Examining Software Architecture Evolution using Change-sets

by

Andrew McNair
B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Examining Software Architecture Evolution using Change-sets

by

Andrew McNair

Supervisory Committee

Dr. Jens H. Weber-Jahnke, Supervisor (Department of Computer Science)

Dr. Daniel M. German, Supervisor (Department of Computer Science)

Dr. Hausi A. Müller, Department Member (Department of Computer Science)

Dr. Kin Fun Li, External Examiner (Department of Electrical and Computer Engineering)

Supervisory Committee:

Dr. Jens H. Weber-Jahnke, Supervisor (Department of Computer Science)

Dr. Daniel M. German, Supervisor (Department of Computer Science)

Dr. Hausi A. Müller, Department Member (Department of Computer Science)

Dr. Kin Fun Li, External Examiner (Department of Electrical and Computer Engineering)

ABSTRACT

A significant challenge in understanding the evolution of a software system is coping with the huge amounts of data left behind during the evolution. One strategy for summarizing this data is to visualize its effect on the system's architecture. Existing tools that implement this strategy often provide mechanisms to filter the data under consideration. However, this filtering is generally limited to showing the evolution over some unbroken sequence of time, for example the changes over the last six months.

In this work we present an alternative approach designed to provide a method for examining the net effect of any set of changes on a systems architecture. We also present Motive, a prototype tool that implements this approach, and demonstrate how it can be used to answer questions about software evolution by describing case studies we conducted on two Java systems.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank my supervisors, Dr. Jens Weber-Jahnke and Dr. Daniel German, for both their wise council and incredible patience.

I would like to express my gratitude to all the members of the PPCI research group for their assistance. In particular, Glen McCallum, Adeniyi Onabajo, and Paul Crawford went to great lengths to help me.

I would like to acknowledge the immense amount of support I have received from friends and family during my time at UVic. I will forever be in your debt.

Finally, I would like to thank Anita Thambirajah who was always there to encourage me.

Dedication

To my parents.

Chapter 1

Introduction

1.1 Motivation

Most software systems exist to support solving problems in real world domains. Over time, the problems these systems are designed to solve will change as the real world context of the software changes. The result, widely known as Lehman's Law of Continuing Change [28], is that the system must adapt or grow less useful. The process by which software adapts over time is termed *software evolution*.

There are three main groups interested in studying software evolution: developers, researchers, and managers. Developers want to understand how the current state of a software system has come to be in order to better maintain the system; researchers want to learn about how systems in general evolve by studying examples of how specific projects evolved; finally, managers want to monitor the progress being made by their development team towards current development goals, and to use information about past progress to help plan future development work.

The study of software evolution is possible because of the existence of what German refers to as *software trails* [16], information left behind by the contributors to the development process of the product. Examples of software trails are mailing lists, web sites, version control logs, software releases, documentation, and source code. A major open re-

search topic is, “how can information from software trails be made accessible so that users can find answers to their questions about software evolution?”

An important concern in making information accessible is managing the amount of data presented to a user. The massive volumes of data present in software trails makes this concern particularly relevant to a user interested in studying software evolution. Our work is motivated by the question, “how can the information stored in software trails be filtered so that users can focus on what is relevant to their current task?”

1.2 The Problem

Although any artifact generated during the development process may provide insight about the evolution of the system, information about how the source code of the system has evolved stands out as the most important software trail. This information is often accessible due to the widespread development practice of using software configuration management (SCM) systems, which store information about the source files of the software system and how these files have changed over time. When a user commits changes to one or more source files as part of the same logical transaction, SCM systems generally record both what files were modified, and some metadata about the transaction, such as the author who made the change, when the change was made, and a text description of the purpose of the change. Software evolution researchers refer to these logical transactions as modification requests (MRs) [18].

Our approach is centered around finding ways to filter the data stored in an SCM system according to the current needs of a user. As software evolution is a process that happens over time, a reasonable starting point at filtering the evolution is to consider how the system changed between two points of time. This *time-based* filtering provides support for answering questions such as, “what changed in the last six months of development?”, “what changed in the system between these two releases?”, and “what changed in the system between these two MRs?”

