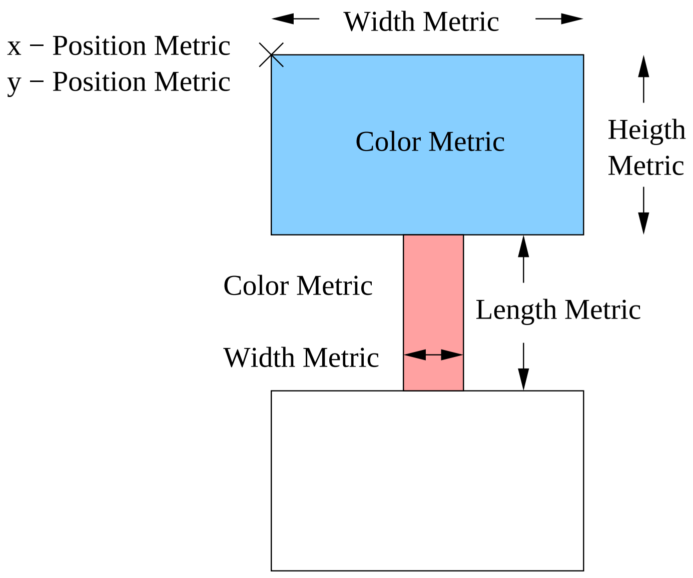


Figure 7.1: Mapping of metrics to graph attributes using polymetric views.

ArchView adapts the polymetric views technique to visualize the various views on computed data models. Metric values are retrieved from the evolution matrices that in case of one release are vectors. The mapping of the values of a vector to graphical attributes (*i.e.*, size, color and position) is done by the graph visualization tool.

We use Lanza's CodeCrawler [Lan03] to draw and layout the different views on the implementation of *one* software release. In addition to the existing polymetric views, we specify a set of new views that also take into account our evolutionary metrics. The example views are taken from the Mozilla case study that we present in Chapter 8.

For describing the configuration of each view we use the following schema:

Nodes:	Entity type represented by a node in the graph. We consider software modules (Mozilla), source files, packages, and classes.
Edges:	Relationship type represented by an edge in the graph.
Scope:	The scope of a view can be all or a selected set of modules of one or n releases.
Node Metrics:	List of graph node attributes and mapped module metrics. For traditional polymetric views graph node attributes are Size, Color, and Order. For views with Kiviat diagrams we list the categories of metrics.
Arc Metrics:	List of graph edge attributes and mapped relationship metrics. In this thesis we limit the edge attributes to the Width of edges.

For each view we present the interpretation of the visualized information on hand of an example and the possible variations in the view configuration.

1. Modification hotspots view

The objective of this view is to highlight the implementation units that were most vulnerable to problems and modified most frequently as well as the idle and stable entities.

Nodes:	Software modules, source files
Edges:	-
Scope:	Full system of 1 release
Node Metrics:	
Size:	Width: NMR
	Height: NPR
Color:	CCMPLX
Order:	NMR
Arc Metrics:	-

View Interpretation Large nodes represent fault prone entities that were frequently modified during a specified observation period. They are the entities that influenced the evolution of a system most. Small nodes represent entities that were not touched during this time hence denote the stable entities. The color of a node indicates the McCabe Cyclomatic Complexity of a module whereby dark nodes represent the complex modules.

Figure 7.2 depicts an example taken from the Mozilla case study. Evolution metric values are of modification and problem reports that were committed and reported during the time from release 1.6 to 1.7. The large nodes at the bottom represent the change prone source files with a large number of reported problems and committed changes. The small nodes on the top represent the stable entities. The color of nodes indicates that most of the change prone source files have also high values for the cyclomatic complexity and program difficulty metrics.

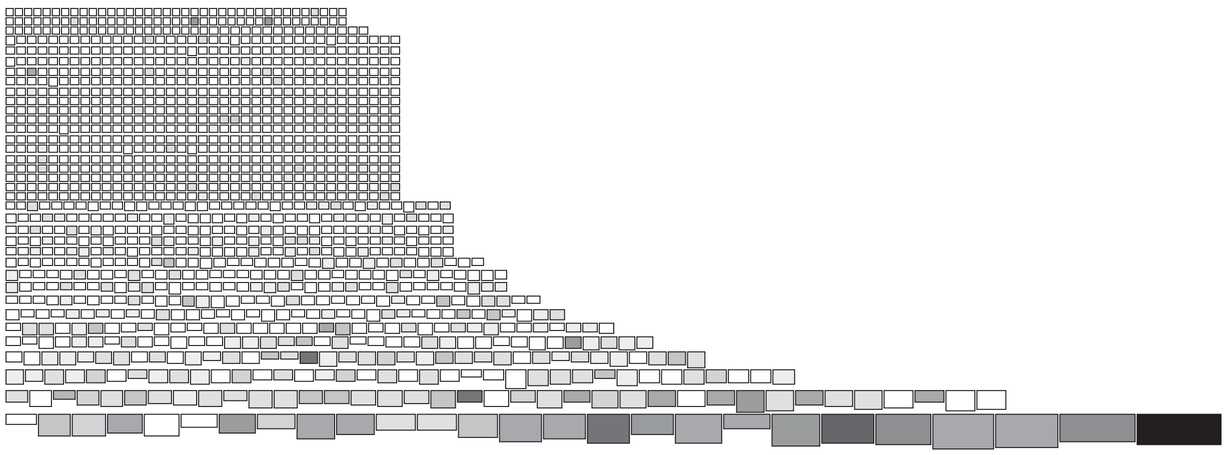


Figure 7.2: Modification hotspots view on Mozilla 1.7 source files with measured evolution and complexity metrics. Node: width=NMR; height=NPR; color=CCMPLX; order by NMR.

Variations For the evolution metrics the user can select different observation periods. Instead of the cyclomatic complexity the user can also select other size and complexity metrics, such as lines of code (LOC), number of functions and methods (NFM), or the Halstead program difficulty (HALDIFF) metric.

2. Source code coupling view

This view is used to represent the coupling between implementation units meaning software modules, source files, packages, or classes. The objective is to highlight the strong coupled entities and the strong coupling relationships. In the presented view configuration coupling relationships refer to function/method calls.

Nodes:	Software modules, source files, packages, classes
Edges:	invokes
Scope:	Selected elements of 1 release
Node Metrics:	
Size:	Width: IFan-out
	Height: IFan-in
Color:	NFM
Order:	-
Arc Metrics:	
Width:	RNAI

View Interpretation Wide, narrow nodes represent implementation units that use functionality offered by other units but themselves are not used by other units. They are typical service requesters. Tall, narrow nodes denote service providers whose provided functionality is used by other units. The color indicates the amount of functionality implemented by a module – the darker a node the more functionality the entity contains.

Figure 7.3 depicts an example of a source code coupling view. The node on the right side of the graph represents a typical provider of services that are used by the requester node on the left side. Edges represent the coupling relationships that are due to function/method calls. The direction of edges denotes the usage of functionality. To indicate the direction of a relationships the edge is drawn from the bottom of the requester node to the top of the provider node. The width indicates the strength of the coupling relationships in terms of number of function/method invocations.

Variations Views can get complex when visualizing a large number of entities and relationships. To produce graspable views the user is able to interpret ArchView provides filtering and user-selection. Thresholds are used to filter nodes and edges, such as standalone nodes and edges of weak coupling relationships.

Regarding the type of coupling the user can select different types of source code relationships and use the corresponding fan-in, fan-out, and size metrics. For instance, to depict the coupling by class inheritance we select the *inherits* relationship type, the IHFan-out and IHFan-in, and the number of classes (NOC) metrics.

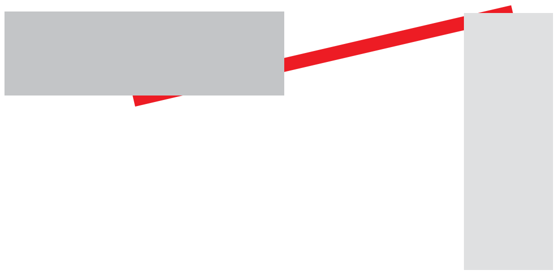


Figure 7.3: Source code coupling view by function/method calls of two software modules. Node: width=IFan-out; height=IFan-in; color=NFM; Arc: width=RNAI.

3. Change coupling view

This is similar to the source code coupling view but shows the change dependencies between implementation units instead of the source code couplings. Visualized metrics refer to an user-specific observation period. The objective is to highlight the entities that most frequently were modified together and consequently have a strong change coupling.

Nodes:	Software modules, source files
Edges:	couples
Scope:	Selected elements of 1 release
Node Metrics:	
Size:	Width: NMR
	Height: NMR
Color:	-
Order:	-
Arc Metrics:	
Width:	RNAC

View Interpretation Large nodes represent frequently modified entities, such as the node on the right side of the graph of Figure 7.4. Small nodes represent stable entities. Each edge denotes a change coupling between two entities that resulted from a modification that affected both entities. The more such pairwise modifications occurred the stronger is the change coupling as indicated by the width of an edge.

Variations The selection of the entities of interest as well as filtering reduces the amount of information and keeps views understandable. Concerning the evolution metrics the user can select a different observation period to analyze the change coupling that occurred during this period.

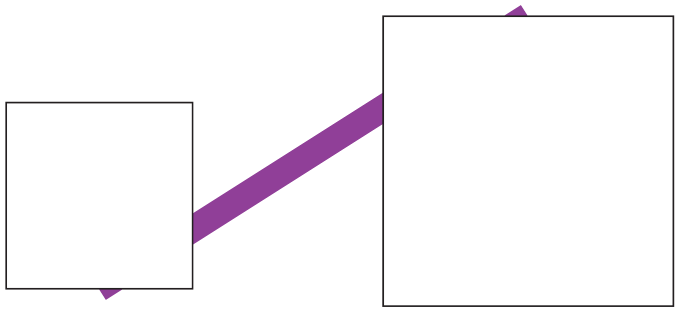


Figure 7.4: Change coupling view of two software modules. Node: width=NMR; height=NMR; Arc: width=RNAC.

7.4 Visualizing Multiple Evolution Metrics

The ArchView visualization approach follows the polymetric views principles of mapping metric values to graphical attributes. However, there is a problem with the number of metrics that can be visualized for a node using current tools, such as CodeCrawler. The primary reason lies in the limited number of attributes for the glyphs used to draw the graph nodes. In case of CodeCrawler the glyphs support up to 5 metrics whereby the metrics for the x- and y-position are not applicable in all graph layout algorithms.

Kiviat (also called radar) diagrams are suited to present multivariate data. Figure 7.5 shows an example of a Kiviat diagram representing measured values of six metrics of $moduleA$. The six metrics $M1, M2, ..., M6$ are circularly arranged. For each metric there is a straight line drawn from the center point of the diagram to its outer boundary. The value of each metric $m_1^{'}, m_2^{'} ..., m_6^{'}$ is plotted on its corresponding straight line and adjacent values are connected by lines. The result is a polygon with 6 vertices.

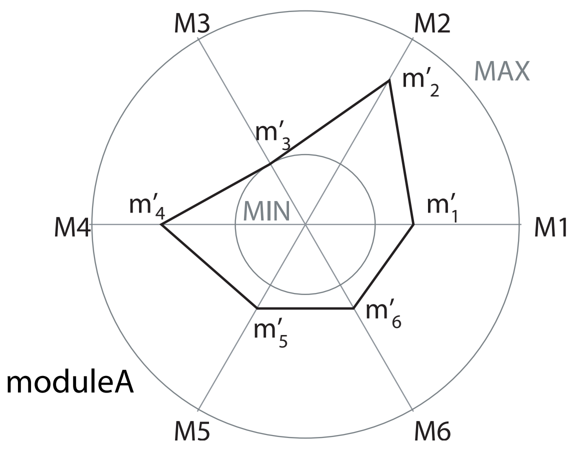


Figure 7.5: Kiviat diagram with values of 6 metrics $M1, M2, ..., M6$ of `moduleA`.

To prevent diagrams to become cluttered with information certain requirements have to be met: 1) normalization of metric values to a maximal drawing length to prevent over-sized Kiviat diagrams; 2) using a minimum (*i.e.*, an offset) that is added to computed values to prevent information cluttering in the center of Kiviat diagrams. Metric values are drawn with respect to these minimum and maximum drawing range. We will see later on that the limitation in size is necessary to link Kiviat diagrams to Kiviat graphs.

In the following we present two new polymetric views with Kiviat diagrams that aid in analyzing the size, complexity, and evolution of implementation units.

1. Detailed system hotspots view

This view is an extension of the system hotspots view described by Lanza [Lan03]. The primary objective is to identify the large and small as well as complex and trivial implementation units.

Nodes:	Software modules, source files, packages, classes
Edges:	-
Scope:	Full system of 1 release
Node Metrics:	
Complexity:	CCMPLX, HALCONT, HALEFF, HALDIFF
Size:	LOC, NOD, NOF, NOP, NOC, NFM, NOV
Arc Metrics:	-

View Interpretation Kiviat diagrams with big circles indicate large and complex elements. If the circles are exceptionally large, then the element is a *God Module*. In contrast, small circles indicate trivial elements of small size and low complexity.

Circles can exhibit indentations and peaks. For instance, a peak in the number of variables (NOV) metric indicates data storage elements. Diagrams with a low NFM value but high complexity denotes modules that implement complex algorithms.

Figure 7.6 shows an example of a Kiviat diagram of one software module. The diagram indicates a large, complex module that contains a large number of variables and attributes. The relation between the number of functions (NFM) and the LOC metrics indicates a module with few but long and complex functions and methods. The indentation by the number of packages (NOP) metric is due to the fact that the developers did not use C/C++ namespaces.

Variations Depending on the abstraction level the user can add or omit different size metrics. For instance, on the file and class level the NOD, NOF, and NOP metrics are not used. Instead of this metrics the user may add metrics quantifying the number of attributes and methods according to the different access modifiers.

2. Detailed modification hotspots view

This view presents the evolution metrics of implementation units that currently are software modules and source files. The objective is to highlight the elements with a large or small number

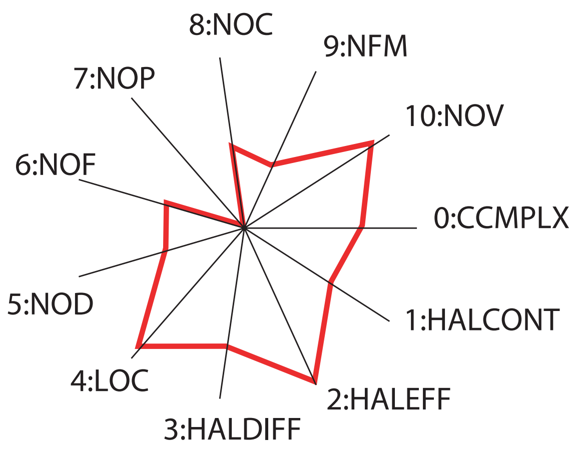


Figure 7.6: Detailed system hotspots view of one software module with complexity (nr. 0..3) and size metrics (nr. 4..10).

of modifications and reported problems. The first category denotes change prone modules to which most of the maintenance and evolution activities were dedicated to. The latter category denotes idle elements.

Nodes:	Software modules, source files
Edges:	-
Scope:	Full system of 1 release
Node Metrics:	
Modifications:	NMR
Problems:	NPR, NPR-s-, NPR-blocker, NPR-critical, NPR-enh, NPR-major, NPR-minor, NPR-normal, NPR-trivial, NPR-p-, NPR-p, NPR-P1, NPR-P2, NPR-P3, NPR-P4, NPR-P5
Arc Metrics:	-

View Interpretation Change prone modules are indicated by peaks for the number of modifications (NMR) and reported problems (NPR). The latter is further detailed by the different problem report metrics. The most change prone modules have peaks in measured NPR-critical and NPR-P1 and NPR-P2 metrics. The first metric denotes the number of critical problems and the latter the number of problem reports of high priority. Figure 7.7 depicts an example of a change prone software module.

Another characteristic diagram pattern denotes elements that underwent cosmetic modifications. They are indicated by relatively high values for the NPR-trivial, NPR-P4 and NPR-P5 metrics that refer to the number of trivial problems with low priority.

Diagrams with small circles denote elements with few modifications and problems. We call these elements *idle* elements.

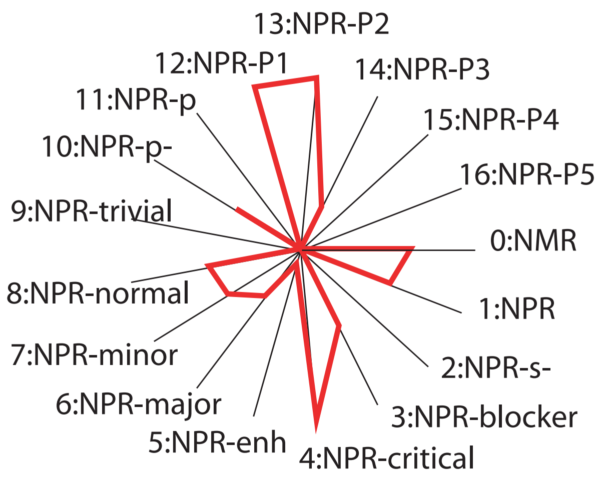


Figure 7.7: Detailed modification hotspots view of one software module with modification and problem report metrics.

Variations The user can reduce the number of metrics to one category, such as severity or priority metrics. Because the metrics are time dependent the user may also select different observation periods.

7.4.1 Visualizing Data of n Releases

In addition to visualizing and analyzing data of one release another objective of ArchView is to communicate the evolution of metrics across n releases. So far we used Kiviat diagrams to visualize multiple metrics. In this section we demonstrate how we extended the Kiviat diagrams to also visualize multiple metrics of n releases.

The two principles that allow ArchView to visualize data of several releases are: 1) normalizing metric values to the range determined by the minimum and maximum of each metric; and 2) using a metric to represent the time-order of releases.

Reconsidering the evolution matrix ArchView computes the maximum of each metric across the n releases.

$$MAX(M_i) = max(m_i^{'}, m_i^{''}, ..., m_i^{n})$$

The minimum of each metric can be considered 0. The effective drawing length of each metric value is computed by normalizing the value by its maximum and adding an offset to it.

$$length(m_i^n) = offset + \frac{m_i^n * c}{MAX(M_i)}$$

The constant c specifies the maximum drawing size and together with the $offset$ constant is used to control the size of Kiviat diagrams. These constants can be configured by the user. The different values computed for a metric across n releases are plotted in the diagram and adjacent metrics of the same release are connected. The result is a diagram that per release shows a polygon that represents the normalized metric values of one feature vector.

The evolution of metrics is highlighted by filling the polygons emerged between two subsequent releases and adjacent metrics with different colors. Using appropriate color gradients, such as the rainbow colors, the order of releases is made transparent and strong changes in metric values are highlighted. But, there is a limit to the number of data and releases because diagrams get blurred with polygons and colors. Based on our experiences with the Mozilla project we set this limit to 20 metrics of 10 releases.

Strong changes in metric values are further emphasized by grouping metrics according to certain properties, such as size, complexity, or method call fan-in and fan-out of modules. Resulting sectors contain metrics that quantify certain aspects of the implementation and evolution respectively and their trends. For example, by grouping metrics that quantify in-coming and outgoing `uses` relationships in two separate sectors of the diagram users can categorize modules into service providers and service requesters or both.

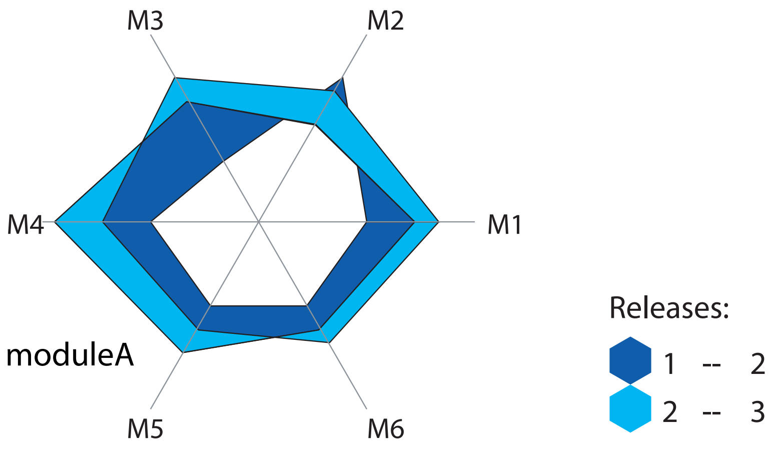


Figure 7.8: Kiviat diagram with 6 metrics $M1, M2, ..., M6$ of 3 releases of `moduleA`.

Figure 7.8 depicts an example of visualizing six metrics of $moduleA$ of three releases 1, 2, and 3. In this example $M1$ presents the number of modifications (NMR) between two subsequent releases which also specifies the chronological order of releases. Consequently, metric $M2$ is decreasing whereas the values of remaining metrics increase from release 1 to release 2. From release 2 to 3 the values of metric $M2$, $M3$, and $M6$ increase whereby $M4$ and $M5$ decrease.

1. Detailed system hotspots evolution view

This view is an extension to the detailed system hotspots view presented before. The extension is by visualizing measured values of the same set of complexity and size metrics of up to n releases. The primary objective of this view is to represent the growth in size and complexity of implementation units, such as software modules, files, packages and classes.

Nodes:	Software modules, source files, packages, classes
Edges:	-
Scope:	Full system of n releases
Node Metrics:	
Complexity:	CCMPLX, HALCONT, HALEFF, HALDIFF
Size:	LOC, NOD, NOF, NOP, NOC, NFM, NOV
Arc Metrics:	-

View Interpretation The diagrams facilitate the detection of several evolution patterns that have been described by Lanza in his thesis [Lan03]. The *Pulsar* is indicated by overlapping polygons. A diagram with small circles in the earlier releases that suddenly grew to a large circle in one of the recent releases denotes a *Supernova*. In contrast, large circles in the earlier releases that suddenly shrank to small circles indicate a *White Dwarf*. Diagrams with large circles indicate *God Modules* that were and still are large in size and complex. Lanza refers to this pattern as *Red Giant*. An *Idle* module is indicated by narrow polygons as shown by Figure 7.9.

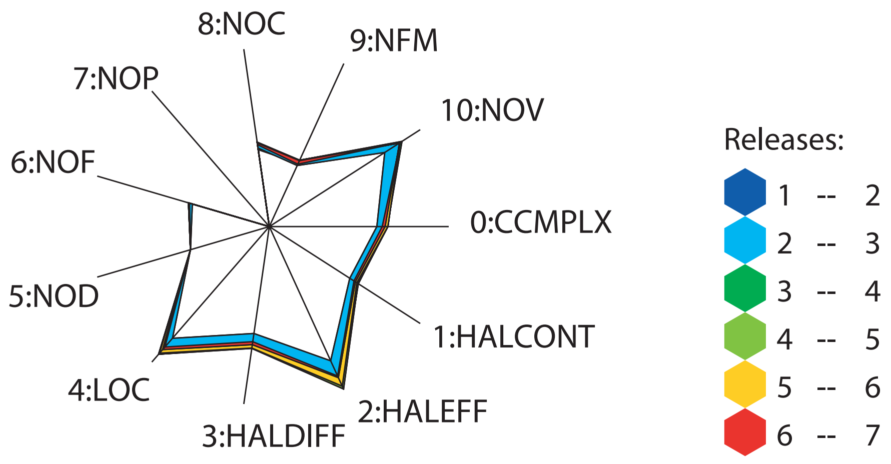


Figure 7.9: Detailed system hotspots evolution view with size and complexity metrics of 7 releases indicating an *Idle* module.

Variations For this view the set of variations of the detailed system hotspots view are applicable. In addition, the number of releases can be configured to focus on a specific observation period.

2. Detailed modification hotspots evolution view

The focus of this view is on visualizing the trends of evolution metrics. The objective is to highlight elements that frequently were involved in modifications and problems as well as to highlight the stable elements.

Nodes:	Software modules, source files
Edges:	-
Scope:	Full system of n releases
Node Metrics:	
Modifications:	NMR
Problems:	NPR, NPR-s-, NPR-blocker, NPR-critical, NPR-enh, NPR-major, NPR-minor, NPR-normal, NPR-trivial, NPR-p-, NPR-p, NPR-P1, NPR-P2, NPR-P3, NPR-P4, NPR-P5
Arc Metrics:	-

View Interpretation Kiviat diagrams with large circles and polygons denote elements that were most vulnerable to problems. They are the change prone modules with respect to the evolution of the system. Typically, they have a high number of modification (NMR) and problem reports (NPR). Indentations are possible for problem report metrics of low priority, such as `NPR-trivial` or `NPR-P5`.

Diagrams that show large polygons in the earlier releases and narrow polygons in the recent releases indicate elements whose evolution became stable. Figure 7.10 depicts such an implementation unit. The inner polygons show an increase of the number of reported critical problems of high priority in the first three releases. Then, the polygons became narrow that indicates a reduction of reported problems.

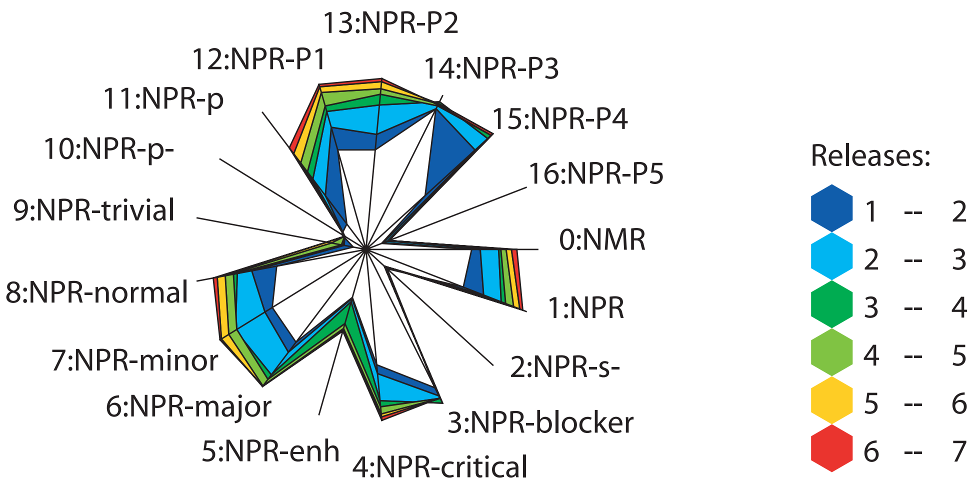


Figure 7.10: Detailed modification hotspots evolution view with evolution metrics of 7 releases.

Variations The user can select a different set of problem report metrics to concentrate on the evolution of a particular problem report category. Further, the observation period can be configured to show the changes in measured values between a specific set of releases.

7.4.2 Kiviat Graphs

ArchView uses a Kiviat diagram per module to represent values of multiple metrics and their changes across several releases. But, although the diagrams provide quantitative measurements they do not explicitly show the coupling dependencies between the implementation units.

To also represent these dependencies ArchView links diagrams to Kiviat graphs. Nodes in the graph represent the implementation units and edges represent the coupling dependencies between them.

1. Source code coupling evolution view

The view is an extension to the source code coupling view. Its objective is to show the coupling by function calls between implementation units. Coupling metrics presented by Kiviat diagrams are of up to n releases and quantify the call fan-in and fan-out properties of modules, files or classes and changes in these metrics. Edges represent the aggregated function call relationships.

Nodes:	Software modules, source files, packages, classes
Edges:	invokes
Scope:	Selected elements of n releases
Node Metrics:	
Node:	NFM, Fan-in (INAR-in, IFan-in, INCE-in, INR-in) and Fan-out (INAR-out, INCE-out, IFan-out, INR-out) metrics
Arc Metrics:	
Width:	RNAI

View Interpretation The view is extension to the Source Code Coupling View. It provides further measured metric values that facilitate the characterization of an implementation unit into service providers, service requesters or both. The node on the right side of the graph shown in Figure 7.11 represents a typical requester – high fan-out values and fan-in values that are almost zero. The node on the left side refers to an element that is both, a requester and a provider of functionality. Its diagram shows high fan-in and fan-out values.

Other interesting aspects visualized by this view concern the change in the provider and requester behavior. For instance, diagrams that show decreasing values for the fan-in and fan-out metrics but a stable value for NFM metric indicates elements that die. Either the functionality has been moved to other elements or it became obsolete. When the values reach zero then the element represents dead code and should be removed.

Steadily increasing fan-in and fan-out metric values indicate increasing coupling with other elements. If the corresponding coupling relationship directs in one direction then it is not a problem per se. However, a cyclic coupling dependency denotes a design flaw.

Variations Regarding the node metrics the same variations as mentioned for the source code coupling view are applicable. The user also can select different observation periods to analyze the coupling between entities at different points in time.

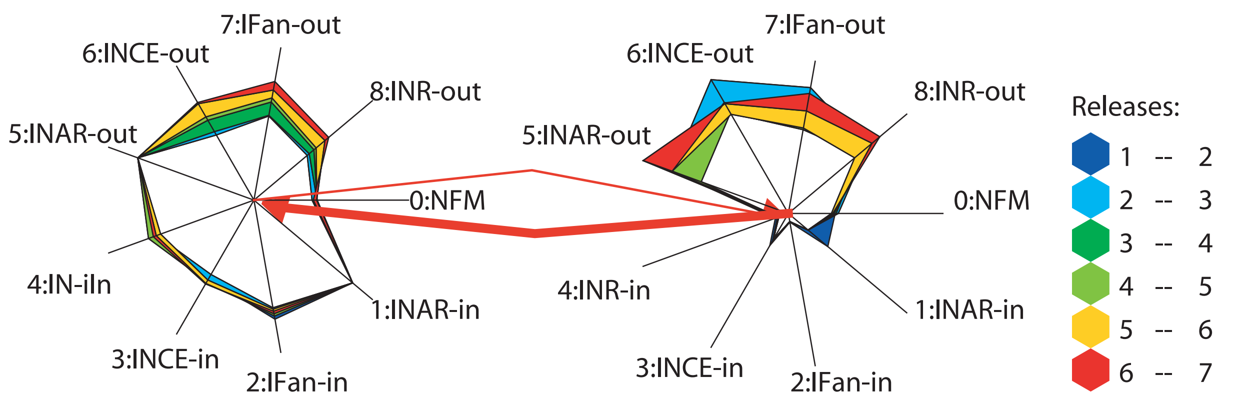


Figure 7.11: Source code coupling evolution view on the coupling by function calls with measured call fan-in and fan-out metric values of 7 releases.

7.5 Summary

In this chapter we presented the concepts and techniques to create views that show implementation and evolution specific aspects of a software system. The focus is on highlighting the change prone modules and heavy coupling relationships.

For the creation of these views we built upon the polymetric views technique and extended it towards the visualization of multiple metrics of n releases. Using polymetric views we introduced a set of hotspots views that show the size, complexity, and coupling metrics of a system's modules as well as their modification and problem report data. Size, complexity, and coupling metrics are used to assess maintenance and evolution efforts. Hotspots refer to modules that in relation to other modules need more effort. How much effort more is needed is indicated by the number of problems and modifications.

In extension to the traditional polymetric views we presented the Kiviat diagrams that we use to: 1) visualize values of multiple metrics tracked over n releases; and 2) visualize source code and change couplings with Kiviat graphs. Based on this technique we introduced several views, such as the Detailed System Hotspots and Detailed Modification Hotspots Evolution View. They facilitate a more detailed diagnosis by highlighting the progression of metric values. Users can spot strong increases and decreases in metric values that either indicate improvements or degradations in the design and architecture. For instance, design degradation is indicated by strong increases in the coupling metrics that are accompanied by increasing numbers of problems and modifications. In contrast, an improvement is indicated by low coupling values accompanied by decreasing numbers of problems and modifications.

In the next chapter we demonstrate the ArchView techniques by applying them to the Mozilla open source project.

CHAPTER 8

MOZILLA CASE STUDY

To demonstrate and evaluate our approach we applied ArchView to the Mozilla open source software project. The source code as well as the release history data (CVS and Bugzilla) are freely available on the Mozilla developers web-site. The primary objective of the case study is to highlight aspects that concern the implementation and the evolution of Mozilla.

8.1 MOZILLA PROJECT

The Mozilla data that we considered in the case study stems from 7 Mozilla releases starting with release 0.92 (28th of June, 2001) up to release 1.7 (17th of June, 2004). The time interval between each two subsequent releases is about half a year. Table 8.1 lists the set of releases together with their release dates, the number of source code files (.h, .cpp, and .c) and the total lines of C/C++ source code (LOC) per release.

Release	Date	#.h	#.cpp	#.c	NOF	LOC
Mozilla 0.92	28th of June, 2001	4.695	3.847	1.600	10.142	3.306.122
Mozilla 0.97	21st of December, 2001	4.824	3.896	1.635	10.355	3.518.124
Mozilla 1.0	5th of June, 2002	5.258	3.961	1.970	11.189	3.868.025
Mozilla 1.3a	13th of December, 2002	5.464	4.119	1.806	11.389	3.924.064
Mozilla 1.4	30th of June, 2003	5.585	4.168	1.832	11.585	3.986.466
Mozilla 1.6	15th of January, 2004	5.473	4.161	1.546	11.180	3.835.173
Mozilla 1.7	17th of June, 2004	5.662	4.278	1.562	11.502	3.912.631

Table 8.1: Selected Mozilla releases with the number of files (NOF) and lines of code (LOC) metrics. The number of header files (#.h) includes header files generated from .idl files.

The table shows that the amount of source code is increasing from release to release. For instance, the number of source files increased by 360 files or 606.509 LOCs from Mozilla release 0.92 to 1.7.

An interesting peak in terms of number of files and lines of code is by release 1.4 with 11.585 source files and 3.966.466 LOCs. Up to this release source code has been added permanently due to addition of new features or extension of existing features. Then, from release 1.4 to 1.6 the amount of source code decreased by 405 source code files (151.293). In particular, several .c and .cpp files have been removed or sourced out to libraries. In the next release the source code again increased.

The Mozilla release history data comprised 494.730 modification reports (CVS log entries) of 40.884 files obtained from Mozilla's CVS repository[1]. The number of files included all moved and deleted files. Regarding reported problems we processed 255.310 problem reports retrieved from the Bugzilla repository[2].

For demonstrating our approach we narrow the Mozilla case study down to a selected set of seven software modules that implement the handling of the content and layout of web pages. These modules are amongst the Mozilla core modules. Table 8.2 lists the selected software modules together with corresponding source code directories containing their implementation. The mapping between modules and source code directories has been derived from Mozilla's design documentation[3].

Module	Source Directories
MathML	layout/mathml
New Layout Engine	layout/base, layout/build, layout/html
XPToolkit	content/xul, layout/xul
DOM	content/base, content/events, content/html/content, content/html/document, dom
New HTML Style System	content/html/style, content/shared
XML	content/xml, expat, extensions/xmlextras
XSLT	content/xsl, extensions/transformiix

Table 8.2: Mozilla content and layout modules and corresponding source code directories.

In the following section we provide the details about obtaining the data from the different sources and building the data models. They contain the facts about the implementation and evolution of the content and layout modules of Mozilla.

8.2 Preparing the ArchView Repository

The basis for the ArchView repository is formed by the information stored in the source code of the different releases and Mozilla's CVS and Bugzilla repositories.

[1]http://www.mozilla.org/cvs.html
[2]https://bugzilla.mozilla.org
[3]http://www.mozilla.org/owners.html

8.2.1 Source Code Model

In order to extract the source code models of each Mozilla release we first checked out the complete source code of each release from Mozilla's CVS repository. We then configured and compiled each Mozilla release to generate the header files out of `.idl` files and the Makefiles for the C/C++ compiler. In the configuration for the compiler we selected *Linux* as the target platform, the GNU C/C++ compiler, and Mozilla's standard components as pre-configured by the Mozilla developers.

For parsing the C/C++ source code of each release we used the Imagix-4D[4] tool. The tool comes with a C/C++ parser that does a full semantic parsing of source files, analyzing all the symbols in the code. The Imagix-4D parser is compiler independent and is able to accurately analyze source code developed for a number of C/C++ compilers. Emulation of compilers is facilitated by the use of compiler configuration files. They contain the compiler specific settings, such as the path to system include files, type definitions, and macro definitions.

Regarding the Mozilla source code we selected the `Linux gcc` as basis configuration. The Mozilla specific compiler settings were added to this file. To get these settings we inspected the different Makefiles of each Mozilla module and obtained the additional include paths, compiler directives, and the set of files to exclude from the parsing process. For instance, we added the directories that contain Mozilla specific header files to the system include path. Mozilla specific compiler directives (*i.e.,* #defines) were taken from the `mozilla_config.h` file. Mozilla's Makefiles contain also the list of directories and files to include. This allowed us to determine the list of files to exclude from the parsing process. For instance, files that contain platform specific (*e.g.,* Microsoft Windows) source code.

The Imagix-4D parser stores extracted source code facts to its Imagix-4D database that is a proprietary database that can not be directly accessed from outside the Imagix-4D tool. Using our Imagix-4D plug-in we exported the database of each parsed source code release to an E-FAMIX compliant (Rigi Standard File) RSF file. Exported facts include the source code entities and relationships as specified by the E-FAMIX meta model and source code metrics computed by the Imagix-4D parser.

We used the RSF file as an intermediate representation of a source code model because this data format is understood by several reverse engineering tools, such as Rigi [MK88], [Won98] or grok [FH00]. We used these tools to edit, filter, cleanse, and visualize exported source code models. For instance, we used grok to compute the transitive closure of the `contains` relationships which was needed in the data abstraction step. And we used Rigi to add the module structure to each of the source code models. The structure is defined by the module-source directories relation as given by Table 8.2.

The cleansed RSF file of each source code release was input to a Perl script that generated the SQL file which specified the layout and the contents of an ArchView source code model database of one Mozilla release. We imported each SQL file issuing `mysql dbname < dbname.sql` command that created and filled the source code model database of a Mozilla release.

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

Table 8.3 provides the measured size metrics of number of extracted source code entities of the seven Mozilla modules per release. Size metrics represent absolute values computed from the snapshots that we took from the Mozilla source code at each particular release date. In its recent release 1.7 the content and layout modules implementation comprises 1.321 C/C++ source and header files (NOF) that are contained by 145 directories (NOD). The files contain 1.677 classes (NOC) with 22.130 methods (NOM) and 4.983 attributes (NOA), and 1.433 global C functions (NGF) and 1.950 global variables (NGV).

Release	NOD	NOF	NOC	NOM	NGF	NFM	NOA	NGV	NOV
Mozilla 0.92	146	1.212	1.369	21.018	1.610	22.628	4.361	1.789	6.150
Mozilla 0.97	147	1.242	1.404	20.189	2.668	22.857	4.486	1.996	6.482
Mozilla 1.0	159	1.347	1.587	22.369	1.702	24.071	5.087	2.079	7.166
Mozilla 1.3a	154	1.364	1.677	23.161	1.420	24.581	5.159	2.092	7.251
Mozilla 1.4	147	1.317	1.681	22.487	1.472	23.959	4.964	2.033	6.997
Mozilla 1.6	146	1.317	1.687	22.657	1.443	24.100	5.000	1.943	6.943
Mozilla 1.7	145	1.321	1.677	22.130	1.433	23.563	4.983	1.950	6.933

Table 8.3: Size metric values of Mozilla's content and layout modules of the seven releases.

The measured value of the size metrics of the seven different releases indicate that the number of source code entities of the implementation of the content and layout modules increased up to release 1.3a and after that slightly decreased. For instance, the number of functions and methods (NFM) increased to 24.581 in release 1.3a and then in the next 3 releases decreased by more than 1.000 functions/methods down to 23.563. Consequently, first functionality was added to the seven modules which then was consolidated in the latter 3 releases.

8.2.2 CVS and Bugzilla Data

For retrieving the modification and problem reports we applied the RHDB Populator tool. The tool traverses the Mozilla source code tree and for each file obtained the modification reports (MRs) from Mozilla's CVS repository. Each report was parsed with respect to the key information as described in the Section 5.4. Extracted facts were stored in the ArchView repository.

Next, we connected to Mozilla's Bugzilla repository and retrieved the problem reports (PRs) in XML format issuing the `wget` command. For instance, to retrieve bug number 12345 we issued `wget https://bugzilla.mozilla.org/xml.cgi?id=12345`. The downloaded XML files were parsed and extracted facts about the problem reports were stored in the repository.

Overall we retrieved 494.730 MRs from the Mozilla CVS repository and 255.310 PRs from the Bugzilla database. The number of modification and problem reports are accumulated over time. Modification and problem reports start with 28th of March, 1998 when the new Mozilla project was set up[5]. The number of MRs and PRs that were involved in the evolution of the source files of Mozilla's content and layout modules are listed by Table 8.4.

[5]http://www.mozilla.org/roadmap.html

Release	NMR	NPR	NPR$_{p1}$	NPR$_{critical}$	NPR$_{resolved}$	NPR$_{fixed}$
Mozilla 0.92	42.749	15.946	2.323	1.808	2.732	14.316
Mozilla 0.97	48.729	20.769	3.174	2.009	5.383	18.219
Mozilla 1.0	56.714	24.145	3.622	2.397	7.183	21.020
Mozilla 1.3a	59.539	26.230	3.970	2.512	8.907	22.673
Mozilla 1.4	62.702	28.003	4.318	2.621	10.423	24.159
Mozilla 1.6	66.441	29.881	4.650	2.702	12.114	25.926
Mozilla 1.7	69.402	30.650	4.728	2.734	12.835	26.528

Table 8.4: Accumulated number of modification and problem reports obtained for the source files of the content and layout modules. The number of PRs is further detailed in four categories comprising the numbers of PRs with highest priority `p1`, severity `critical`, status `resolved`, and resolution `fixed`.

For instance, in the time from the 28th of March, 1998 to the 28th of June, 2001 (release of Mozilla 0.92) the Mozilla developers committed 42.749 change logs to Mozilla's CVS repository for the source files implementing the content and layout modules. Then, from release 0.92 to release 1.7 another 26.653 modification reports were added (see NMR column of Table 8.4). Regarding the problem reports different attributes exists in the Bugzilla database that allow for a further categorization of PRs by its status, resolution, severity, and priority. Table 8.4 lists the total number of PRs and the numbers of four important categories of problem reports:

- `p1`: most important PRs with highest priority.
- `critical`: PRs with serious negative impact on the system (*i.e.*, system crashes, loss of data, and severe memory leak).
- `resolved`: PRs for which a resolution has been taken, and it is awaiting verification.
- `fixed`: PRs for which a fix is checked into the source code tree and tested.

Similar to the size metrics of source code models the problem report metrics were computed based on snapshots that we took from the Bugzilla database at certain points in time (*i.e.*, when we filled the release history database). However, problem report attributes, such as *status* and *resolution* change over time because work is going on on bugs. For instance, when a problem is reported the report gets the status `new`. Then the problem is `assigned` to a developer who works on it and provides a resolution. When there is a resolution the problem reports gets the status `resolved`. The resolution has to be verified (*i.e.*, tested) and finally if successful the status of the problem report is changed to `verified`. In this case study we did not take into account the activity log and left out the metrics concerning the status and resolution of problem reports. This is part of our future work.

Referring to the metrics listed in Table 8.4 we gained that in the earlier Mozilla releases more problems (bugs) have been reported than in recent releases. For instance, up to release 0.97 in total 20.769 PRs were entered into the Bugzilla database that concern problems in the content and layout modules. With respect to Mozilla release 0.92 this presents an increase by 4.823 new PRs. 851 PRs out of the added PRs 851 were of highest priority. In contrast, from release 1.6 to 1.7 the increase is by 769 PRs in total. This is a significantly smaller amount of PRs taking into account that the amount of time spent for moving from release 1.6 to 1.7 is about the same as from release 0.92 to 0.97. The other categories of problem report metrics show a similar tendency.

A possible interpretation is that Mozilla's content and layout modules were more "buggy" in the earlier versions when developers introduced new layout mechanisms and dom standards than in recent releases.