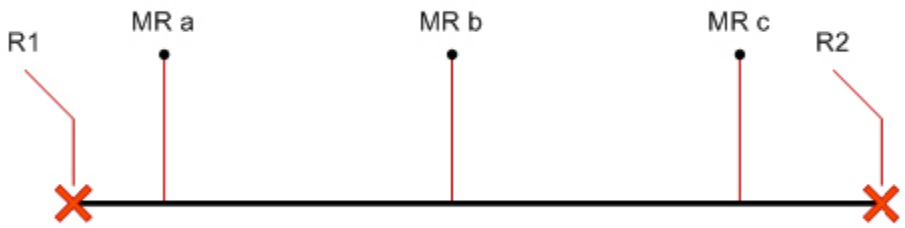


Figure 1.1. *A timeline of an example software system between two releases. The labels R1 and R2 indicate when releases 1 and 2 occurred. Between the two releases MRs a, b, and c occurred.*

However, it is our belief that time is just one of many attributes which can be used to filter evolution. Other methods of filtering may be useful in order to isolate a particular logically related thread of evolution. In order to show these different threads, which may not represent unbroken periods of time, our approach filters according to a *change-set*, a subset of the MRs that occurred to the system.

To illustrate the concept of a change-set, we introduce a very simple example. Figure 1.1 shows a timeline of changes to a sample system. Three MRs (a, b and c) have occurred to the system between its two releases, $R1$ and $R2$.

There are six possible change-sets in this example that time-based filtering approaches could examine:

1. (a). The effect of MR a
2. (b). The effect of MR b
3. (c). The effect of MR c
4. (a,b). The change between a and b (inclusive).
5. (b, c). The change between b and c (inclusive).
6. (a,b,c). The change between a and c (inclusive).

Time-based filtering approaches, however, would not be able to show (a,c), the effect of MRs a and c together, ignoring the effect of MR b. There are many reasons a user might be

interested in this change-set. For example, MRs a and c might have been made by the same author, and the user is interested in examining that author's work. MRs a and c might have been made in response to the requests of a particular stakeholder, and the user is interested in examining the impact of that stakeholder. MRs a and c might each modify the same software feature, and the user is interested in examining the effort required to maintain that feature. In general, a change-set could be constructed based on any logical relationship among the MRs within the change-set.

There are a number of ways in which the impact of a change-set might be understood. In this thesis we focus on visualization of the net-effect of the change-set. Visualization is a common technique when trying to summarize large amounts of data, as is the case when trying to understand the impact of a large change-set. There has been a lot of prior work done in using visualization to show the evolution of a system's architecture over time, and we build upon that work in order to show the evolution of a system's architecture over a change-set.

1.3 Outline

This thesis is laid out as follows. This chapter provided a brief description of the difficulty in analyzing the huge amount of data stored in software trails, and why we believe our change-set approach may be a useful technique in filtering this data. Chapter 2 discusses background material related to the study of software evolution, in particular examining existing approaches to visualizing software evolution. Chapter 3 describes our model and method for computing and visualizing the impact of change-sets. Chapter 4 describes a prototype tool we have developed, Motive, that implements our model and method. Chapter 5 describes an evaluation we conducted to determine whether our approach had merit. Chapter 6 provides a conclusion to the thesis, summarizes our contributions and identifies future work to be done in this area.

Chapter 2

Background

This chapter begins with a brief overview of the history and motivations of software evolution research, in particular work done on visualizing software evolution. A categorization of tools for viewing software evolution is presented, and examples of tools in each category are described. This chapter concludes with a summary of what makes the change-set approach novel, and why it may be useful.