Linkage of Modification and Problem Reports

Using the algorithm described in Section 5.4 we established the links between modification and problem reports. In total 323.409 links were computed between 225.978 different modification and 36.474 different problem reports. Linked MRs involved 21.025 files out of all files managed by Mozilla's CVS repository. Our validation of these links indicated that 91% of the referenced reports fell either into the group `fixed/resolved` or `fixed/verified`. The other categories were sparsely filled which may indicate a positive false detection or incorrect tracking status of PRs. By comparing this data with all reports downloaded from the Bugzilla database, we recognized that a large number of PRs within the groups `duplicate`, `invalid`, `won't fix`, and `works for me` were not referenced. These results supported our assumption in two ways: 1) only records about PRs were made that have an effect on the CVS repository; 2) a significant number of the identified IDs was valid if we presumed that `duplicate`, `boxed`, etc. reports were equally distributed over the ordinary scale of report IDs.

Therefore, our conclusion from linked modification and problem reports were: references to PRs are available in a sufficient quantity and quality that allowed for further analysis. More details are presented in [FPG03a] in which we analyzed the Mozilla release 1.3a.

Change Coupling

For the computation of the change coupling relationships between source files we configured the Algorithm 5.6 presented in Section 5.4 with a search radius of 15 minutes and applied it to the modification reports stored in the ArchView repository. In total the algorithm detected 4.058.473 change coupling relationships between 16.029 different source files. For each relationship the algorithm also output the total number of involved MRs, the number of MRs for which there is a link to a problem report, and the sum of lines added and deleted. These attributes characterize the strength of a change coupling relationship. Both, relationships and their attributes were stored in the ArchView repository.

For more information on the computation of change couplings we refer the reader to the related publications of our group [PFG05] and Zimmermann *et al.* [ZWDZ04].

8.2.3 Data Integration

This step is concerned with establishing the links between the file identifiers of the different source code models and the release history data. Table 8.5 provides the numbers of established links including also the number of multiple linked files.

Release	NOF	#links	#SCM files	#RHDB files	#multi. SCM	#multi. RHDB
Mozilla 0.92	1.212	1.422	1.165	1.394	225	28
Mozilla 0.97	1.242	1.461	1.193	1.433	233	28
Mozilla 1.0	1.347	1.606	1.318	1.578	242	28
Mozilla 1.3a	1.364	1.614	1.348	1.610	233	4
Mozilla 1.4	1.317	1.558	1.308	1.554	221	4
Mozilla 1.6	1.317	1.559	1.315	1.559	218	0
Mozilla 1.7	1.321	1.559	1.319	1.557	213	2

Table 8.5: Number of established links between files of source code models and release history data together with the numbers of multiple linked files.

For release 0.92 and 0.97 the tool established around 96% of the links between file entities of the different data models (1.165 out of 1.212 SCM files). The missing 4% of the links basically were due to files that were generated during Mozilla's configuration and pre-compilation steps. Most of these files had a dummy entry in the release history database and zero modification reports that caused the filtering of these files. In the latter four releases these files were not part of the source code release that was input into the Imagix-4D parser. This led to a recall ratio of more than 99.8% (1.319 out of 1.321 SCM files).

Multiple Linked File Entities

The fact that several entries for the same file existed in the source code and also release history data models led to multiple links between file entities. The `#mult. SCM` column of Table 8.5 lists the number of multiple linked files of each source code model. For instance, in release 0.92 225 files have more than one pendant in the release history data model. An investigation of these files in the release history data yielded that multiple entries are due to the reasons mentioned in Section 5.5. All entries of the file in the release history data were linked without any false positive link. Hence, it is possible to navigate from each linked file in the source code model to *all* its modification reports in the release history data model.

In the other direction the number of multiple linked release history data files was 28 in the first three releases (see column `#multi. RHDB` of Table 8.5). This number decreased to 2 files in subsequent Mozilla releases. A check of these file entities in the source code models yielded that multiple links were due to duplicated entries for the same file in the source code models. It turned out that the duplicates were generated by the Imagix-4D parser because several exemplars of the same file existed in different source code directories and the parser was not able to distinguish between them. Most of these duplicated entries referred to header files. For instance, in release 0.92 the header file `dom.h` existed in the two source code directories

`../xml/dom/` and `../xml/dom/standalone/` which led to two different file entities in the source code models. But, because links were established for each of the duplicated file entity also the navigation to the corresponding release history data was possible from each entity.

Our conclusion is that the number of false positive links is low and as a consequence established links are provided in reasonable quality. This facilitates a detailed integration of the different source code and release history data models as is needed by our approach.

Integration of Change Coupling Relationships

Source code models contain the file entities that existed at the point in time of a release. In contrast, change couplings occur during the development a new release hence at arbitrary points in time. To integrate the change coupling relationships with the source code model data the different change coupling relationships are aggregated. The aggregation involves the accumulation of the metric values of a change coupling relationship (see Table 6.6).

Based on the containment hierarchy we query the source code model of each release to get the list of source files implementing the content and layout modules. With this list we apply a query to the mapping table that returns the list of file identifiers known by the release history data model. Using this list of identifiers and the dates of two subsequent releases we finally query the change coupling table. The query aggregates the matched change coupling relationships and sums up the number of involved MRs (RNMR), the number of MRs linked with a problem report (RNMRPR), and the number of lines added and deleted (RENT).

For each aggregated coupling relationship a record is created in the database that holds the references to the two files of the source code model and the accumulated attribute values. Table 8.6 lists the numbers of integrated change coupling relationships (#rels), number of affected file entities of the source code models, and the file with the highest number of change couplings.

Release	#rels	#affected files	#rels per file	peak
Mozilla 0.92	59.985	908 (74,92%)	66,06	428 (nsXULElement.cpp)
Mozilla 0.97	39.279	837 (67,39%)	46,93	358 (nsPresShell.cpp)
Mozilla 1.0	30.401	807 (59,91%)	37,67	341 (nsPresShell.cpp)
Mozilla 1.3a	25.168	754 (55,28%)	33,38	321 (nsPresShell.cpp)
Mozilla 1.4	51.866	840 (63,78%)	61,75	412 (nsCSSStyleRule.cpp)
Mozilla 1.6	40.613	676 (51,33%)	60,08	425 (nsCSSFrameConstructor.cpp)
Mozilla 1.7	27.648	642 (48,60%)	43,07	271 (nsPresShell.cpp)

Table 8.6: Number of integrated change coupling relationships per Mozilla release.

The numbers listed by Table 8.6 indicate that regarding the content and layout modules most of the change couplings occurred in the development of the two releases 0.92 (59.985 created relationships between 908 files) and 1.4 (51.866 created relationships with 840 files). Relatively low change coupling arose in the releases 1.0, 1.3a and 1.7. The rate of affected source files of total files decreased from 74,92% in release 0.92 to 48,6% in the most recent release 1.7. This indicates improvements in the implementation of Mozilla's content and layout modules.

The `#rels per file` shows the number of change coupling relationships normalized by the number of affected source files. The largest values are from the releases 0.92, 1.4, and 1.6. For instance, in release 1.4 the number of change couplings of each affected source code model file is 61.75 on average. The `peak` column lists the largest number of change coupling relationships of a source file of each release. For instance, in the time between release 1.4 and 1.6 the file `nsCSSFrameConstructor.cpp` was changed together with 425 out of 1.317 source files or 32.37% respectively.

Conclusion of Data Integration

Links between the source code and release history data models were established in reasonable quality and stored in the mapping tables. Multiple entries in both the release history and source code data models were resolved by our mapping algorithm. Using the name of file entities turned out to provide a reasonable work around for this issue.

The result of the ArchView data integration step a data model that facilitates the navigation from source code model entities to corresponding release history data and vice versa. For instance, we can navigate from the source code model data of a file to its modification and problem report data in the release history including computed source code and evolution metrics as well as dependencies with other source files. The model forms the basis for our abstraction, analysis, and visualization algorithms.

8.2.4 Data Abstraction

For the abstraction of the data models we selected the class, file, and module level. The enriched source code model of each release was input to the abstraction tool that implements the abstraction algorithm presented by Listing 6.1. According to the abstraction level we configured the tool to:

- Abstract relationships of lower-level entities as specified by the containment hierarchy model;
- Compute the size and complexity metrics of classes, files, and software modules;
- Compute the evolution metrics out of modification and problem report data; and
- Compute the source code and change coupling metrics of each class, file, and software module.

Concerning the abstracted relationships Table 8.7 lists the different types of relationships per abstraction level. Additionally, the right column lists the types of relationships that we considered for the computation of the coupling metrics.

Abstracted relationships and computed metric values were stored in the ArchView repository. This concludes the data preparation phase. The results were integrated and abstracted data

Level	Rel. types to aggregate	Rel. types for coupling measurements
Class	aggregates, invokes, overrides, accesses, hasType	inherits, aggregates, invokes, overrides, accesses, hasType
File	all relationship types of Class level plus inherits	all of Class level plus couples and includes
Module	all relationship types of File level plus couples and includes	same as for File level

Table 8.7: Abstraction levels and types of relationships considered for abstraction as well as for computation of the coupling metrics of implementation units.

models enriched with a set of source code and change coupling metrics and other source code metrics as presented in Section 6.3. In the next sections we demonstrate the usage of the data models to analyze and visualize certain aspects of the implementation of Mozilla's content and layout modules ant their evolution.

8.3 Views on Mozilla Release 1.7

In the first analysis step we concentrate on the most recent Mozilla release 1.7. The objective of the analysis is to compute views on the integrated data model that address and provide answers to the analysis issues of the scenario described at the beginning of this chapter. The analysis follows the order of these issues starting with highlighting the large and complex entities. For each issue we present views on the module level that are followed by views on the level of source files. First views aid in gaining an initial understanding that next is detailed by the views on the file level. Each analysis issue is concluded by a presentation and interpretation of the findings. Further, when presenting polymetric views we provide the metric mapping in the caption of the figure.

8.3.1 Large and complex entities

To get an initial understanding of the implementation of Mozilla's content and layout modules we start top-town with building views on the system hot-spots. For the composition of these views we apply the CodeCrawler tool [Lan03].

Module Views Figure 8.1 depicts a system hot-spots view on the content and layout software modules using three module size metrics. The node width represents the number of contained global variables and class attributes (NOV), the height represents the number of contained functions and methods (NFM), and the color represents the number of files (NOF).

From Figure 8.1 we gain that the largest module is `DOM` with 1.522 global variables and attributes and 11.398 functions contained in 492 source files. The second largest module is `NewLayoutEngine` that contains 1.394 variables and 4.404 functions in 229 source files. The

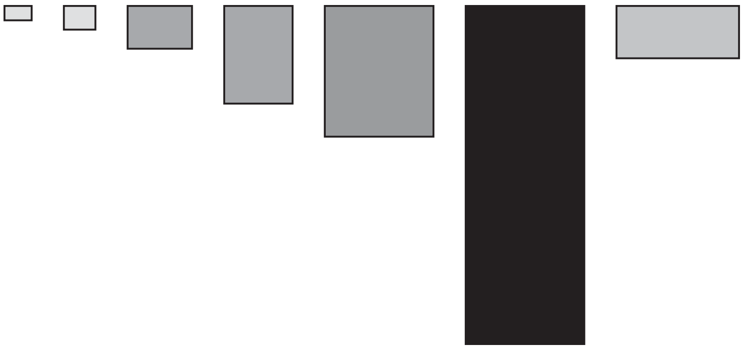


Figure 8.1: System hotspots view on the Mozilla 1.7 content and layout modules with module size metrics. Node: width=NOV; height=NFM; color=NOF; order by NOV.

module with the highest amount of variables is `NewHTMLStyleSystem` depicted on the right side. It contains 1.572 variables.

Figure 8.2 depicts the same set of modules but this time with two complexity metrics. The width of nodes represents the McCabe cyclomatic complexity metric (CCMPLX). The height represents the Halstead program difficulty metric (HALDIFF) and the node color maps to the number of functions (NFM) implemented by a module.

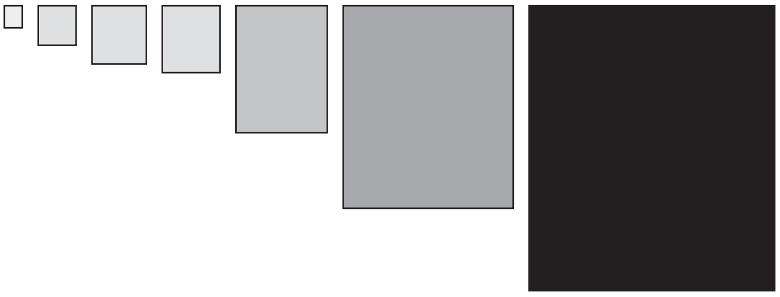


Figure 8.2: System hotspots view on Mozilla 1.7 content and layout modules with module complexity metrics. Node: width=CCMPLX; height=HALDIFF; color=NFM; order by CCMPLX.

`DOM` is the most complex software module with an accumulated cyclomatic complexity value of 26.873 and an accumulated Halstead program difficulty value of 51.425. It is drawn as a large dark rectangle. The color gradient of the nodes indicates that the modules with the largest number of functions are the most complex ones.

File Views Software modules as used by the Mozilla project are rather abstract entities that consist of a number of source files. To get more into detail and point out finer-grained hot-spots we decrease the abstraction level down to source files.

Figure 8.3 depicts the 1.321 C/C++ source files implementing the seven software modules. Files with a high number of variables and functions are highlighted by mapping the NOV and

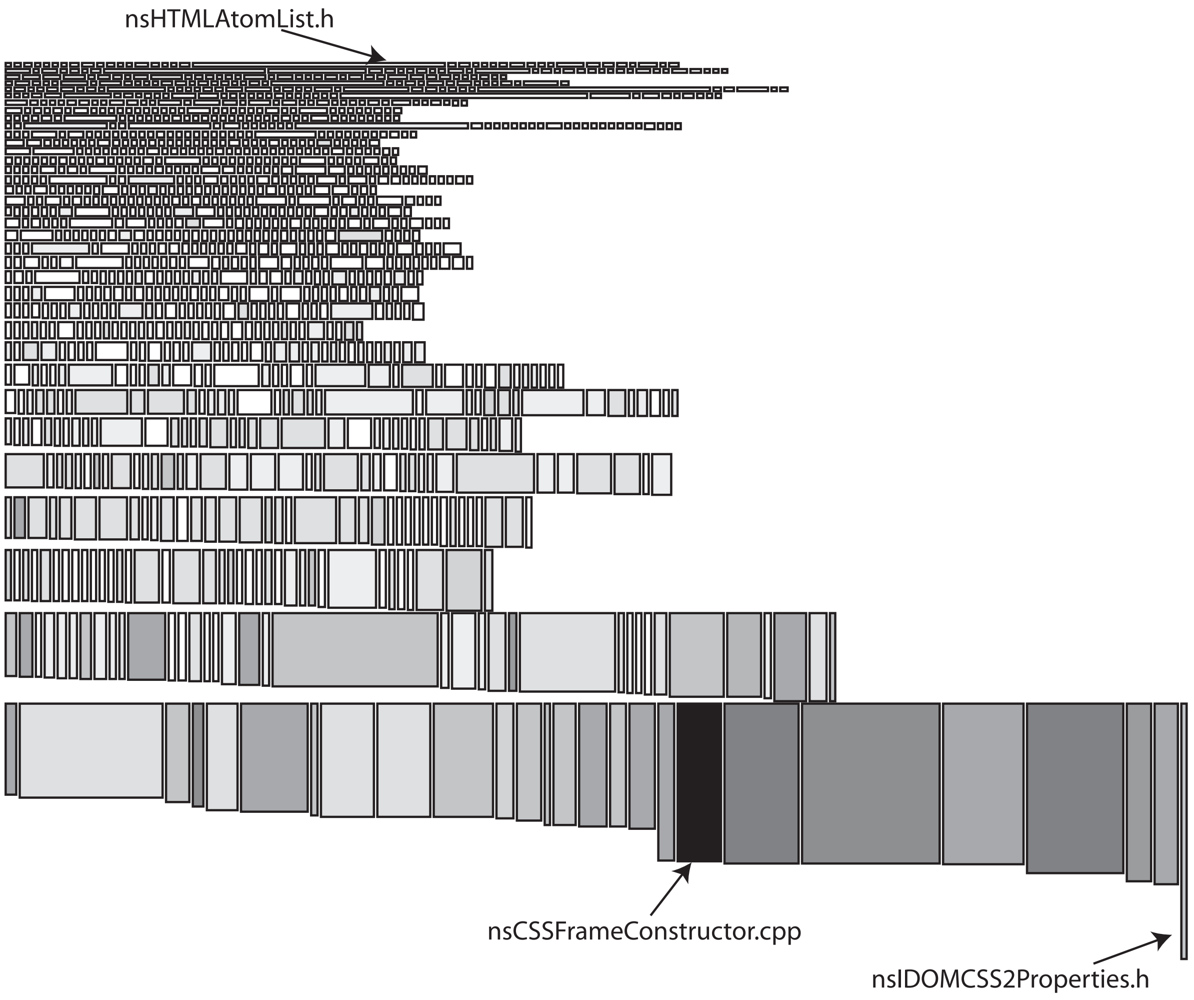


Figure 8.3: System hotspots view on Mozilla 1.7 content and layout source files with file size metrics. Node: width=NOV; height=NFM; color=LOC; order by NFM.

NFM metrics to the width and height of nodes. The color of nodes maps to the length of files in lines of code (LOC).

The source files that contain a large number of functions are listed on the bottom of the figure. For instance, the file with the highest number of functions is `nsIDOMCSS2Properties.h` which contains 340 functions and is 3.516 lines long. Because it contains zero variable declarations it is drawn as a long small rectangle on the right of the bottom row. As highlighted by the black color the largest file is `nsCSSFrameConstructor.cpp` with 13.320 lines of code. It implements 208 functions and declares 36 global variables and attributes.

Source files that contain an large number of global variables attributes but almost zero functions are depicted in the top rows of the graph. They are drawn as flat small rectangles. Most of them are header files that declare constants for the different tag elements, such as for 233 HTML tags in `nsHTMLAtomList.h`, or 214 MathML tags in `nsMathMLAtomlist.h`.

Figure 8.4 depicts the same set of files with complexity metrics mapped to the width and height of nodes. The width of nodes depicts the accumulated McCabe cyclomatic complexity metric (CCMPLX), the height of nodes depicts the Halstead program difficulty (HALDIFF) metrics, and the color depicts the number of contained functions (NFM). Using this mapping files with a high complexity are drawn as large dark rectangles.

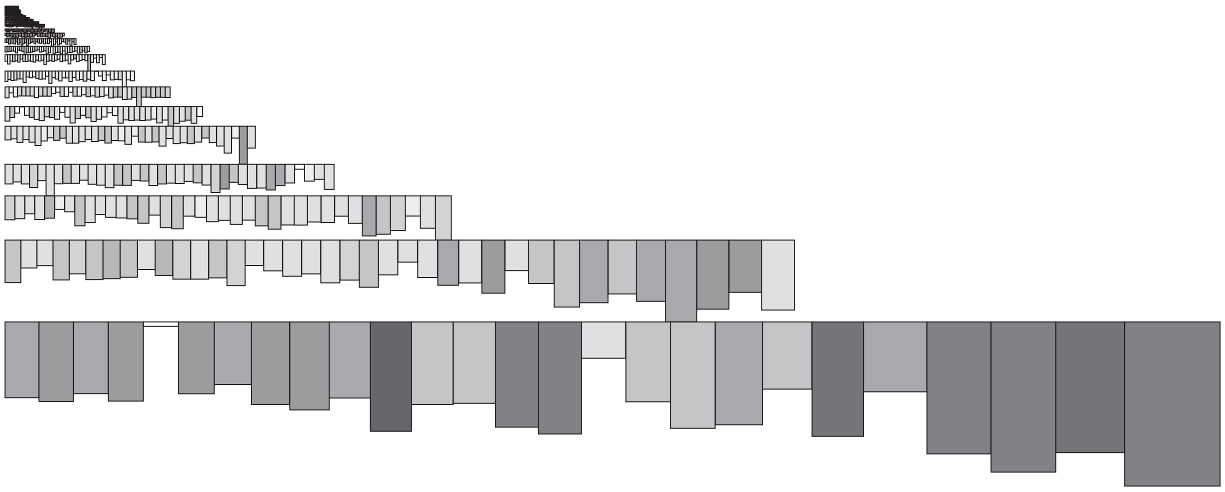


Figure 8.4: System hotspots view on Mozilla 1.7 content and layout source files with file complexity metrics. Node: width=CCMPLX; height=HALDIFF; color=NFM; order by CCMPLX.

The last row of the figure highlights source code files with high complexity. The last four nodes depict (from right to left) the files `nsCSSFrameConstructor.cpp`, `nsGlobalWindow.cpp`, `nsTableFrame.cpp`, and `nsSelection.cpp`. The accumulated cyclomatic complexity of these files is above 1.200 and each contains the implementation of more than 200 functions.

Files with low complexity are located in the top rows of the graph. Most of them are header files (.h). They contain less implementation hence their complexity is low.

Results Presented hotspots views give an initial insight into the implementation of Mozilla's content and layout features by depicting involved elements together with size and complexity metrics. The views clearly point out the `DOM` module as the largest and most complex module. With respect to the source files we identified 40 files that contain 100 functions and more. Almost all of these files are highly complex as highlighted by the graphical node attributes (*i.e.*, width, height, and color). They are the files that implement most of the content and layout functionality hence are those most likely to be involved in evolution and maintenance activities.

8.3.2 Frequently modified and "buggy" entities

In addition to the size and complexity metrics we computed evolutionary metrics that concern the number of reported problems and the frequency of changes. Visualizing these metric values yields the list of software modules and source code files to which most of the work was dedicated to. For the analysis we take into account the time period between the releases 1.6 and 1.7.

Another analysis aspect concerns the check of the hypothesis that “complex elements are more vulnerable to problems and modifications than are elements with low complexity”. The following set of module and file views deal with this aspect and highlight the elements with exceptional high metric values.

Module Views Figure 8.5 shows the seven software modules with the values of evolution and complexity metrics. The width of nodes represents the number of modification reports (NMR), the height represents the number of problem reports (NPR), and the color attribute is used to represent the accumulated cyclomatic complexity of the modules (CCMPLX).