2.1 Software Evolution

It is generally considered that the first work to systematically study software evolution was the study of OS/360 conducted by Lehman (originally printed in a confidential 1969 IBM report, reprinted in a 1985 publication [26]). Together with Belady, Lehman's empirical studies of OS/360 and other software systems led to the identification of what today are termed the "Laws of Software Evolution". The motivation of Lehman's work is increasing the high-level understanding of how software in general evolves, and refining a "theory of software evolution" [27]. Lehman believes that a better understanding of software evolution would provide insight into how to build and maintain systems so that they remain useful for as long as possible.

Another motivation for the study of software evolution is to help programmers, and their

managers, in performing software maintenance. The concepts of software maintenance and software evolution are closely related. Tu points out that software evolution is the process by which a software system changes over time, whereas software maintenance is an attempt to control this change process [41]. An understanding of how a software system has changed in the past can improve the planning and carrying out of future maintenance activities. One example of this type of work is Hipikat [7], which uses information about previous software development activity to suggest software development artifacts that may be pertinent to the current needs of a user.

Regardless of the motivation, the study of software evolution is based around examining software trails. There are many data mining techniques that can be used to help extract relevant information from software trails. A comprehensive survey of work done in this field is given by Kagdi, Collard, and Maletic [24].

2.1.1 Software Evolution Visualization

Once the information of interest has been extracted from the software trails, a common method for presenting it to the user is visualization. The varied motivations for studying software evolution have resulted in a diverse set of visualization approaches. To gain an overview of these approaches it is necessary to adopt some sort of classification scheme that groups similar approaches.

One approach for classification is the framework of software visualization tools that provide awareness of human activities in software development presented by Storey, Čubranić, and German [36]. They classify tools according to five dimensions:

Intent. The general purpose and motivation that led to the design of the visualization.

Information. The data sources that a tool uses to extract relevant awareness information.

Presentation. How the tool or proposed tool presents the extracted and derived information to the various user roles.

Interaction The interactivity and liveness of the tool.

Effectiveness The feasibility of the proposed approach, whether it has been evaluated and whether it has been deployed.

An alternative set of dimensions for classifying software visualization tools was developed by Maletic, Marcus and Collard [29]. They consider:

Tasks *why* is the visualization needed?

Audience *who* will use the visualization?

Target *what* is the data source to represent?

Representation *how* to represent the data source?

Medium *where* to represent the visualization?

Our approach is based around viewing the effect of a change-set on the software architecture, which most closely maps to the *target* dimension. Therefore, to show how our work fits into the field of software evolution we present tools in terms of the target they are designed to visualize. From this classification method we have identified four broad categories of tools:

1. Artifact-centric

   The tool is designed to provide a view of how some artifacts stored in the SCM system's repository change over time, especially the source code files and the lines of code within the files. Artifact-centric tools do not need to do any preprocessing of the data in a repository, though for performance purposes data might be extracted to a database before visualization.

2. Architecture-centric

   The tool is designed to provide a view of how the architecture of the software has changed over time by showing changes to the entities and relationships that make up the architecture. Architecture-centric tools need to do some parsing of the source code in the repository in order to recover an understanding of the architecture.

3. Metric-centric

   The tool is designed to provide a view of how some software metrics have changed over time. Metric-centric tools need to do some preprocessing of repository data in order to recover metrics.

4. Feature-centric

   The tool is designed to provide a view of how features of the software have changed over time. Feature-centric tools need to perform preprocessing of repository data. As well, in order to link source code to features, there must be an analysis of at least one other data source, such as comments in the logfiles or data from an issue tracking repository.

2.2 Artifact-centric Visualization

Arguably the most widely used visualization of software evolution is that used by diff [12], and closely related tools. These tools are focused on showing the change between two versions of a file in terms of lines added, removed, and modified. This visualization is very useful to developers trying to understand particular changes to files in source control repositories; however, diff does not provide good support for developing a high-level understanding of how the software system has evolved over time.