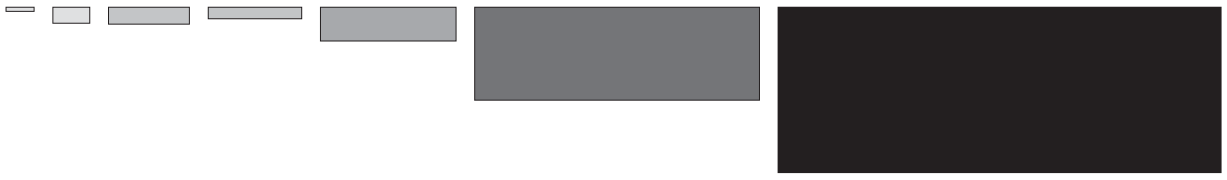


Figure 8.5: Modification hotspots view on Mozilla 1.7 content and layout modules with evolution and complexity metrics. Node: width=NMR; height=NPR; color=CCMPLX; order by NMR.

From the graph we obtain that the `DOM` module is the module with the highest number of reported problems and modifications. In detail from release 1.6 to 1.7 1.496 new modification and 554 problem reports were assigned to this module. The order and the color gradient of nodes indicate that the complex modules are modified more frequently than modules with low complexity.

File Views Figure 8.6 shows the graph with the 1.321 source files enriched with values of evolution and the cyclomatic complexity metrics. The width of nodes represents the number of modification reports (NMR), the height of nodes represents the number of problem reports (NPR), and the color represents the accumulated cyclomatic complexity metric (CCMPLX).

Using this configuration the view highlights the source files that have been most worked on during the development of the new release 1.7. These are: `nsCSSFrameConstructor.cpp` (53 MRs), `nsPresShell.cpp` (47 MRs), `nsGenericHTMLElement.cpp` (38 MRs), and `nsHTMLDocument.cpp` (37 MRs). These files are also amongst the files with a large number of reported problems. In detail, for `nsCSSFrameConstructor.cpp` 16 PRs, for `nsPresShell.cpp` 14 PRs, for `nsGenericHTMLElement.cpp` 18 PRs, and for `nsHTMLDocument.cpp` 19 PRs were reported.

Ordering the nodes by the NMR values the graph highlights the source files with zero or few modifications. These files are positioned at the top of the graph. For instance, from release 1.6 to 1.7 742 out of the 1.321 source files were modified not more than once. Consequently, more than 56% of the files are almost stable. Furthermoer, for 934 files that are about 70% of the

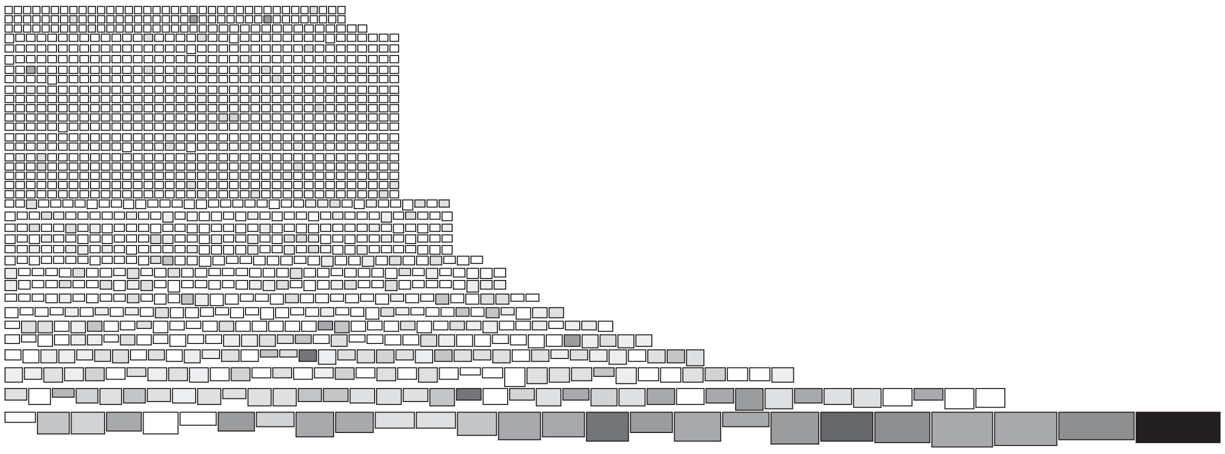


Figure 8.6: Modification hotspots view on Mozilla 1.7 content and layout source files with evolution and complexity metrics. Node: width=NMR; height=NPR; color=CCMPLX; order by NMR.

files no problems were reported. The relation between the width and height of nodes is almost proportional. Hence, the more problem reports are reported for a file the more modifications had to be performed.

The graph provides further clues to verify the hypothesis stated before. According to the order and color of nodes complex files are more vulnerable to problems and are more frequently modified than files with low cyclomatic complexity. But, there is a small number of complex source files that conflicts with the hypothesis. For instance, `xmltok_impl.c` or `xmlparse.c` are rated as complex files but refer to zero MRs and zero PRs. The conclusion is that although the implementation of these files is complex it is stable.

Results From the views presented in this section the user derives the set of software modules and source files to which most of the evolution and maintenance activities were dedicated to. Concerning the content and layout modules the views identified the `DOM` module as the most complex, default prone, and frequently modified module. The view on the source files yielded that 56% of the source files were not touched. In contrast, 19 source files were modified more than 20 times in the observation period from release 1.6 to 1.7. On the problem report side 76% of the investigated source files are stable.

The combination of the modification and problem report metrics with the cyclomatic complexity measure of software modules clearly showed that the cyclomatic complexity is proportional to the number of problems and modifications. The view on the source files further verified this hypothesis with a few exceptions to report. Basically, these exceptions indicated stable source files. Both, stable and unstable elements were highlighted by the system hotspots views.

8.3.3 Views with multiple metrics

In order to reason about the relations between multiple metrics we mapped the values of multiple metrics to graph nodes. This functionality is provided by our Kiviat diagram visualization technique that we applied to compose views with multiple metrics.

Module Views Figure 8.7 depicts the main size, complexity, and evolution metrics of Mozilla's content and layout modules. The order of metrics of each Kiviat diagram corresponds to the three categories of metrics. Metrics that belong together are located side by side: 0..3 - complexity metrics; 4..9 - size metrics; 10,11 - evolution metrics.

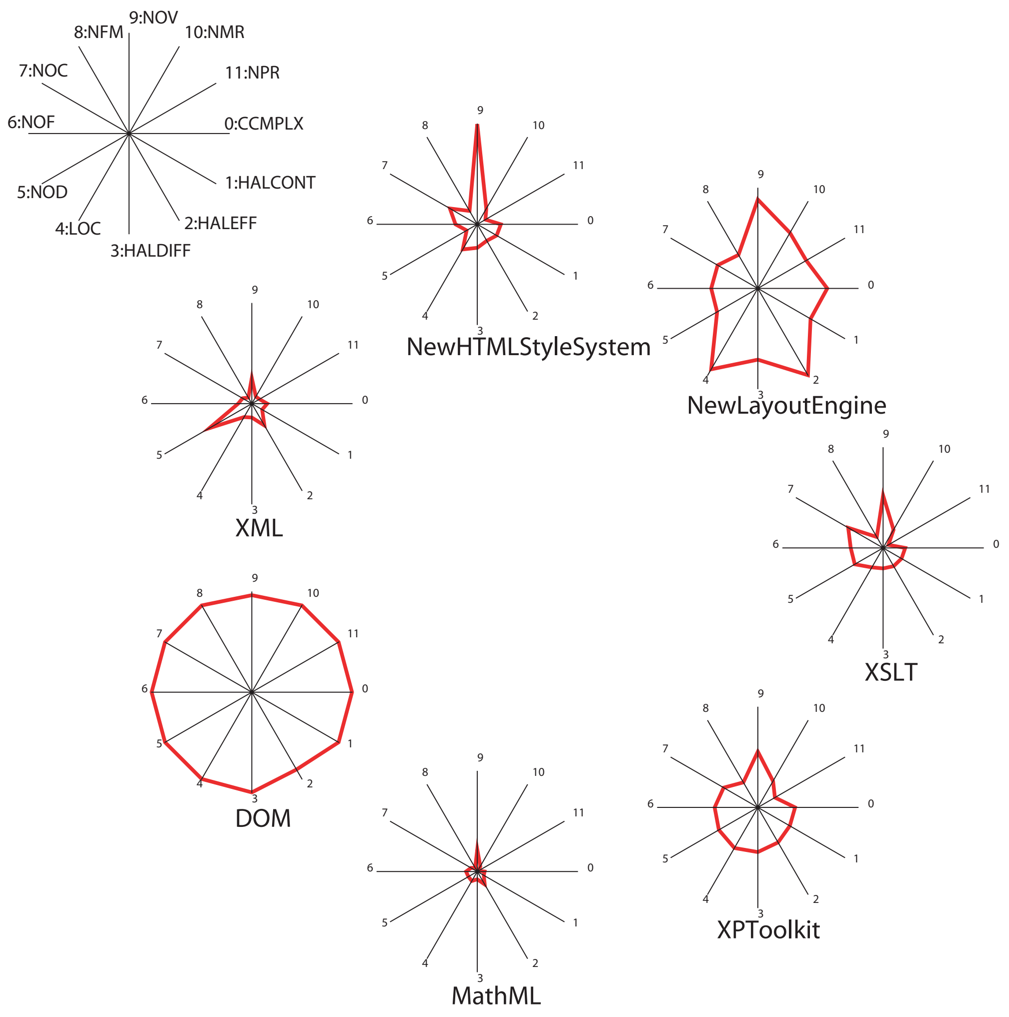


Figure 8.7: Detailed system hotspots view on Mozilla 1.7 content and layout modules with Kiviat diagrams showing evolution, size and complexity metrics.

The graph highlights the `DOM` module as the largest and most complex module that also is changed most frequently. The big circle drawn in the diagram indicates the `DOM` module as a "God Module".

The second largest module is `NewLayoutEngine` followed by `XPToolkit`. Peaks in certain metric values are also highlighted by the Kiviat diagrams. For instance, the diagram representing the `NewHTMLStyleSystem` module shows a peak in the number of contained variables (NOV).

Bugzilla allows the categorization of problem reports into different severity and priority levels. According to this categorization we established evolution metrics that we measured in the Mozilla case study. An excerpt of these metrics are depicted by the Kiviat diagrams of Figure 8.8.

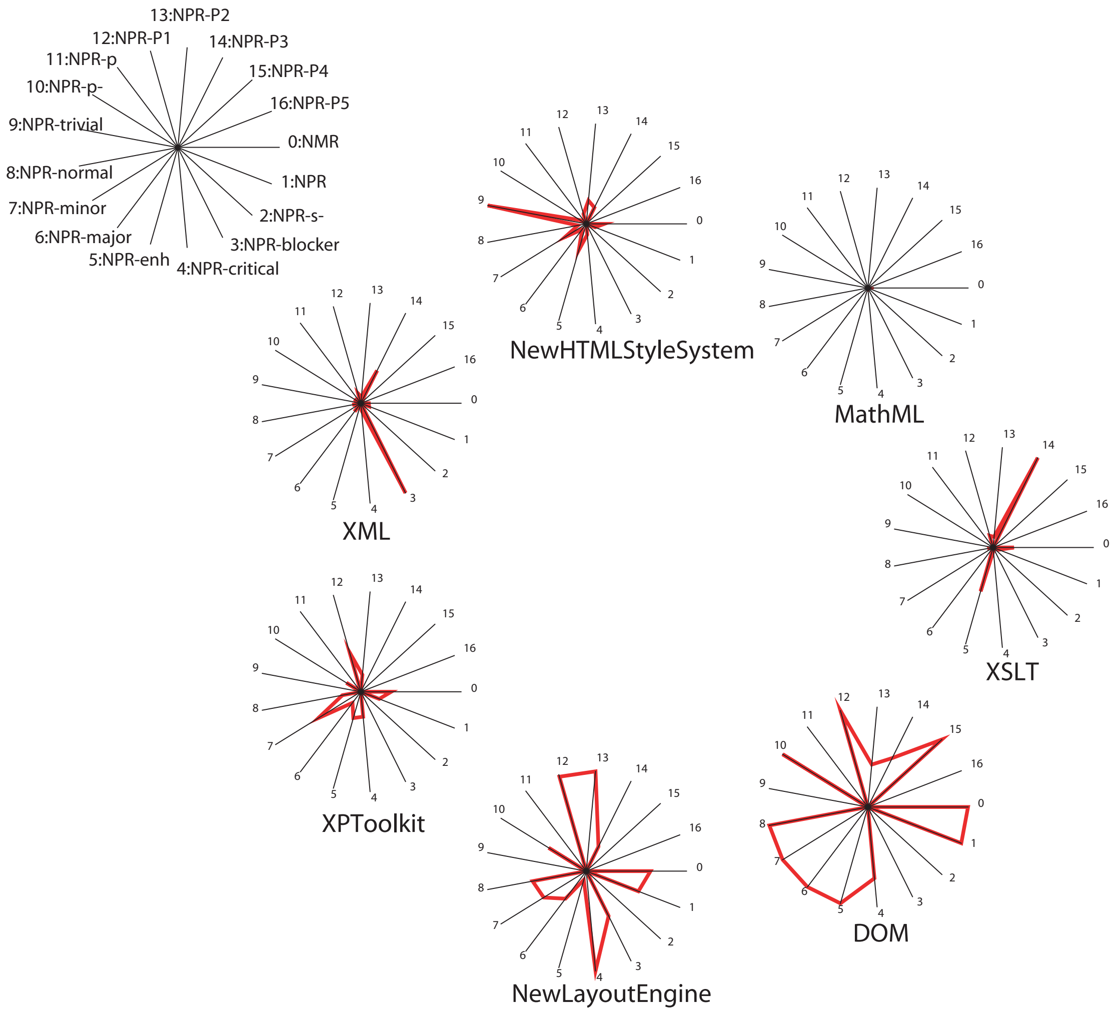


Figure 8.8: Detailed modification hotspots view on Mozilla 1.7 content and layout modules with Kiviat diagrams showing metrics of different categories of problem reports.

As already shown with the system hotspots view the `DOM` module by far is the module with the most problem and modification reports. However, the most critical bugs in the development of release 1.7 have been assigned to the `NewLayoutEngine` module with 28 critical PRs followed by the `DOM` module with 20 critical PRs. The other severity categories again are leaded by the `DOM` module.

The results of measurements that concern the priority of problem reports draw a similar picture. `DOM` was assigned the highest number of PRs with highest priority followed by the `NewLayoutEngine` module.

Certain diagrams contain single peaks, such as the modules `XSLT` for the NPR-P3 metric, `NewHTMLStyleSystem` for the NPR-trivial metric, and `XML` for the NPR-blocker metric. They tend to indicate a high number of problems but basically are due to the algorithm that we use for normalizing the values to the size of the Kiviat diagrams. For instance, the maximum for the NPR-blocker metric is 2. Having another NPR-blocker metric of value 1 results in a normalized value that is 50% of the maximum length.

File Views On the level of source files we investigated similar aspects as on the module level before. However, to maintain the clearness of graphs we concentrate on the critical source code files. Figure 8.9 depicts the top 7 critical source code files. The interest is on the distribution of the metric values of the different problem report categories.

The file with the highest number of assigned problem reports is `nsHTMLDocument.cpp` with 19 PRs. The Kiviat diagram depicts that almost all of these 19 PRs have been rated as `normal` and of low priority. 16 PRs were assigned to `nsCSSFrameConstructor.cpp` but 10 of these reports have been rated of highest priority `P1` and `P2`. This clearly indicates this file as the most critical file. Another two files that are change prone are `nsXULElement.cpp` and `nsPresShell.cpp` (see NPR-P1 metric).

Results In this section we used Kiviat diagrams to visualize a number of different size, complexity, and evolution metrics. On the module level we first concentrated on the size and complexity metrics. The resulting graph indicated the `DOM` module as a “God Module” and the `NewLayoutEngine` as exceedingly complex.

The objective of the second module diagram was to show the distribution of metric values of the different problem report categories as offered by the Bugzilla database. According to the diagrams most of the critical bugs with highest priority were assigned to the `NewLayoutEngine` module directly followed by the `DOM` module. As shown by the graph the center of gravity in maintaining and evolving Mozilla’s content and layout behavior in the time between the releases 1.6 and 1.7 was on these two modules.

On the file level we investigated similar aspects and presented one view on the most change prone source files (*i.e.*, files with the most modification and problem reports). In particular, this view indicated one change prone file which is `nsCSSFrameConstructor.cpp`.

8.3.4 Source code coupling

The next couple of views present information about the coupling relationships between the Mozilla content and layout modules and their contained source files. An interesting aspect to

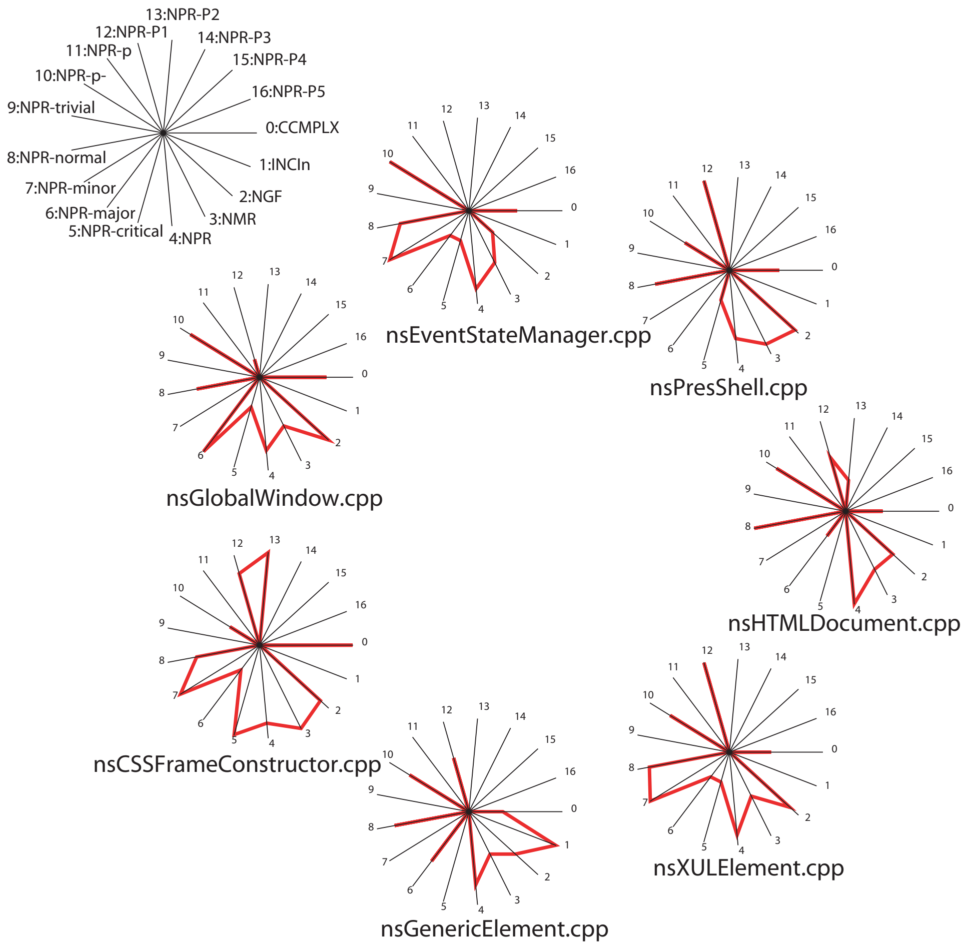


Figure 8.9: Detailed modification hotspots view on frequently modified Mozilla 1.7 content and layout source files with Kiviat diagrams showing size, complexity, and evolution metrics.

highlight is the intermodule coupling – which modules are coupled and how strong are these couplings. Showing the cyclic coupling dependencies is also subject to these views. They indicate "Bad Smells" in the design which should be removed.

The focus of the views is on showing the strong coupling relationships. Therefore, we apply filtering by thresholds to clarify views. The values for the thresholds are given in the caption of the figures.

Module Views On the level of modules we are interested in the *uses* and *inheritance* relationships as well as the change coupling between the software modules. Figure 8.10 depicts the function calls crossing the module boundaries. The width of nodes represents the fan-out of modules (IFan-out), the height the fan-in (IFan-in), and the color represents the number of contained methods and functions (NFM). To highlight strong coupling relationships we map the number of aggregated function calls (RNAI) to the width of edges.

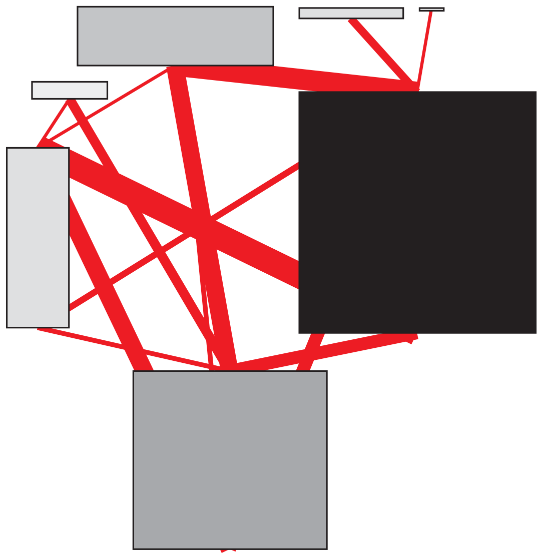


Figure 8.10: Source code coupling view (invokes) on Mozilla 1.7 content and layout modules. Node: width=IFan-out; height=IFan-in; color=NFM; Arc: width=RNAI; Arc-filter: RNAI<50.

The view emphasizes the strong coupling relationships, such as between the DOM and the XPToolkit modules or between the DOM and the NewHTMLStyleSystem modules. The first edge represents 702 and the latter 870 aggregated method calls. Another module strongly coupled with these three modules is the NewLayoutEngine module. The color of nodes and the width of edges indicate that strong couplings basically affect the large modules.