Seesoft [9] [1] is a tool designed to show, in one view, a summary of how a large number of files have changed. Files are viewed as rectangles, where the size of the rectangle reflects the size of the file. Lines are displayed as pixels, with the color of the pixel indicating certain metrics about the line. Figure 2.1 shows a view where the color of a pixel represents the age of the corresponding line. The figure also shows how a user can view greater detail about the changes to a particular section of a file. The Seesoft approach allows the user to, in one view, gain an overview of how up to 50,000 lines of code have evolved. Other researchers have recognized the scalability of this approach and built upon it, for example the Augur system [11] which is designed to be an open framework for

Figure 2.1. *Seesoft [1]. Files are displayed as rectangles and the lines of the file are displayed as pixels, with the color of the pixel indicating metrics about the line. In this example, the color of the line represents its age.*

exploring the relationship between software artifacts and developer activities.

CVSScan [42] is an artifact-centric evolution visualization tool that integrates a number of different views. The file-based and line-based views together show how the lines of a file have been modified over time, when files went through periods of great change, when they became stabilized, and what areas of the file needed a lot of modification. Metric views display metrics about each version of a file, such as number of lines or the author. As well, a text view allows users to zoom in and see the evolution of selected code fragments.

Figure 2.2 shows these multiple views working together. The main view is a line-based view of how lines have changed over time. The x coordinates correspond to versions of the

Figure 2.2. *CVSScan [42]. CVSScan integrates a line-based view, two metric views, and two text views (indicated in the diagram as the code view).*

file, and the y coordinates to global line position. So, each row shows when a line was introduced to the file and what modifications the line has gone through. There are two metric views. The horizontal view can display metrics about the file version, such as its size or its author, and the vertical view metrics about the line of code.

Other artifact-centric approaches are focused more on showing how files have evolved over time. In [34], Van Rysselberghe and Demeyer show a visualization designed to summarize in one view how all the files in the system have changed. They describe a 2D graph with files on the x-axis, time on the y-axis, and dots to indicate when a file was involved in a change. Figure 2.3 shows this approach applied to the open source Apache Tomcat project.

Although this visualization is very simple, some interesting patterns can be detected. For example, unstable files can be identified by long vertical lines that show a lot of changes, and related entities can be identified by looking for entities with similar change patterns.

A related visualization is the Evolutionary Spectrograph approach developed by Wu, Holt, and Hassan [44] to show how a spectrum of components have changed over time.

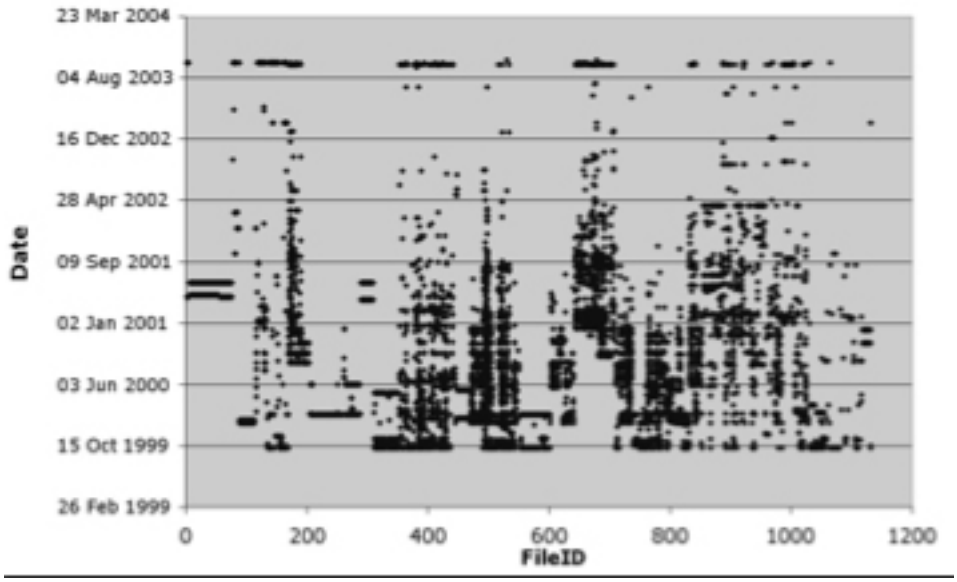


Figure 2.3. *Van Rysselberghe and Demeyer's visualization [34] applied to Tomcat. Date is shown on the y-axis and files on the x-axis. From the diagram patterns about the evolution of files can be detected.*

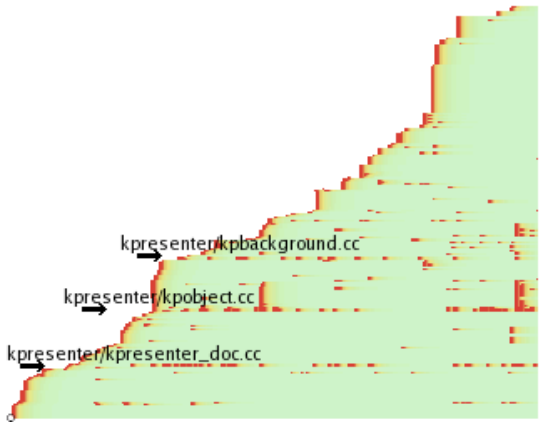


Figure 2.4. *An evolutionary spectrograph showing changes to files in KOffice over 200 commits [44]. Files are laid out on the y-axis and commits to the software system on the x-axis. Red indicates versions in which files were changed, with the color of the file fading to green over successive versions as it is not changed.*

This is done using a graph, with software units (files or subsystems) laid out on the y-axis, and versions on the x-axis. The color of the software unit at each version is defined by the user so as to indicate some aspect of evolution. For example, in Figure 2.4 units are colored red in versions in which they were changed, then over successive versions in which the unit is not changed its color fades to green. One disadvantage with this view is that it is hard to locate patterns between units if the tool does not place the units closely to each other. CVSGrab [43] uses a similar visualization but attempts to alleviate the difficulty with recognizing patterns by allowing the reordering of units based on different user-defined criteria.

Revision Towers [39] is a visualization geared towards languages which have separate implementation and header files, such as C and C++. A tower, as shown in Figure 2.5, represents a view of how the implementation and header file have changed over a set of software releases. Varying thickness and height is used to show how the files have changed in size, when a file was updated, and how many times a particular file has been modified during a release. Color can be used to indicate the author that modified a file. Animation

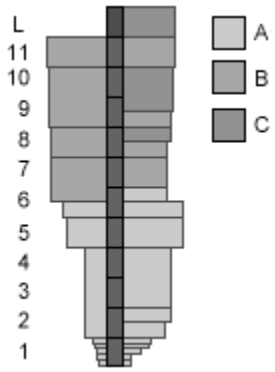


Figure 2.5. *A revision tower [39] showing the evolution of a header file (the left side of the central line) compared to its corresponding implementation file (on the right side of the line). The y-axis indicates the software release, with the earliest release at the base of the tower. The width of the file indicates its size. The color of the file indicates the author that modified it.*

of these towers shows the evolution of the software system over time.

Evolutionary Storyboards [2] is an approach focused on animation as a technique for showing the dynamic process of evolution. The tool shows the evolution of a graph by using panels to represent the change the graph has undergone in a particular time period. Groups of panels can be combined into an animation to show the evolution over several consecutive time periods. This method is programming language independent and can also be extended to work with other artifacts, such as documentation. We classify this approach as artifact-centric because the specific graph Beyer and Hassan choose to describe is the co-change graph, a representation of the changes to the software files over time.

Color can be added to this graph in two ways. Figure 2.6 shows nodes assigned colors based on their subsystem decomposition. For example, the Query Evaluation Engine is colored yellow. Alternatively, nodes can be colored according to how much they have moved over time, with files that have moved more being involved in more changes with other files.