We mapped the values of fan-in and fan-out metrics of function calls to the width and height of nodes to distinguish the modules into service providers and requesters of or both. For instance, the XML, XSLT, and MathML modules are marked as service requesters. In contrast to these modules, the NewHTMLStyleSystem module is a service provider. The module contains 1.760 functions whereby 322 (18,29%) out of them are used by other modules. On the other side, only 108 functions (6,14%) of this module call functions of other modules. The DOM and NewLayoutEngine module play both roles because the width and height of nodes representing the two modules are of equal size.

The graph also depicts several cyclic dependencies between modules that indicate shortcomings in the design. The strongest cycle is between the DOM and NewLayoutEngine modules spanned by two aggregated invokes relationships with 389 and 432 aggregated function calls.

Another cyclic call dependency is between the NewLayoutEngine and XPToolkit modules. It consists of a strong and a relatively weak coupling relationship. The weak relationship comprising 137 function calls is the candidate to remove for resolving the cyclic coupling. We will come back to this issue when presenting the views on the file-level.

The next view depicted by Figure 8.11 shows the inheritance structure of the content and layout modules. The width and height of nodes represent the number of contained classes (NOC), the color the fan-in of overriding methods (OFan-in). The width of arcs represents the number of aggregated inheritance relationships crossing the module boundary (RNAIH).

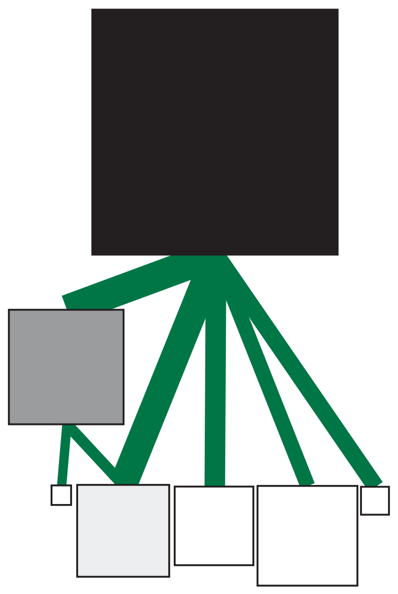


Figure 8.11: Source code coupling view (inherits) on Mozilla 1.7 content and layout modules. Node: width=NOC; height=NOC;color=OFan-in; Arc: width=RNAIH; Arc filter: RNAIH $\leq$ 5.

The view presents the `DOM` module as the basis module from which all other modules inherit methods and attributes. The OFan-in value indicates that `DOM` is the module whose methods are overridden most by methods of other modules. According to these metric values the inheritance of the content and layout module is soundly implemented.

The cyclic coupling analysis of the inheritance relationships revealed only minor design flaws that are subject to re-factor. For instance, there are several light-weight cyclic coupling dependencies due to class inheritance that are of strength up to 3. On the level of source files we will further elaborate on this.

File Views The objective of the file-level views is to visualize the files that are most involved in the coupling between the content and layout modules of Mozilla release 1.7. Furthermore, we provide details about the cyclic coupling relationships that have been detected on the module-level. Coupling dependencies include the different source code relationships, such as file includes, class inheritances, method calls, or variable accesses.

Figure 8.12 depicts the intermodule coupling by method calls. Nodes represent the source files whereby the width and height of nodes maps to the number of functions (NFM) metric. Edges represent method invocations whereby the width maps to the number of aggregated method calls between two source files (RNAI). To clarify the graph we applied a threshold filter of 30 to the RNAI metric - relationships with less than 30 aggregated functions calls are filtered.

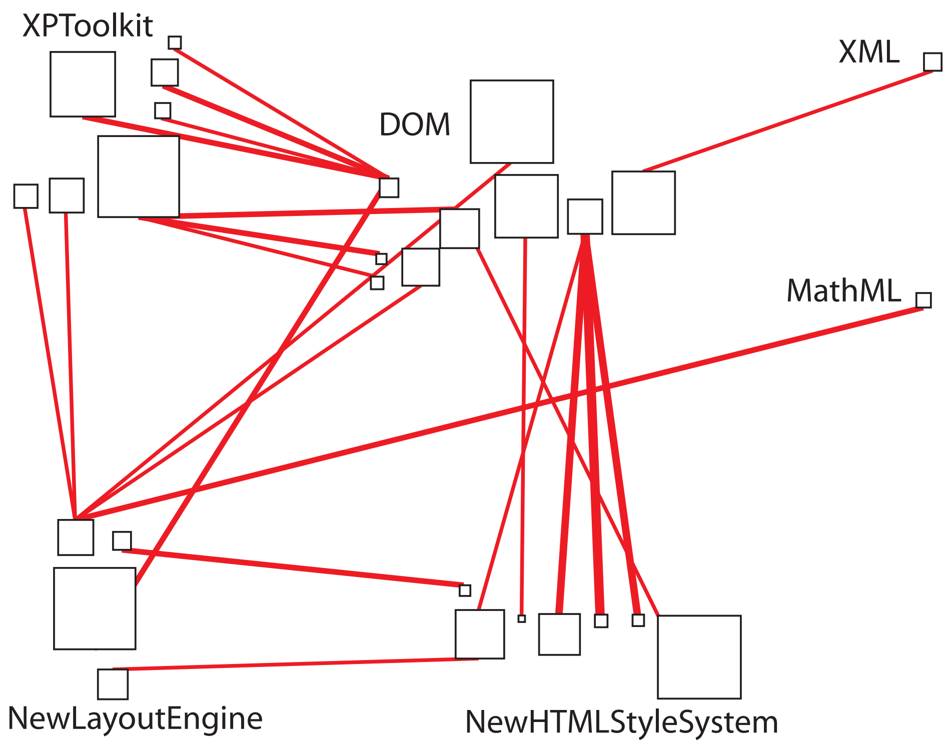


Figure 8.12: Source code coupling view (invokes) on Mozilla 1.7 content and layout source files. Node: width=NFM; height=NFM; Arc: width=RNAI; Arc-filter: RNAI<30.

The filtering results in a graph that shows 29 nodes and 22 edges representing the source files that are most involved in the coupling between the modules. Out of these files there are three source files that have strong call relationships with 4 or more source files. These files are `nsIContent.h`, `nsRuleNode.cpp`, and `nsIFrame.h`. The other nodes represent source files that also have been highlighted by the system hotspots view before. For instance, `nsCSSFrameConstructor.cpp`, `nsXULDocument.cpp`, or `nsXULElement.cpp`.

Regarding the detailed analysis of the cyclic coupling dependencies detected on the module-level we navigated the graph of involved modules down to their source files and isolated the files that cause this design flaw. In terms of the cyclic coupling between the `NewLayoutEngine` and `XPToolkit` modules we found out that involved functions are contained by five files of the module `NewLayoutEngine`: `nsCSSFrameConstructor.cpp`, `nsListControlFrame.cpp`, `nsTextControlFrame.cpp`, `nsGfxScrollFrame.cpp`, and `nsScrollPortFrame.cpp`.

The analysis of the cyclic coupling by inheritance yielded the source files: `nsTextControlFrame.h`, `nsGfxScrollFrame.h`, and `nsScrollPortFrame.h`. Interesting is that these source files are also involved in the cyclic coupling by function calls. Consequently, clearing the inheritance contributes to the resolution of the cyclic coupling by function calls.

Results For the analysis of the source code coupling between Mozilla's content and layout modules we presented two views on the module level and one view on the file level. The views on the module level highlighted the strong couplings by method invocations and class inheritance. Concerning the first type of source code coupling the four largest modules also are the modules that are coupled most with each other. Several cyclic call dependencies between these modules were detected that needed a detailed analysis. The view on the inheritance hierarchy yielded a clear structure in which the `DOM` module is the super-module from which all other modules inherit behavior.

The detailed analysis of the intermodule coupling was done on the level of source files. The intention was to highlight the source files that contribute most to the coupling between the content and layout modules or are involved in cyclic coupling dependencies. The view on the coupling by method invocations yielded 29 source that are coupled with files of other modules by more than 30 method calls. The modules heavily involved in the coupling are the modules already pointed out at the module level: `DOM`, `XPToolkit`, `NewLayoutEngine`, and `NewHTMLStyleSystem`.

8.3.5 Change Coupling

The views presented in this subsection are used to visualize the change coupling dependencies between software modules and their source files. They complete the source code coupling views by highlighting the implementation units that most frequently were changed together.

Module Views Figure 8.13 depicts the change coupling between the seven content and layout modules. The width and height of nodes indicate the number of modifications that were committed to the CVS repository during the time between Mozilla releases 1.6 and 1.7. Edges represent the aggregated change coupling relationships. The width of edges maps to the accumulated number of pairwise changes (RNMR). To filter out weak relationships we used a threshold of 200 on the RNMR metric.

The graph again highlights the `DOM` module that with an NMR value of 1.496 is the module most frequently changed. The module exhibits strong change coupling relationships to all software modules except the `MathML` module. The strongest change coupling is between the `DOM` and `NewLayoutEngine` modules with 2.635 shared modification reports (RNMR). Another two strong change coupling dependencies occur between the `XPToolkit` and `DOM` (RNMR of 1.783) and `XPToolkit` and `NewLayoutEngine` modules (RNMR of 1.323). Almost all modifications that occur in one of these three modules affected the other modules.

File Views Figure 8.14 depicts the change coupling relationships between source files that cross the module boundaries. We configured the mapping to represent the number of modification reports (NFM) on the width and height of nodes. Edges represent change coupling relationships between source files whereby the size of edges maps to the number of shared modification reports (RNMR).

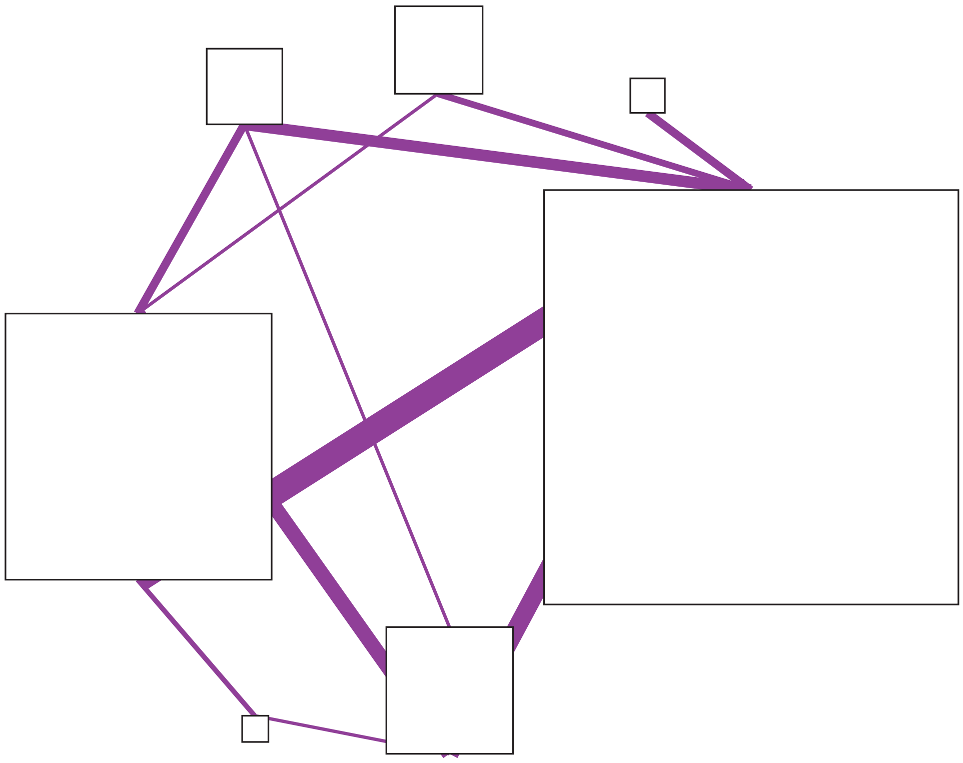


Figure 8.13: Change coupling view on Mozilla 1.7 content and layout modules. Node: width=NMR; height=NMR; Arc: width=RNMR; Arc-filter: RNMR<200.

Using a threshold of 5 we filtered out the weak change coupling relationships and standalone nodes. Remaining files present the candidates that most frequently were changed together. According to the size of nodes the set also includes the files that most frequently were touched during the development of release 1.7. The detailed investigation of the CVS history yielded that these files are among the "top-ten" of all source files concerning the number of committed MRs during the development of the Mozilla release 1.7.

Most of the modifications and most of the pairwise changes affected source files of the three modules `DOM`, `XPToolkit`, and `NewLayoutEngine`. The remaining four modules contain only one source file having high coupling (`XML` and `NewHTMLStyleSystem`) or none (`XSLT` and `MathML`). Related to the module view presented by Figure 8.13 this view further verifies our previous results that the first three modules are most critical for source code modifications – modifying the implementation of one of these modules impacts changes to the other two modules.

Results The views presenting the modification measurements and change coupling relationships derived the software modules and source files that were changed together most frequently. On the module level the view clearly highlighted the three most dependent modules. The view on

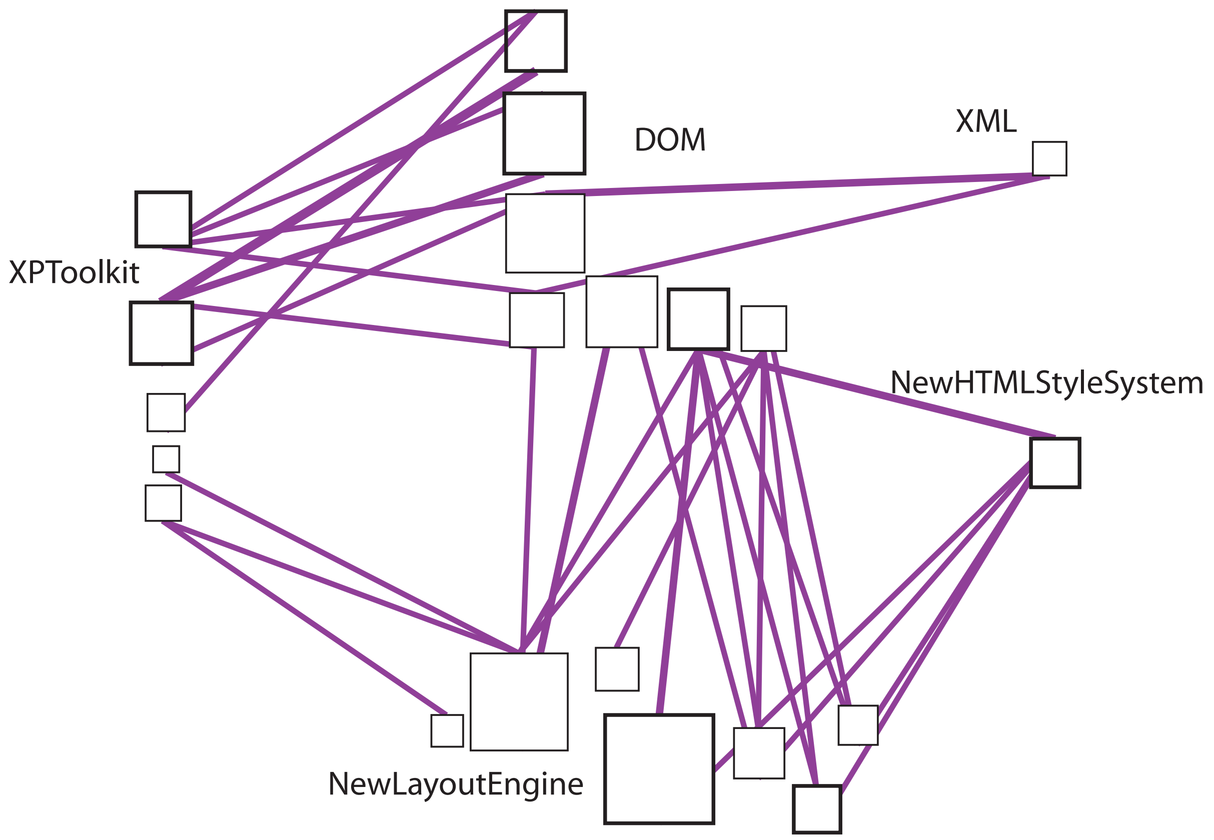


Figure 8.14: Change coupling view on Mozilla 1.7 content and layout source files. Node: width=NMR; height=NMR; Arc: width=RNMR; Arc-filter: RNMR<5.

the file level further strengthened this effect and provided detailed information about the source files most involved in the strong change couplings.

8.4 Evolution from Mozilla Release 0.92 to 1.7

In the previous section we demonstrated that our integrated data model facilitates the generation of different polymetric views on one release of a software system. In this section we go one step further and increase the amount of data to visualize up to n releases.

The primary objective of these views is to sketch the evolution of Mozilla's content and layout modules by including the dimension of time. For this we used the Kiviat graph tool that facilitates the composition of measurements and structures of different Mozilla releases.

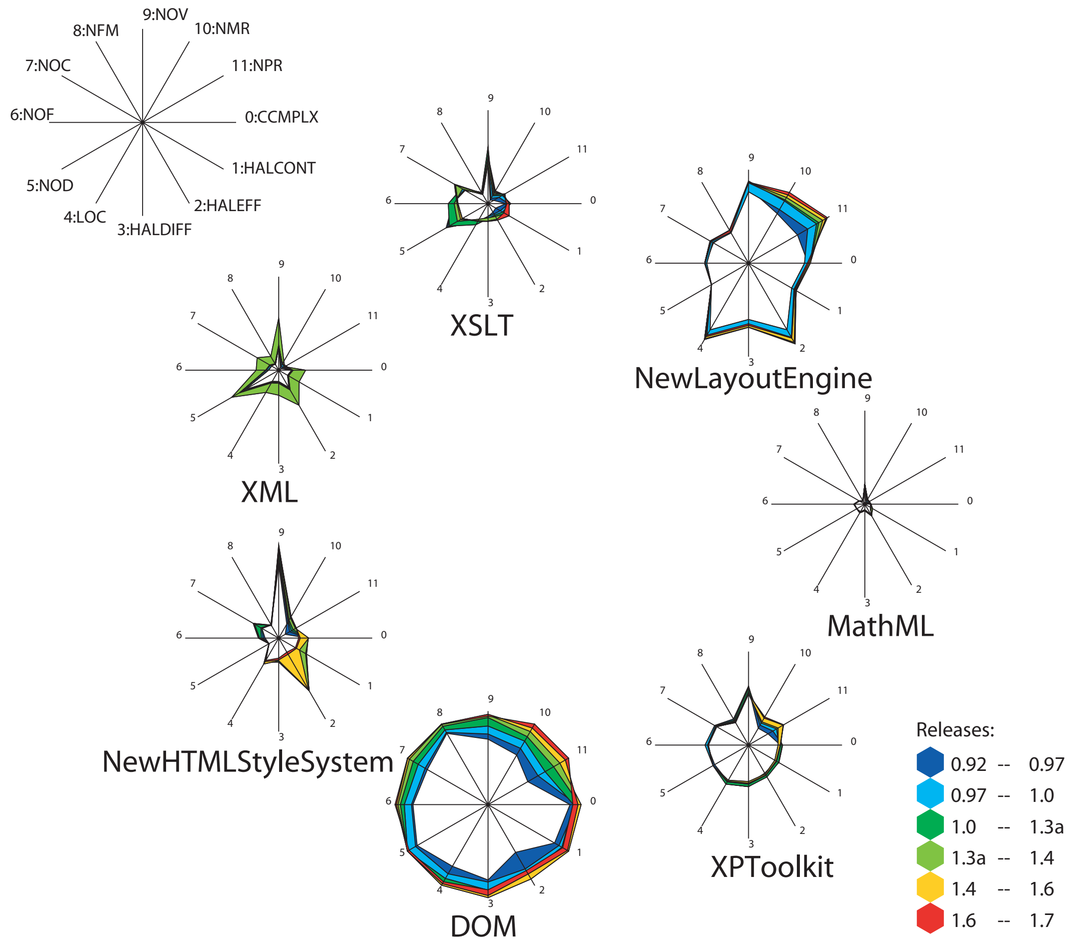


Figure 8.15: Detailed system hotspots evolution view on Mozilla content and layout modules with size and complexity metrics of 7 releases from 0.92 to 1.7.

8.4.1 Evolution of modules

The first set of views concerns the evolution of the seven Mozilla content and layout modules. Figure 8.15 depicts the Kiviat diagrams with the source code size and complexity metrics. Each circle denotes a release starting with release 0.92. The different colors indicate the time periods between two subsequent releases. For instance, the blue polygon spanned by the two inner circles denotes the change in the measured metric values between release 0.92 and release 0.97. The different values depicted by the Kiviat diagrams are normalized by its corresponding maximum.

The graph highlights the `DOM` module as the module that over the seven releases was the largest and most complex module. The color gradient of its Kiviat diagram indicates continuous growth in the different size and complexity metrics.