Xia [45], a visualization tool designed as an Eclipse plugin built on top of SHriMP

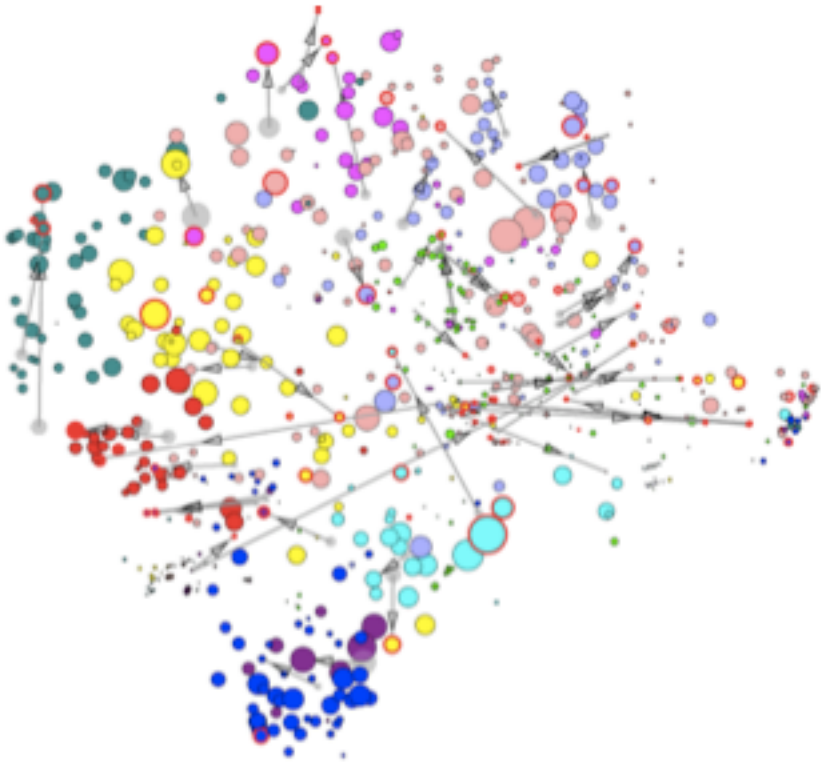


Figure 2.6. *An evolutionary storyboard panel for PostgreSQL [2]. Nodes represent files, with the position of the nodes indicating what files were changed together. The arrows show the change in node position from one panel to another. The color of the node represents the subsystem the node belongs to.*

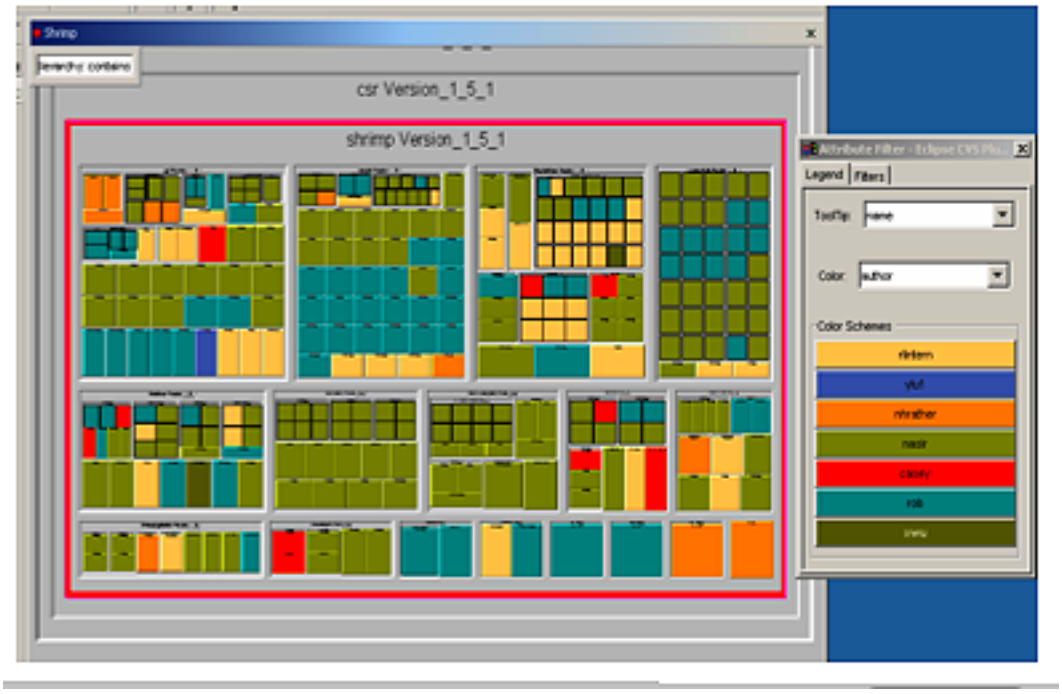


Figure 2.7. *Xia visualizing SHriMP. The colored nodes represent files, with the outer surrounding nodes representing the position of the file in the directory structure. Color indicates what author made the last change to a file.*

[37], demonstrates the power of allowing the user to query and filter data. The main view provided is a nested graph, with the nodes being the files and directories. Color is used to show different nominal attributes, such as the author who committed the last change, or the type of change last performed. Intensity of color is used to show ordinal attributes, such as the date of the last committment. Figure 2.7 shows an example of using color to indicate which author last modified a node.

Xia provides two dynamic filtering mechanisms, as shown in Figure 2.8. Each value of a nominal attribute has a checkbox to indicate whether nodes with this attribute should be displayed or hidden. For example, the user can choose to show only nodes whose last change was made by a particular author. Double sliders are used to specify the range of values ordinal values a user is interested in. For example, the user can choose to show only

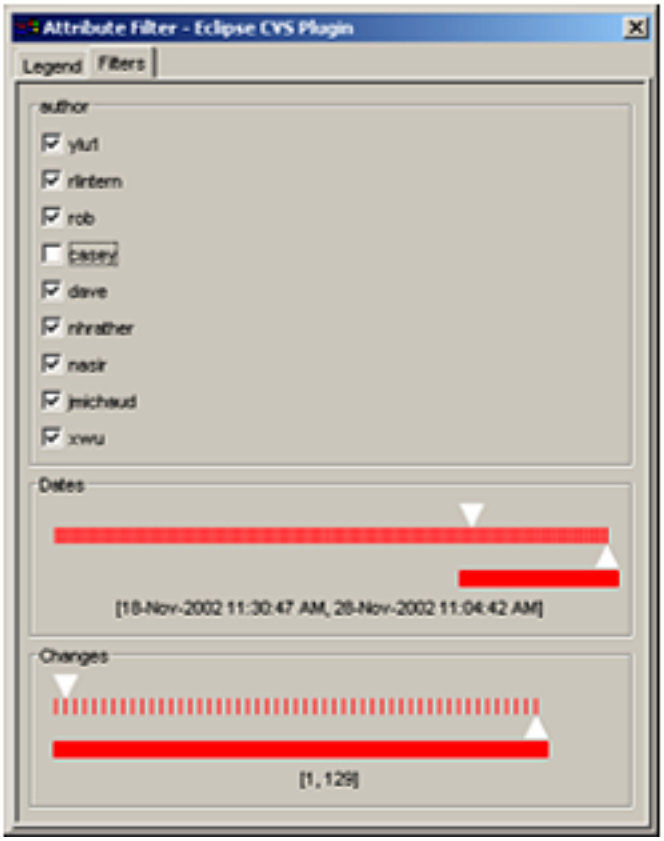


Figure 2.8. *The dynamic filters of Xia. The user can filter the nodes of a diagram by specifying the nodes of interest according to the author that last changed the node, the period of time that the node was changed in, and the number of times that the node was changed.*

nodes that were changed 10 times within the last month. The two filtering mechanisms can be combined.

2.3 Architecture-centric Visualization

There are three main approaches to visualization of architectural evolution: visualizing the entire architectural evolution at once, showing the architectural differences (deltas) between two releases, and a unified approach that gives both an overview of the architectural change

and allows the highlight of specific differences between releases.