The Kiviat diagrams highlight outstanding changes in metric values, such as for the `XML` and the `NewHTMLStyleSystem` module. The diagram of the `XML` module indicates a strong

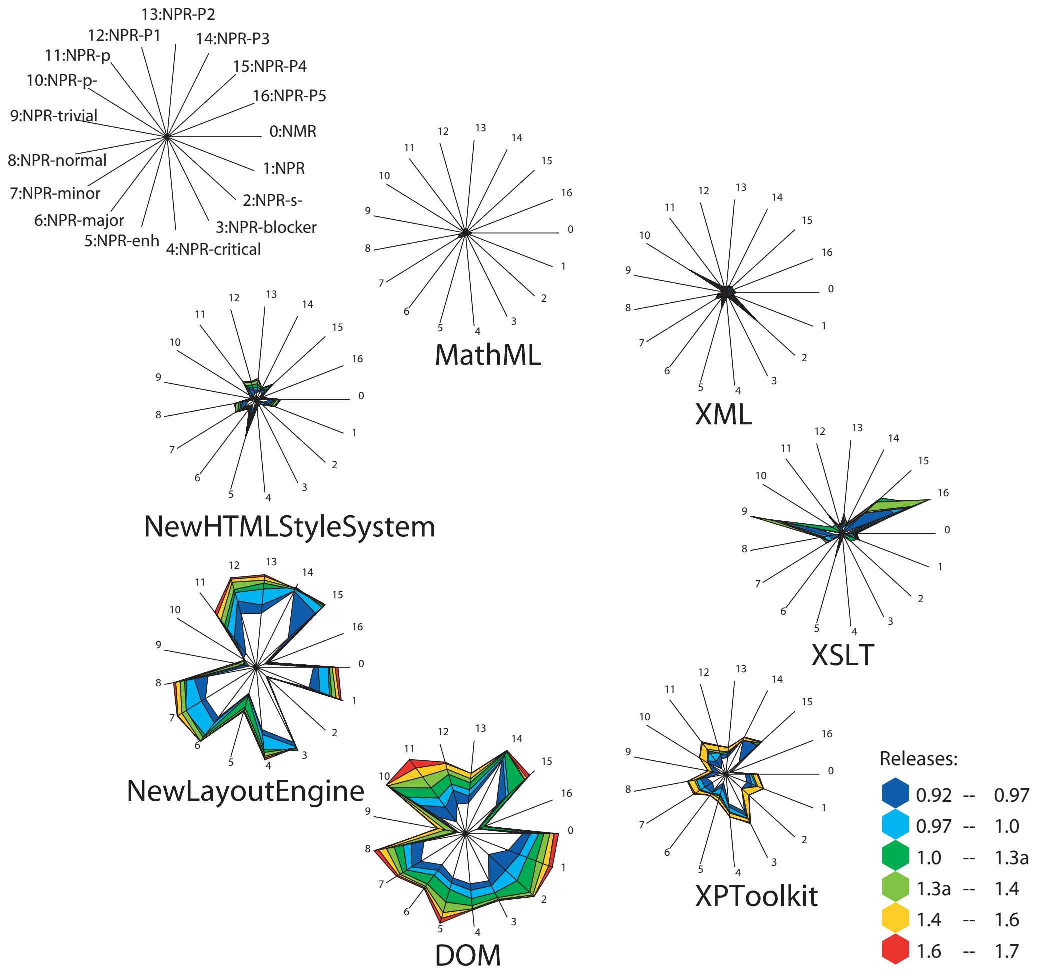


Figure 8.16: Detailed modification hotspots evolution view on Mozilla content and layout modules with problem report metrics of 7 releases from 0.92 to 1.7.

change in almost all values between release 1.3a and 1.4. An inspection of this change revealed that the value decreased because there was a change in the Makefile that reduced the amount of files needed for building the `XML` module. Another change highlighted by our approach happened to the `NewHTMLStyleSystem` module. It affected the HALEFF metric that from release 1.4 to 1.6 decreased from 3.989 to 1.247. This change in complexity is further strengthened by the decreasing values for the HALCONT and CCMPLX metrics. The metric values of the other modules indicate a stable evolution of these modules with only minor changes.

The view depicted by Figure 8.16 sketches the different problem report metrics. The intention of this view is to highlight large increases or decreases in the number of problem reports over the seven releases. Large polygons highlight these strong changes in the Kiviat diagrams.

The diagrams show that the `NewLayoutEngine` and the `DOM` module were assigned the largest number of the critical problem reports. For the `NewLayoutEngine` module most of the problems were reported in the time between release 0.92 and 1.0 as indicated by the blue and cyan polygons. The color gradient of the `DOM` module indicates that `DOM` was most vulnerable to problems in the time from release 0.92 to 1.3a.

Interestingly, the largest number of trivial reports and reports with lowest priority (P5) were reported for the `XSLT` module. From release 0.92 to 0.97 the number of trivial problem reports increased from 56 to 159 (NPR-trivial). The smallest number of problem reports were reported for the `MathML`, `XML`, and `NewHTMLStyleSystem` modules. In relation to the other modules and from the perspective of problem report metrics these modules are stable.

8.4.2 Evolution of critical source files

On the level of source files we applied the Kiviat visualization technique to sketch the evolution of the seven most critical source files. Figure 8.17 shows the Kiviat diagrams with the size and complexity metrics. As on the module level the intention of this view is to point out strong changes of these values over time.

Regarding complexity the `nsCSSFrameConstructor.cpp` was and is the most complex source file. And, this fact did not change during the development of release 1.7. Overall, the Kiviat diagrams highlight only few strong changes. For instance, the decrease of the HALEFF and NOV metrics of `nsGlobalWindow.cpp` between the releases 1.3a and 1.4 or the decrease of the NOC, NFM, and NOV metrics of `nsPresShell.cpp` between the releases 1.0 and 1.3a.

Figure 8.18 depicts the different problem report metrics observed over the seven Mozilla releases.

The file most affected by problems was and is `nsCSSFrameConstructor.cpp`. Its Kiviat diagram shows a continuous increasing number of critical problems (NPR-blocker, NPR-critical, NPR-major) whereby most of them were reported in the earlier releases. The same behavior can be stated for the number of problems with highest priority (NPR-P1, NPR-P2, NPR-P3). The measured metric values of `nsPresShell.cpp` show a similar behavior except that the problems reported for this file were of lower priority. Furthermore, big changes in metric values are depicted by the diagrams of `nsGlobalWindow.cpp` (NPR-s-, NPR-enh) and `nsEventStateManager.cpp` (NPR-P4).

Taking into account the previous view that showed the size and complexity metrics most of the reported problems concerned existing features and not the addition of new features. Relatively small increasing size and complexity metrics but steadily high increasing numbers of reported problems verify this hypothesis. The decline of reported problems in the latter releases indicates that these source files became more stable in the recent releases.

8.4.3 Kiviat graphs

The focus of the views presented in this section is on providing more detailed graphs on the coupling dependencies between software modules. For this we use the Kiviat graph visualization technique that connects the set of Kiviat diagrams by edges which represent the coupling

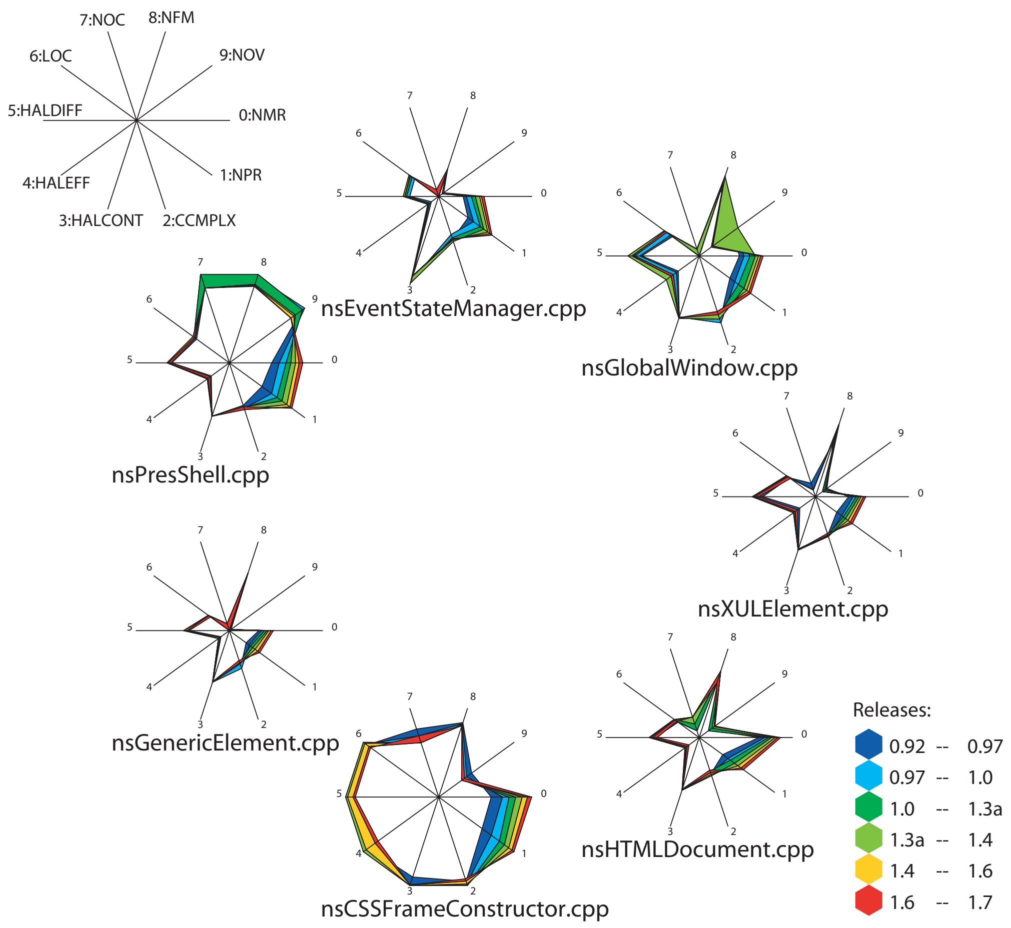


Figure 8.17: Detailed system hotspots evolution view on Mozilla content and layout source files with size and complexity metrics of 7 releases from 0.92 to 1.7.

dependency. For the Kiviat diagrams we select the corresponding fan-in and fan-out metrics to visualize the service provider and requester behavior of each software module.

Figure 8.19 shows the module coupling by function calls. Edges denote aggregated function calls taken from the recent Mozilla release 1.7. We applied a threshold of 50 to filter the weak relationships. The values of fan-in metrics are represented by the attributes 1–4. Fan-out values are represented by the attributes 5–8. The attribute labeled NFM shows the number of functions and methods contained by a module. The measurements are taken from the Mozilla releases 0.92 to 1.7.

The view is an extension to the view shown by Figure 8.10. It highlights the strong coupling relationships and several cyclic dependencies between four modules. Several changes in the values are emphasized by the Kiviat diagrams. For instance, the number of out-going function calls (INR-out) of the `DOM` module steadily increased from release to release until release 1.6. After that release the value decreased by 499 from 1.808 down to 1.309 calls. This denotes a decoupling of the `DOM` module in release 1.7. On the in-coming side of the `DOM` module the

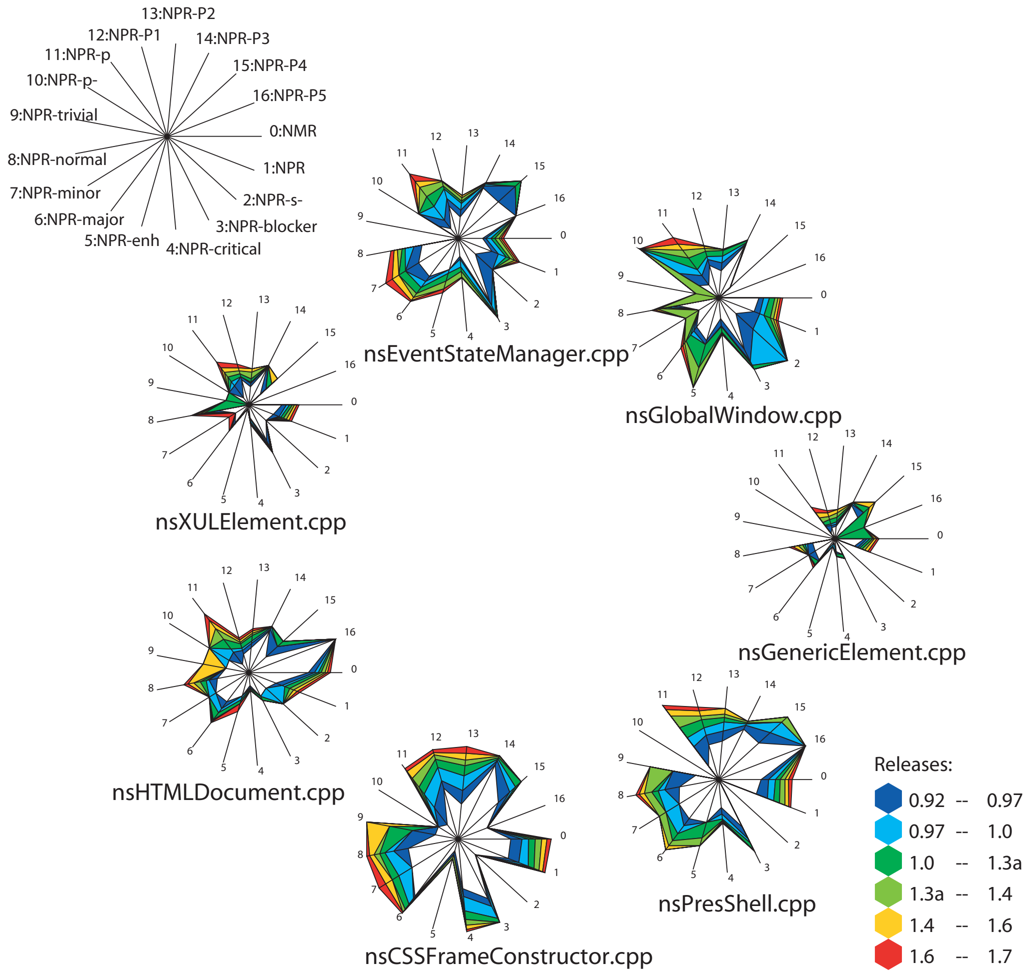


Figure 8.18: Detailed modification hotspots evolution view on Mozilla content and layout source files with problem report metrics of 7 releases from 0.92 to 1.7.

number of function calls between the releases 1.4 and 1.6 increased by 360 up to 1.459 calls.

The Kiviat diagram representing the `NewHTMLStyleSystem` emphasizes changes in the in-coming calls metrics between the two recent releases. The value of the IFan-in metric continuously increased until the release 1.6 but then in release 1.7 decreased by 173 down to 547 functions. The Kiviat diagrams of the remaining nodes show only minor changes except the diagram of `XPToolkit` which shows an up and down for the metrics of out-going function calls.

Figure 8.20 represents the content and layout modules with the aggregated inheritance relationships and corresponding fan-in and fan-out metrics. Regarding the filtering of the weak relationships we applied a threshold of 5.

The graph provides detailed information to the view depicted by Figure 8.11. By grouping the fan-in and fan-out metrics the diagrams categorize the modules into sub- and super-classes

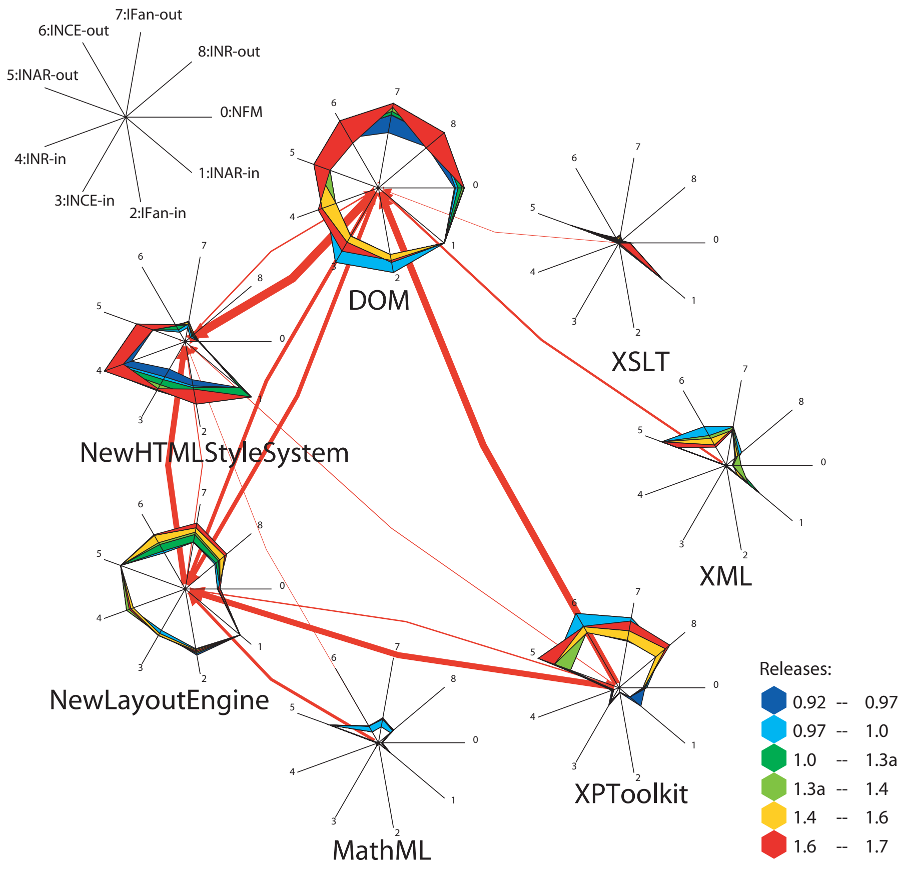


Figure 8.19: Source code coupling evolution view on Mozilla content and layout modules with metric values of in-coming and out-going call relationships. Values are of 7 releases from 0.92 to 1.7. Edges denote aggregated invokes relationships taken from release 1.7 filtered using a threshold of 50 for RNAI.

whereby high fan-out values denote sub-classes and high fan-in super-classes. In addition, the diagrams show the evolution of these metric values. For instance, in the earlier releases the `XPToolkit` module inherited more than 40 super-classes from other modules. Then, from release 1.3a to 1.4 the number decreased to 30 super-classes. The change is highlighted by the big green polygon.

Other large changes are indicated by the diagrams of the `NewHTMLStyleSystem` and `DOM` modules. Both diagrams denote a decrease of inter-module inheritance relationships.

8.4.4 Results

In this section we presented a number of views that were focused with visualizing multiple metrics and their evolution over the seven selected Mozilla releases. Using the Kiviat diagram approach these views highlighted the strong changes in the metric values that were clearly indicated

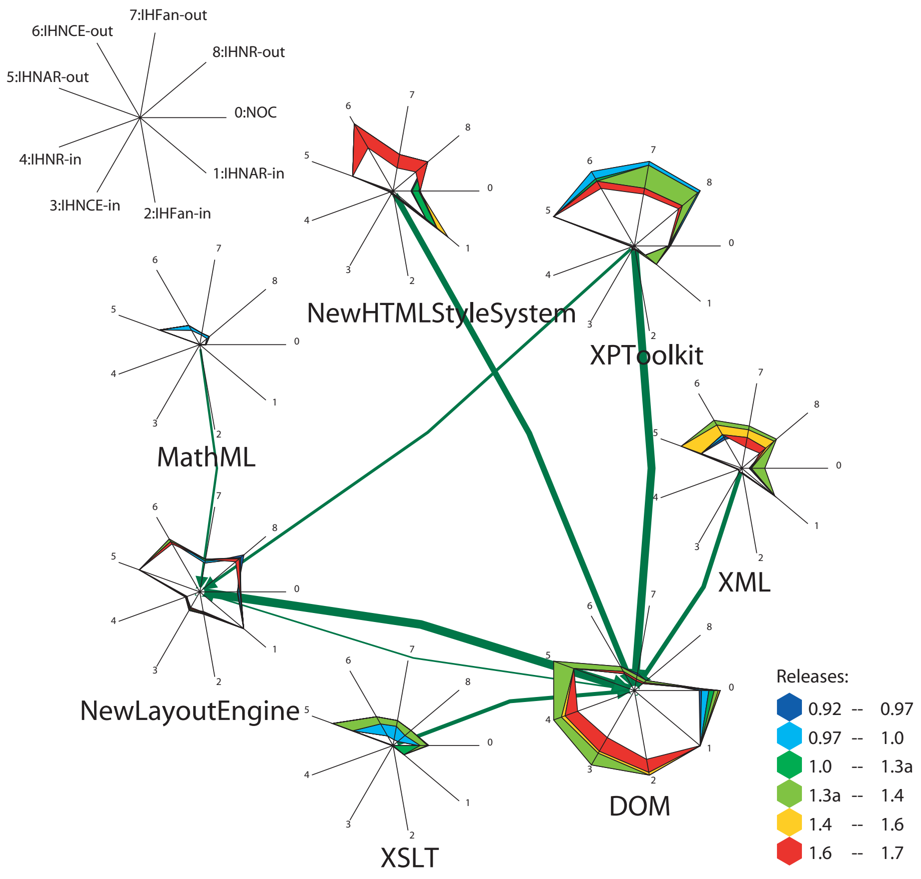


Figure 8.20: Source code coupling evolution view on Mozilla content and layout modules with metric values of in-coming and out-going inheritance relationships. Values are of 7 releases from 0.92 to 1.7. Edges denote aggregated inherits relationships taken from release 1.7 filtered using a threshold of 5 for RNAIH.

by large polygons.

Regarding Mozilla's content and layout modules the diagrams again highlighted the `DOM` module and `nsCSSFrameConstructor.cpp` file as the most critical entities concerning size, complexity, past modifications, and reported problems. But, the trend of the metrics of the two entities also yielded that the most critical phase of the two entities was in the earlier releases. In the recent release they became more stable as indicated by the Kiviat diagrams.

For the detailed analysis of the inter-module coupling relationships we composed Kiviat diagrams to Kiviat graphs. The focus was on visualizing the trend of coupling metrics together with the different types of aggregated relationships that constitute the coupling. We presented two views on the module level, one on intermodule function calls, and another view on intermodule class inheritance relationships. Also in these views the Kiviat diagrams indicated improvements of the most critical modules whose values of coupling metrics decreased in recent releases. This corresponds to the overall impression that we got from the case study: the implementation of

Mozilla's content and layout functionality became more stable in recent releases.

Summarized, our proposal to the Mozilla developer is to split the critical entities highlighted by our views and to resolve the cyclic dependencies. They were and are the central cost factors in terms of numbers of modifications and problems. And, without a directed refactoring they will remain these factors in the up-coming releases.

8.5 Summary of Results

In this section we summarize the case study results and provide answers to the questions concerning the implementation and evolution of the seven Mozilla modules:

- *Which are the main building blocks?*
  In the recent Mozilla release 1.7 seven modules including 1.321 source files implement Mozilla's content and layout handling. The `DOM` module is by far the largest and most complex module. It consists of more than 11.000 functions and methods contained in 492 source files. Another module pointed out as large and complex is `NewLayoutEngine`. It consists of 4.404 functions and methods contained in 229 source files. Compared to these two modules the remaining five modules are small and less complex (see Figure 8.1, Figure 8.2, Figure 8.7).

  On the file level the views highlighted 40 source files with more than 100 functions and high cyclomatic and program complexity. Among these the largest file is `nsCSSFrameConstructor.cpp`. It comprises 13.320 lines of code that implement 208 functions (see Figure 8.3, Figure 8.4, Figure 8.9).

- *Which units are coupled with each other and how strong are these coupling dependencies?*
  In the case study we analyzed the source code and change coupling relationships between modules and their source files. The views on the function calls showed weak intermodule coupling. However, there is the problem of cyclic coupling dependencies that are between the `DOM` `NewLayoutEngine`, and `XPToolkit` modules. A deeper investigation on the file-level pointed us to 29 source files that make up most of the coupling. Concerning the cyclic coupling we determined five source files that are basically responsible for that (see Figure 8.10, Figure 8.12).

  The view on the inheritance relationships depicted a well designed inheritance structure. We found only one weak cyclic coupling between the `NewLayoutEngine` and the `XPToolkit` module that has to be removed. Interestingly, three of the files causing the inheritance cycle also are involved in the cyclic function calls (see Figure 8.11).

- *Are there entities and relationships that indicate Bad Smells?*
  On the module level we identified the `DOM` module as a "God Module". Compared to the other modules it is too large and complex and therefore should be re-factored into smaller sub modules. `NewLayoutEngine` is another module that should be divided into sub

modules. It also is large, complex, and strongly coupled with the other modules. Reducing the size will reduce the complexity and improve maintainability and evolvability of these modules (see Figure 8.3, Figure 8.4, Figure 8.9).

On the file-level we identified 40 source files that implement more than 100 methods. This is far too much behavior that is difficult to understand and maintain. The developers should re-factor the files and move methods to new files (see Figure 8.3, Figure 8.4, Figure 8.9).

With regard to intermodule method invocations and class inheritance we identified a number of cyclic coupling relationships between the content and layout modules. Basically, the modules involved in these design shortcomings are the large and complex modules, namely `DOM`, `NewLayoutEngine`, `XPToolkit`, and `NewHTMLStyleSystem`. The analysis on the file level retrieved the set of source files that cause the cycles. For instance, the cycle between the NewLayoutEngine and XPToolkit modules basically is caused by five source files and a small shortcoming in the inheritance structure (see see Figure 8.10, Figure 8.11, Figure 8.12).

- *How did the implementation units (i.e., modules) and coupling dependencies evolve?*
  Views on the modules indicated a growing content and layout implementation most of all in the large and complex modules `DOM` and `NewLayoutEngine`. However, the increase dropped in the recent releases (see Figure 8.15). The view on the largest source files further strengthen this trend: size and complexity increased in earlier releases but dropped in recent releases (see Figure 8.17).

  The decrease in size and complexity is also visible in the diagrams showing the intermodule coupling dependencies. They highlight strong coupling in the earlier Mozilla releases that have been reduced in recent releases (see Figure 8.19, Figure 8.20). Concerning the intermodule coupling this clearly denotes improvements.

- *Which modules were most vulnerable to problems and modified most frequently?*
  Here the module views showed a clear picture: the large and complex modules were most affected by high-priority problems and therefore were assigned the highest number of modification reports. Normalizing the metric values by the number of source files (NOF) we obtain that the `NewLayoutEngine` module by far is the most change prone module. On average 44 problem reports were assigned to a file of this module that is more than twice as much as for the other modules. For instance, `DOM`, `XPToolkit`, and `NewLayoutEngine` have a problem report rate of 20 reports per source file (see Figure 8.8). The progression of these metrics indicated a stabilization regarding the number of new problems and modifications (see Figure 8.16).

  On the file level we identified the change prone source files. The resulting graph is dominated by the `nsCSSFrameConstructor.cpp` file that was and still is the most change prone file followed by `nsPresShell.cpp`, and `nsGlobalWindow.cpp`. But also for these files the progression of evolution metric values depict a reduction in the number of new problems and modifications (see Figure 8.18).

- *Are there change couplings between modules and how strong are they?*
  We visualized the change coupling relationships between the software modules that occurred from Mozilla release 1.6 to 1.7. The graph shows that except one module (`MathML`) all modules have change couplings with the `DOM` module. Consequently, changes to the `DOM` module lead to changes in the other modules. Other two modules that are involved in the change coupling are `XPToolkit` and `NewLayoutEngine` (see Figure 8.13).

  Details about the change coupling are provided by the file-level view. They showed that the heavy change coupling relationships are between files of the three modules pointed out on the module level. The source files that were most involved in this kind of coupling between the releases 1.6 and 1.7 are those files that are also involved in the intermodule coupling via method invocations. They truly propagated modifications across modules to other source files. To find out the reasons for this we have to investigate the problem and modification reports in more detail than the file level.

8.6 Discussion of Results

The Mozilla case study demonstrated that our integrated data model and visualization techniques facilitate the identification of the implementation units and the coupling dependencies that are critical for maintaining and evolving the content and layout modules. The integration of the release history allowed for the computation of evolution metrics that addressed the frequency of modifications and problems as well as the change couplings.

Based on this data model we applied the polymetric views visualization technique. Resulting views showed that this technique is useful to point out the entities and relationships of interest by mapping metric values to graphical attributes of one release. We demonstrated that our extension of polymetric views to Kiviat graphs facilitated the visualization of multiple metrics of up to n releases. Kiviat diagrams and graphs provided additional clues of the evolution of implementation units and in particular highlighted the strong changes which were of primary interest. They indicate improvements but also degradation in the implementation.

During the case study with the Mozilla open source software project we encountered a number of problems that affected our results. These problems were concerned with extracting the facts, integrating the data models, and creating the views.

8.6.1 Fact extraction

In the fact extraction the primary issues were related with properly configuring the C/C++ parser to obtain a fact base of reasonable quality. For the Mozilla project we had to manually inspect the Makefiles of the different software modules to obtain the compiler settings. The effect of these settings emerged in certain metric measurements and consequently graphs. For instance, the Kiviat diagram of the XML module depicted in Figure 8.15 shows such a strong change that basically results from a change in the Makefile between the releases 1.3a and 1.4. In the latter the directory `schema` and `soap` were not included by default as was done in release 1.3a. The result is a decrease in size and complexity of the XML module.

The quality of the parser influences the results. The version 5.0.2 of the Imagix-4D parser performs a full semantic analysis but fails when parsing certain C/C++ template constructs. Because of this deficiency the parser missed a considerable amount of function calls that occur in the Mozilla source code. Basically, missed function calls involved functions of classes that are wrapped by a `XPCom` component. XPCom is the component framework of Mozilla that heavily uses C/C++ templates which caused the parsing problems.

Listing 8.1 depicts an example of a class (interface) nsIScrollable that is implemented with the XPCom framework.

Listing 8.1: C/C++ template example of a function call missed by the Imagix-4D parser.

```
// nsIScrollable.h
class NS_NO_VTABLE nsIScrollable : public nsISupports {
public:
  NS_IMETHOD GetDefPrefs(PRInt32 scrollOrientation,
                                        PRInt32 *scrollbarPref) = 0;
  ...
}

// nsBarProps.h
NS_IMETHODIMP ScrollbarsPropImpl::GetVisible(PRBool *aVisible) {
  nsCOMPtr<nsIScrollable> scroller(do_QueryInterface(docshell));

  // Missing/Mismatched function declarator
  scroller->GetDefPrefs(nsIScrollable::ScrollOrientation_Y, &prefValue);
  ...
}
```

The class `nsIScrollable` contains a method `GetDefPrefs(..)` that is called by the GetVisible(..) method of another class named `ScrollbarsPropImpl`. The reference to this call is missed by the Imagix-4D parser because it can not statically determine the type of the class to which the method `GetDefPrefs` belongs to. This effected the quality of our fact base and the analysis results of the coupling by function calls. The other calls were fully retrieved hence there was sufficient function call information available for our analysis. However, having also the set of missed calls would further improve our analysis results. It is part of our on-going and future work to develop a workaround that uses the GNU gcc parser to obtain the missed calls. The initial approach and first results are presented in [GPG04].

The other data extraction issues concerned the CVS and Bugzilla data, in particular the linkage of modification and problem reports. The links were reconstructed by matching the bug numbers in the messages entered by the Mozilla developers when committing their changes to the CVS repository. Because these numbers were entered as free text there is a number of false positives and negatives that have to be considered. Summarizing the results presented in [FPG03a] we retrieved a high number of correctly established links and a low number of false positives and negatives. However, we have to thank the Mozilla developers for entering the report numbers. Without this information the linkage of modification and problem report data needs a serious workaround.

8.6.2 Data model integration

In the current version of our approach we consider integration of data models on the level of source files and higher. As mentioned in the integration step of the Mozilla case study we used the short file name for determining the links between the file entities of the source code and release history data models. The percentage of established links was 96% in the earlier and 99.8% in the latter Mozilla releases. This resulted in a well integrated data model that was useful to perform analysis on the level of source files and software modules.

But, to perform a more detailed analysis of modifications and problems on the level of classes and methods the model integration algorithm has to be refined. Whereas, for linking source files the file name is sufficient, it is not for classes and moreover for methods. Typically, in a large source code base there exists several methods with the same name and even the same signature that lead to false positive matches. Further, methods are more likely to be changed or moved around. Determining a link between such methods is not trivial and subject of on-going and future research.

8.6.3 Visualization

The Polymetric Views technique and in particular the hotspots views turned out to be useful to get an overview of the system by highlighting its core elements. In addition to the views of Lanza *et al.* in [LD03] we incorporated additional metrics that addressed the source code and coupling between entities as well as metrics computed of the release history data. Including these metrics we were able to compute views that highlight the entities that were the critical cost factors in terms of number of modifications and problems.

By extending the polymetric views technique to use Kiviat diagrams instead of rectangles our approach facilitates the visualization of multiple metrics in each node. In the Mozilla case study the diagrams facilitated us to put more details about certain aspects, such as size and complexity or usage relationships between software modules, into one view. They also enabled us to analyze the relation between certain metrics, such as between the size and complexity and the number of modifications of a module in one and multiple releases.

In this context, a lesson learned from the case study was that the user has to make sure:

- To select the proper set of metrics to be visualized (*e.g.*, avoid too many metrics); and
- To properly order metrics in the Kiviat diagrams (*e.g.*, group metrics that belong together).

The views we presented in the case study considered these aspects and represent an initial set of views from which the user can derive other view configurations. However, to claim a view configuration to be useful we need user-studies that are still an open issue.

Using the Kiviat diagrams to present multiple metrics observed over several subsequent Mozilla releases proofed to be suitable to highlight strong changes in metric values. We identified such changes in several software modules and source files of Mozilla. During the case study we encountered also a number of limitations of the Kiviat diagram approach but also potential solutions to handle them. These are:

1. *Views become cluttered with information when visualizing complex graphs or a large number of metrics*: Information cluttering can be handled by reducing the set of entities to visualize or by applying filters.

2. *Polygons representing the metric values of recent releases overlap polygons of earlier releases*: This problem emerges especially when visualizing measured values of multiple metrics of a large number of releases. A workaround is to select only the releases with the large changes in metric values and redraw the graph again.

3. *Normalization to the maximum obfuscates small values*: To gain insights into the entities with little changes we remove the large and complex entities from the list of selected entities. This lowers the different maxima and emphasizes small metric values.

In summary, the ArchView approach provided us with several techniques to analyze the aspects that we mentioned in the Mozilla scenario. The quality of the integrated data model as well as the abstraction and visualization techniques provided us with the views that we needed for the analysis. The interpretation of the views then is subject to user.

CHAPTER 9

CONCLUSION

In this chapter we summarize the contributions of this dissertation, discuss the benefits of our approach, and indicate directions of future work.

9.1 CONTRIBUTIONS

In this dissertation we tackled the issues concerned with creating and presenting higher-level views on the implementation of a software system and its evolution. Summarized, these issues comprised:

- *The building of an integrated data model.* The analysis of evolutionary aspects needs the consideration of different data sources that provide information about the implementation, problems, and modifications of a software system. We presented the techniques and tools to process these data sources and extract the corresponding data models. For the integration of the different data models we introduced the E-FAMIX meta model that serves for subsequent abstraction, visualization, and analysis tasks. The E-FAMIX meta model facilitates the navigation of entities and relationships across source code releases, from source code to release history data, and vice versa.

- *The Abstraction of lower-level data.* The sheer amount and complexity of information obtained from several source code releases, versions, and bug reporting systems of large software systems blurs views. We built on an existing aggregation technique to condense implementation and evolution specific information to higher-level views. In this context we presented the containment hierarchy model that specifies the entities and paths along which lower-level entities and relationships are abstracted to the level of modules. The abstraction was accompanied by the computation of size, complexity, modification, and problem report metrics. The latter two metrics represented important extensions to existing source code metrics. Basically, they were used to highlight the change prone modules and heavy change coupling relationships.

- *The creation of coarse and fine-grained graphical views*. For the creation of these views we built upon the existing polymetric views technique and extended it towards the visualization of multiple metrics of up to n releases. We realized these extensions by using Kiviat diagrams instead of trivial graph glyphs. We described the measurement mapping technique and showed the composition of Kiviat diagrams to Kiviat graphs. Based on the integrated, abstracted, and enriched data model we provided a set of new hotspots views used to answer questions concerning structural and evolutionary aspects of the implementation. In particular, we were able to:

    – Identify the main modules of a software system their size and complexity.

    – Show the inter-module coupling relationships and the type and strength of these relationships.

    – Identify the modules and coupling dependencies indicating design shortcomings (Bad Smells, such as God Modules, cyclic dependency, and dead code).

    – Identify the modules that have been most vulnerable to problems (*i.e.,* change prone modules) and, on the other side, the stable modules.

    – Show the heavy change coupling relationships indicating frequent propagation of changes between modules (*i.e.,* Shotgun Surgery).

    – Show the growth in size and complexity of modules across releases and identify periods with improvements or degradations.

    – Show the progression of problem vulnerability and modification frequency and identify modules that are or tend to become vulnerable and modules that are or tend to become stable.

The different techniques and algorithms have been integrated into the ArchView approach. In order to demonstrate the ArchView approach we applied it to the open source project Mozilla that provided us with a representative amount and complexity of source code and release history data. The focus was on analyzing the implementation of Mozilla's content and layout functionality, and in particular its structural and evolutionary aspects.

The results of the case study comprised a number of views on the level of software modules and source files identifying the large, complex, buggy, and frequently modified modules and source files as well as the heavy source code and change coupling relationships. In addition, we presented views depicting metric trends indicating improvements and degradations in the design and implementation of Mozilla's content and layout modules. Questions, such as when has the number of problem reports of module A increased rigorous, could be answered. In this way the ArchView views identified the change prone content and layout modules and source files to which most of the maintenance effort has been dedicated to. They represent the candidates for a refactoring and this was what we claimed to provide in the case study.

Regarding the case study we also presented a discussion of the results and described possible pitfalls and shortcomings of the ArchView approach. A number of these pitfalls and shortcomings are subject to future work described in the next section.

9.2 Future Work

In the following we list open issues in the area of architecture recovery and software evolution analysis that showed up while developing the ArchView approach. Basically, they concern data extraction, model integration, view visualization, and view analysis.

- *Dynamic analysis*: A possible extension to ArchView is by the inclusion of run-time data that can be obtained from execution traces. We then can apply ArchView for analyzing the structural and evolutionary aspects of features, components, and connectors. Initial approaches that address these issues have been presented by Fischer *et al.* [FG04, FOGG05]. Future work is concerned with integrating these techniques into ArchView.

- *Extraction and evolution of patterns*: Similar to feature data obtained from dynamic analysis we plan to integrate data from extracted patterns and analyze how they influenced the evolution of software systems. Patterns range from code patterns, over object-oriented design pattern to architectural patterns and styles. Concerning code patterns we plan to integrate the Revealer approach [PFGJ02] into ArchView to do pattern-supported architecture recovery and analysis [PG02].

- *Fine-grained analysis of modifications*: Another possible direction of future work is in the detailed analysis of modifications, such as which classes, methods, fields, or even more detailed which statements where inserted, added, or changed. This enables the reconstruction of modifications and a detailed analysis of them. For instance, based on the detailed information a categorization of changes into “good” changes that led to improvements and “bad” changes that led to “Bode Smells” is possible. Based on these categorization of modifications “Evolutionary Smells” can be identified. Along with the detailed information about modifications another open issue is concerned with the detailed analysis of reported problems. Questions, such as which problems where trivial to solve and which caused changes in the design and architecture of a system could be answered. Both issues give deeper insights into the evolvability of software systems.

- *Inclusion of bug-activity data*: In the current version of ArchView we do not take into account the bug activity log data as tracked by the Bugzilla bug reporting system. Basically, this data describes the work that is going on to resolve a reported problems. It keeps track of the bug fixing process starting with reporting the problem, assigning it to a developer, and usually ending with verifying the resolution. With this data it is possible to navigate the history of each report and perform more detailed analysis of the problem resolution process. Preferable, this open issue has to be combined with the fine-grained modification analysis to develop an early-warning-system that, integrated into an IDE, warns the developer when a class or method exceeds its allowed fault and change proneness threshold.

- *Data extraction from framework-based software systems*: In our previous work [POG03] and [KP03] we showed that the extraction of framework specific facts is mandatory to analyze framework-based software systems. The problem is that program logic is hidden away in extra configuration files or even in comments. Future work is needed to develop extraction tools that obtain and integrate the framework specific data within source code model and release history data. This allows for the detailed analysis of such software systems. Because of the diversity of existing frameworks we propose to start with the major frameworks, such as Sun's J2EE or Microsoft's .NET.

- *Data mining*: Taking into account the fine-grained modification and bug activity data the fact repository contains a richness of data to analyze. Future work can be spent on applying sophisticated data mining algorithms to this data with focus on the source code and evolution metrics. Issues, such as what is the relation between metrics, which metrics have to be computed for the characterization of the evolvability of software systems, are there metrics that are irrelevant, or prediction of problems and modifications have to be addressed.

- *View visualization*: The amount and type of data provided by the ArchView repository demands improvements of visualization techniques. In particular, future work has to be done in the presentation and navigation of views. There are a number of ideas future research can play with, such as using animations or 3D graphics to visualize the evolution of software systems. Animations, for instance, show the series of snapshots taken from parts or the whole system animating the change in size, complexity, problem and modification frequency. In addition, further support for navigating the huge amount of complex data has to be investigated and tested.

APPENDIX A

THE EXTENDED FAMIX META MODEL

This chapter provides the background information about the meta model that is used by ArchView to store extracted and abstracted source code and configuration management data.

Recent and ongoing work in the field of meta models is concerned with integrating existing source code meta models into a common meta model that can be extended towards programming language and application specific requirements.

The FAMIX model [Sof99] from the University of Bern follows this ideas. FAMIX is a meta model for a language-independent representation of object-oriented source code. It supports extension points to tailor the meta model to include programming language specific features such as C++ templates.

A.1 E-FAMIX META MODEL

ArchView uses the E-FAMIX meta model which is an extension to FAMIX source code meta model. The FAMIX model [Sof99] was developed by the University of Bern and provides a meta model for a language-independent representation of object-oriented source code. It supports extension points to tailor the meta model to include programming language specific features such as C++ templates. We use these extensions points to include:

- Entity and relationship types that are related with representing modification and problem report data as obtained from the CVS and Bugzilla systems. Concerning relationships we add the "couples" relationship type that denotes a change coupling.

- Metrics that are computed for abstracted entities and relationships. Entity metrics denote the size, complexity, modification report, problem report, and coupling metrics (see Table 6.1, Table 6.2, Table 6.3, Table 6.4, and Table 6.5). Relationship metrics denote the weight of an abstracted source code or change coupling relationship (see Table 6.6).

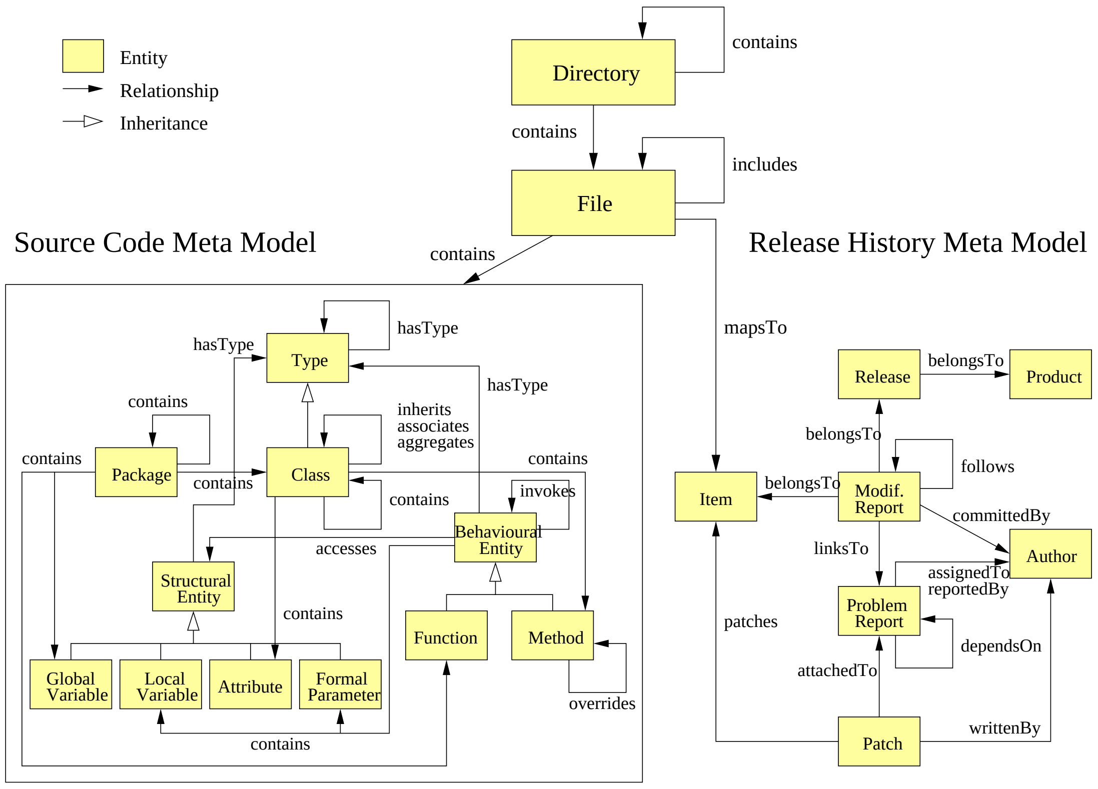


Figure A.1: E-FAMIX meta model - based on the FAMIX source code meta model, extended by the release history meta model.

Figure A.1 shows the the core of the E-FAMIX meta model. The meta model consists of two major models - the source code model and the release history model. Entities common to both models are directories and files. In the source code model files contain the source code (implementation) specific entities such as packages, classes, methods, and attributes. Files can include other files (*e.g.*, C/C++ header files) and are contained in a directory. Directories may be nested and often base directories contain the implementation of a specific software module. With respect to the release history model files are items that are managed by configuration management systems. Modifications are made to files and checked in into the source code repository managed by such systems. Therefore, files are first class entities for establishing links between the source code model and release history model.

The E-FAMIX meta model described above can be extended towards the inclusion of additional entity and relationship types that are mandatory for specific analysis tasks. In context of this thesis the current version of E-FAMIX is sufficient.

A.2 SOURCE CODE MODEL

The ArchView source code model specifies the source code related entity and relationship types needed to represent source code of object-oriented and procedural like programming languages. Figure A.1 depicts the core of the E-FAMIX source code model. For a detailed description interested readers are referred to the FAMIX documentation [Sof99]. The following is a brief description of the model:

- Entities:
  - Package: A Package is a named source code unit used to group source code entities that logically belong together. Packages are programming language dependent, for example, Java uses packages whereas C++ uses namespaces.
  - Type: The Type entity represents data types as used by typed based programming languages. For example, Java supports primitive data types (*e.g.,* boolean, int, long) and complex data types (*i.e.,* classes). Regarding C++ the Type entity also represents type aliasing by typedef statements.
    - Class: Class is derived from the Type entity to represent classes as used by object-oriented programming languages. Different types of classes such as C/C++ structs, unions or Java interfaces are not supported per se but can be distinguished by setting corresponding attribute values.
  - Behavioral Entity: This entity type comprises source code entities that implement behavior. Based on the scope of the entity E-FAMIX considers:
    - Function: Functions are behavioral source code entities of global scope. Typically, they are used by procedural/functional programming languages such as C.
    - Method: Methods are similar to functions but denote behavioral entities of a class. Object-oriented programming languages such as Java or C++ use the concepts of methods whereas in C++ methods are called member functions.
  - Structural Entity: Structural entities are source code entities that concern the representation of the state of a system. They denote locations in memory where information about the current state is stored. Depending on the scope we distinguish between:
    - Global Variable: Structural entities of global scope are denoted global variables. The global scope implies that these variables are globally accessible and valid during the lifetime of the running system.
    - Local Variable: In contrast to global variables local variables are defined locally within a behavioral entity (*i.e.,* function, method). Access to local variables is limited to entities within the scope of the function or method.

        - Attribute: Attributes are used in object-oriented programming languages to define structural entities within the scope of a class or instance of a class (*i.e.,* object).
        - Formal Parameter: Formal parameters are similar to local variables in that their scope also is limited to functions or methods in which they are declared. The difference to local variables is that they specify arguments that are passed to a behavioral entity.

- Relationships:
    - contains: Relationships of this type define the containment of entities by other entities. Based on these relationships a containment-hierarchy is established that defines the path along which low-level elements are abstracted (see Section 6).
    - hasType: These relationships are used to define the data type of structural entities and the return type of behavioral entities. Further, type aliasing (*e.g.,* C++ typedef) is expressed by these relationships.
    - includes: The inclusion of files is a functionality that is typical for the C/C++ programming language. An includes relationship denotes that a source file includes another source file to, for instance use the class declaration of the included file.
    - inherits: The inherits relationships reflects the inheritance between classes. Inheritance is a basic concept used by object-oriented programming languages to inherit responsibilities from base classes. The application of this concept in source code is interesting on the design and architectural level.
    - associates, aggregates: These two relationship types have been taken from the Unified Modeling Language (UML) to denote relationships between classes in the design level. However, implementation of these relationships vary in source code and often can not be distinguished by the parser.
    - invokes: Invocations denote calls between behavioral entities such as function are methods calls.
    - overrides: Overriding is a concept used by object-oriented programming languages to override functionality inherited from a base class. Extracted relationships provide information about the use of inherited and added functionality.
    - accesses: Accesses to structural entities are denoted by relationships of this type. Currently, ArchView does not distinguish between set and reading of structural entities.

A.3 Release History Model

The ArchView release history model specifies entity and relationship types that are related to modification specific data such as version, modification and problem report data as obtained from configuration management systems. Integration of release history data into source code model data is beneficial in that it adds information about changes to source code entities. Consequently, ArchView uses this information to analyze the impact of changes to certain source code entities to other entities. The core of the release history model is depicted in Figure A.1. It consists of:

- Entities:
  - Product: Products are the results of software engineering efforts. Usually, the output of software projects comprises a number of products that are delivered to customers.
  - Release: According to Jacobson *et al.* [JBR99] a release is a relatively complete and consistent set of artifacts that is delivered to internal or external users. During the lifetime of software products several releases are developed. Basically, each development cycle concludes with a product release that is ready for delivery. Regarding version management systems a release is a snapshot on the source code repository whereas source files are tagged with the release number.
  - Item: Item denotes entities that are subject to store in version management systems. Basically, items are files such as source and configuration files or design documents but may be of any type that can be handled by version management systems. ArchView focuses on implementation units and therefore concentrates on source and configuration files that implement a particular release of a software product.
  - Problem Report: Problem reports are created by users that discover bugs or are responsible for reporting bugs in the system. The evidence of a bug report leads to modifications (*e.g.,* bug fixes) in the implementation of a software system.
  - Modification Report: A modification report describes a particular modification to an item that has been committed to the repository. Changes to the implementation of a software system may affect n files. When committing these changes n modification reports are generated (one per affected file). From the point of view of the modification (*e.g.,* bug fix, add feature, refactoring) such commits bring about change related couplings between affected items (*i.e.,* source files) that are utilized by the ArchView architecture recovery approach.
  - Author: Author denotes software developers that report, comment or fix bugs, or commit modifications to source code repositories managed by version systems.
  - Patch: Patches denote bug fixes. Typically, patches are provided in form of source code and contain information about the source files the patch affects and the lines of code be added or deleted. Depending on the bug reporting system patches often are attached to bug reports.
- Relationships:
  - belongsTo: Similar to contains these relationships denote containment. They are used to organize entities in hierarchies.

- follows: Each commit of modifications to a particular item results in an increment of the revision number assigned to each item by the version management systems. In addition to the revision numbers these relationships describe the sequence of modifications to a particular item.
- linksTo: Bug reports typically lead to bug fixes which lead to modifications in source files. Relationships of this type reflect this change process and denote the bug reports that led to associated modification reports.
- dependsOn: These relationships denote dependencies between bugs, for example, a new bug occurred from a fix of a previous bug.
- reportedBy, assignedTo, writtenBy, committedBy: These relationships indicate users reporting bugs and users who are responsible for fixing them. Further, they denote the user who implemented a patch to fix a bug and the user who commits fixes to the source code repository.
- attachedTo: Often patches are attached to bug reports. These relationships describe this attachment and consequently indicate the patch that fixes a given bug.
- patches: These relationships denote the application of which patches have been applied to which source files.

For each entity and relationship type there are a number of attributes that result from extraction and abstraction processes. During extraction of the source code model details and metrics about each entity are extracted that are stored in the entity's attributes. For example the file location of entities, the signature of methods, access control qualifiers of methods and attributes, etc. In the release history model attributes include the date of modification or bug reports, the size of modifications (*e.g.,* lines added/deleted), severity, priority, and status of bug reports, etc.

APPENDIX B

PUBLICATIONS

This chapter presents the list of publications on which this dissertation is based on.

B.1 VISUALIZATION & ANALYSIS

Visualizing Multiple Evolution Metrics

Abstract: Observing the evolution of very large software systems needs the analysis of large complex data models and visualization of condensed views on the system. For visualization software metrics have been used to compute such condensed views. However, current techniques concentrate on visualizing data of one particular release providing only insufficient support for visualizing data of several releases. In this paper we present the RelVis visualization approach that concentrates on providing integrated condensed graphical views on source code and release history data of up to n releases. Measures of metrics of source code entities and relationships are composed in Kiviat diagrams as annual rings. Diagrams highlight the good and bad times of an entity and facilitate the identification of entities and relationships with critical trends. They represent potential refactoring candidates that should be addressed first before further evolving the system. The paper provides needed background information and evaluation of the approach with a large open source software project.

Published: In Proceedings of the ACM Symposium on Software Visualization, pages 67–75, St. Louis, Missouri, 2005. ACM Press.

Towards an Integrated View on Architecture and its Evolution

Abstract: Information about the evolution of a software architecture can be found in the source basis of a project and in the release history data such as modification and problem reports. Existing approaches deal with these two data sources separately and do not exploit the integration of their analyses. In this paper, we present an architecture analysis approach that provides an integration of both kinds of evolution data. The analysis applies fact extraction and generates specific directed attributed graphs; nodes represent source code entities and edges represent relationships such as accesses, includes, inherits, invokes, and coupling between certain architectural elements. The integration of data is then performed on a meta-model level to enable the generation of architectural views using binary relational algebra. These integrated architectural views show intended and unintended couplings between architectural elements, hence pointing software engineers to locations in the system that may be critical for on-going and future maintenance activities. We demonstrate our analysis approach using a large open source software system.

Published: In Electronic Notes in Theoretical Computer Science, 127(3):183–196, April 2005.

B.2 Architectural View Abstraction

Abstracting Module Views from Source Code

Abstract: In this paper we present ArchView an approach for abstracting and visualizing software module views from source code. ArchView computes abstraction metrics that are used to filter out architectural elements and relationships of minor interest resulting in more reasonable and comprehensible module views on software architectures.

Published: In Proceedings of the International Conference on Software Maintenance, page 533, Chicago, USA, 2004. IEEE Computer Society Press.

B.3 Model Extraction & Integration

Towards the Integration of Versioning Systems, Bug Reports and Source Code Metamodels

Abstract: Concurrent Versioning System (CVS) repositories and bug tracking systems are valuable sources of information to study the evolution of large open source software systems. However, being conceived for specific purposes, *i.e.,* to support the development or trigger maintenance activities, they do neither allow an easy information browsing nor support the study of software evolution. For example, queries such as locating and browsing the faultiest methods are not provided.

This paper addresses such issues and proposes an approach and a framework to consistently merge information extracted from source code, CVS repositories and bug reports. Our information representation exploits the property concepts of the FAMIX information exchange metamodel, allowing to represent, browse, and query, at different level of abstractions, the concept of interest. This allows the user to navigate back and forth from CVS modification reports to bug reports and to source code. This paper presents the analysis framework and approaches to populate it, tools developed and under development for it, as well as lessons learned while analyzing several releases of Mozilla.

Published: In Electronic Notes in Theoretical Computer Science, 127(3):87–99, April 2005.

Analyzing and Understanding Architectural Characteristics of COM+ Components

Abstract: Understanding architectural characteristics of software components constituting distributed systems is crucial for maintaining and evolving them. One component framework heavily used for developing component-based software systems is Microsoft's COM+. In this paper we particularly concentrate on the analysis of COM+ components and introduce an iterative and interactive approach that combines component inspection techniques with source code analysis to obtain a complete abstract model of each COM+ component. The model describes important architectural characteristics such as transactions, security, and persistency, as well as create and use dependencies between components, and maps these higher-level concepts down to their implementation in source files. Based on the model, engineers can browse the software system's COM+ components and navigate from the list of architectural characteristics to the corresponding source code statements. We also discuss the Island Hopper application with which our approach has been validated.

Published: In Proceedings of the International Workshop on Program Comprehension, pages 54–63, Portland, Oregon, 2003. IEEE Computer Society Press.

BIBLIOGRAPHY

[14700] IEEE Std 1471-2000. Ieee recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, 2000.

[ABF04] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, NY, June 1990. ACM Press.

[APGP05] Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, and Martin Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, 127(3):87–99, April 2005.

[Arn96] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[BDW99a] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January 1999.

[BDW99b] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance*, pages 475–482, Oxford, England, UK, 1999. IEEE Computer Society Press.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[Bel99] Bell Canada Inc. *DATRIX - Abstract semantic graph reference manual*, 1.2 edition, July 1999.

[Ber74] Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[BM99] Elizabeth Burd and Malcolm Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering*, pages 168–174, Atlanta, Georgia, 1999. IEEE Computer Society Press.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[CBB$^+$02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CFV99] Aniello Cimitile, Anna Rita Fasolino, and Giuseppe Visaggio. A software model for impact analysis: a validation experiment. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 212–222, Atlanta, GA, 1999. IEEE Computer Society Press.

[CM03] Davor Cubranić and Gail C. Murph. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Portland, Oregon, 2003. IEEE Computer Society Press.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999.

[CMW02] Katja Cremer, André; Marburger, and Bernhard Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance*, 14(4):257–292, 2002.

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

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[EKRW02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. Gupro - generic understanding of programs. *Electronic Notes in Theoretical Computer Science*, 72(2):59–68, 2002.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FG04] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:385–403, 2004.

[FH00] Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance*, pages 88–96, San Jose, CA, October 2000. IEEE Computer Society Press.

[FHK$^+$97] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[FKvO98] Loe Feijs, Rene Krikhaar, and Rob van Ommering. A relational approach to support software architecture analysis. *Journal of Software Practice and Experience*, 28(4):371–400, 1998.

[FO00] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8), 2000.

[FOGG05] Michael Fischer, Johann Oberleitner, Harald Gall, and Thomas Gschwind. System evolution tracking through execution trace analysis. In *Proceedings of the International Workshop on Program Comprehension*, pages 237–246, St. Louis, Missouri, 2005. IEEE Computer Society Press.

[FP96] Norman E. Fenton and Shari Lawrence Pfleeger, editors. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, 2nd edition, 1996.

[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*, pages 90–99, Victoria, B.C., Canada, November 2003. IEEE Computer Society Press.

[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*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[Fre03] Free Software Foundation. *Version Management with CVS*, 1.11.14 edition, 2003. http://www.cvshome.org/docs/manual.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[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 the International Conference on Software Maintenance*, pages 40–49, Chicago, Illinois, 2004. IEEE Computer Society Press.

[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*, pages 190–198, Bethesda, Maryland, USA, 1998. IEEE Computer Society Press.

[GHJ04] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings the 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.

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

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 13–23, Helsinki, Finland, 2003. IEEE Computer Society Press.

[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*, pages 160–166, Bari, Italy, 1997. IEEE Computer Society Press.

[GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, UK, 1999. IEEE Computer Society Press.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(18):751–761, 1991.

[GL00] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of the Second International. Symposium on Constructing Software Engineering Tools*, Limerick, Ireland, June 2000.

[GM03] Nicolas Gold and Andrew Mohan. A framework for understanding conceptual changes in evolving source code. In *Proceedings of the International Conference on Software Maintenance*, pages 432–439, Amsterdam, The Netherlands, 2003. IEEE Computer Society Press.

[GPG04] Thomas Gschwind, Martin Pinzger, and Harald Gall. Tuanalyzer—analyzing templates in c++ code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering*, pages 48–57, Delft, Netherlands, November 2004. IEEE Computer Society Press.

[GSV02] David Grosser, Houari A. Sahraoui, and Petko Valtchev. Predicting software stability using case-based reasoning. In *Proceedings of the 17th International Conference on Automated Software Engienering*, pages 295–298, Edinburgh, Scotland, UK, September 2002. IEEE Computer Society Press.

[Hal77] Maurice H. Halstead. Elements of software science, operating, and programming systems series. *Elsevier*, 7, 1977.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[Hol98] Richard C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering*, pages 210–219, Honolulu, Hawai, 1998. IEEE Computer Society Press.

[HS95] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1995.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. Gxl: Toward a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 162–171, Brisbane, Australia, November 2000. IEEE Computer Society Press.

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

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JRvdL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[KC99] Rick Kazman and S. Jeromy Carriére. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

[KP03] Jens Knodel and Martin Pinzger. Improving fact extraction of framework-based software systems. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 186–195, Victoria, B.C., Canada, November 2003. IEEE Computer Society Press.

[Kri99] Rene Leo Krikhaar. *Software Architecture Reconstruction*. PhD thesis, Universiteit van Amsterdam, 1999.

[Kru95] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KS03] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 36–45, Victoria, Canada, 2003. IEEE Computer Society Press.

[KWC98] Rick Kazman, Steven G. Woods, and S. Jeromy Carrire. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of the 5th Working Conference on Reverse Engineering*, pages 154–163, Honolulu, Hawai, 1998. IEEE Computer Society Press.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 37–42, Vienna, Austria, September 2001. ACM Press.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, 2003.

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

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

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

[LPR$^+$97] Manny 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 *Proceedings of the 4th International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, USA, 1997. IEEE Computer Society Press.

[LPR98] Manny 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*, pages 208–217, Bethesda, Maryland, USA, 1998. IEEE Computer Society Press.

[LR03] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Portland, Oregon, 2003. IEEE Computer Society Press.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[MK88] Hausi A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, April 1988. IEEE Computer Society Press.

[MMCG99] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, Oxford, England, September 1999. IEEE Computer Society Press.

[MMF03] Jonathan I. Maletic, Andrian Marcus, and Louis Feng. Source viewer 3d (sv3d): a framework for software visualization. In *Proceedings of the 25th International Conference on Software Engineering*, pages 812–813, Portland, Oregon, 2003. IEEE Computer Society Press.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287, Sorrento, Italy, 1994. IEEE Computer Society Press.

[PFG05] Martin Pinzger, Michael Fischer, and Harald Gall. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196, April 2005.

[PFGJ02] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pages 170–178, Richmond, Virginia, October 2002. IEEE Computer Society Press.

[PFJG04] Martin Pinzger, Michael Fischer, Mehdi Jazayeri, and Harald Gall. Abstracting module views from source code. In *Proceedings of the International Conference on Software Maintenance*, pages 533–533, Chicago, USA, 2004. IEEE Computer Society Press.

[PG02] Martin Pinzger and Harald Gall. Pattern-supported architecture recovery. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.

[PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75, St. Louis, Missouri, 2005. ACM Press.

[PGG$^+$03] Martin Pinzger, Harald Gall, Jean-Francois Girard, Jens Knodel, Claudio Riva, Wim Pasman, Chris Broerse, and Jan Gerben Wijnstra. Architecture recovery for product families. In *Proceedings of the 5th International Workshop on Product Family Engineering*, Lecture Notes in Computer Science, pages 332–351, Siena, Italy, 2003. Springer-Verlag.

[POG03] Martin Pinzger, Johann Oberleitner, and Harald Gall. Analyzing and understanding architectural characteristics of com+ components. In *Proceedings of the International Workshop on Program Comprehension*, pages 54–63, Portland, Oregon, 2003. IEEE Computer Society Press.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Riv04] Claudio Riva. *View-Based Software Architecture Reconstruction*. PhD thesis, Vienna University of Technology, 2004.

[Sab01] Michael Saboe. The use of software quality metrics in the materiel release process experience report. In *Proceedings of the 2nd Asia-Pacific Conference on Quality Software*, pages 104–109, Hong Kong, 2001. IEEE Computer Society Press.

[SBLE00] Houari A. Sahraoui, Mounir Boukadoum, Hakim Lounis, and Frédéric Ethève. Predicting class libraries interface evolution: an investigation into machine learning approaches. In *Proceedings of 7th Asia-Pacific Software Engineering Conference*, pages 456–464, Singapore, December 2000. IEEE Computer Society Press.

[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 Systems and Software*, 44(3):171–185, 1999.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 1995 International Conference on Software Maintenance*, pages 275–284, Opio, France, October 1995. IEEE Computer Society Press.

[Sof99] Software Composition Group, University of Berne. *The FAMIX 2.0 specification*, 2.0 edition, August 1999. http://www.iam.unibe.ch/ scg/Archive/famoos/FAMIX/.

[Som00] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition edition, 2000.

[Tuf90] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[Tuf97] Edward R. Tufte. *Visual Explanations*. Graphics Press, 1997.

[Uni05] Uni Stuttgart. *Bauhaus: Software Architecture, Software Reengineering, and Program Understanding*, May 2005.

[VRD04] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 328–337, Chicago, Illinois, USA, September 2004. IEEE Computer Society Press.

[War00] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[Won98] Kenny Wong. *The Rigi User's Manual — Version 5.4.4*. University of Victoria, 5.4.4 edition, 1998.

[WSHH04] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution spectrographs: Visualizing punctuated change in software evolution. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 57–66, Kyoto, Japan, September 2004. IEEE Computer Society Press.

[YCM78] Stephen S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Computer Society Press, 1978.

[YMNCC04] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[ZDZ03] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83, Helsinki, Finland, 2003. IEEE Computer Society Press.

[ZWDZ04] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Edinburgh, Scotland, 2004. IEEE Computer Society Press.

Curriculum Vitae

Personal Information

Name:	Martin Pinzger
Nationality:	Austria
Date of Birth:	September 8th, 1974
Place of Birth:	Zams, Tirol, Austria

Education

2001 - 2005	Ph.D. in Computer Science in the Distributed Systems Group at the Vienna University of Technology Subject of the Ph.D. thesis: “ArchView - Analyzing Evolutionary Aspects of Complex Software Systems”
1996 - 2001	Student in Computer Science at the Vienna University of Technology Master in Computer Science in the Distributed Systems Group at the Vienna University of Technology Subject of the Master thesis: “Re-engineering von Flugplanungssoftware” at the EADS Dornier GmbH in Friedrichshafen, Germany
1996 - 1996	Military service in the Austrian Federal Armed Forces
1993 - 1995	Kolleg for EDV und Organisation at HBLV für Textilindustrie in Vienna Graduation on September 26th, 1995
1989 - 1993	Scientific Bundes- Oberstufenrealgymnasium in Landeck Graduation on June 13th , 1993
1980 - 1989	Primary Schools in Pfunds