2.3.1 Overview

Gall, Jazayeri and Riva [13] developed a technique for visualizing software release histories using 3D diagrams. Each release had its structure displayed as a 2D diagram, and the 3D diagram displayed a succession of these releases on a line, as shown in Figure 2.9. The color of the elements in the structure indicate how many times the elements have been modified. For example, black is mapped to 0 modifications, so items shown as black in the first release have not been created yet. Between the first and second release, elements that change color from red to pink were modified in the second release, items that change from black to red were introduced in the second release, and items that remain red were not modified in the second release. This 3D view enables a user to detect the main changes in the evolution of the system. As well, the user can zoom into particular subsystems or modules to examine them more closely, or use 2D color histograms to, for example, view the percentage of elements in a particular release that have been modified once.

2.3.2 Deltas

GASE, Graphical Analyzer for Software Evolution [22] is an example of an approach geared towards displaying architectural deltas to compare two releases of a software system, which we refer to as Release 1 (the earlier release) and Release 2 (the later release). Modules are drawn as rectangles, and relationships between modules are drawn as edges between the modules. Modules and relationships are colored red if they were added in Release 2, blue if they were removed in Release 2, and grey if they are present in both Release 1 and Release 2. Holt and Pak, the authors of GASE, also mention that their visualization approach could be extended to viewing multiple releases by using the intensity of color to represent how recently a module or relationship was added or removed.

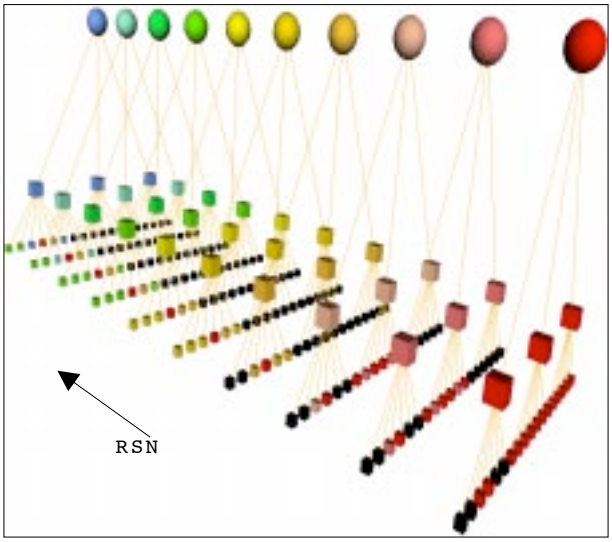


Figure 2.9. *A 3D overview of software evolution. The line represents a series of releases of the software system, with each 2D diagram displaying the structure of the software system at that version. The color of the elements indicate how many times they were modified up to that version.*

2.3.3 Unification

It is useful to have both an overview of how the software architecture has evolved and a more detailed description of the changes between particular releases. One tool that provides both these views is Beagle [41]. Beagle's main view, shown in Figure 2.10, consists of two panels. The panel on the left uses a tree view to show the structure of the software system at a particular version. The panel on the right shows a dependency diagram describing how the software has evolved over a selected number of versions either to or from the version shown in the left panel. Color is used to indicate entities and relationships that have been added, modified, and deleted, with intensity used to indicate ordinal attributes, such as how long ago an entity was added.

Beagle integrates a number of innovative features. One of these is what they term *Bertillonage* analysis, used to detect similar entities from one release to another. Many approaches have the limitation that when, for example, a function is renamed in a release, it is viewed as the deletion of one function (the original name) and the creation of a new function (the new name). The Beagle approach more accurately shows this change as a modification of one function. Bertillonage analysis was extended by Zou to support detecting the merging and splitting of entities [49], and is a similar technique to that used by UMLDiff [47].

Beagle also provides metric views and a powerful query mechanism. Metrics are provided about particular software entities when the user selects them in the structure tree, with different metrics provided for functions and for files. The query mechanism is provided in two ways; the database can be queried via SQL, or the user can use the interface provided to query what releases to display. The SQL method is flexible, but results are displayed in datasets or evolution graphs. The user interface method returns architecture visualization results, but is very limited in the types of queries it supports.

Another technique for viewing both an overview of how software changes over time and a more detailed view of particular times is to use animation. However, as Beyer and Hassan [2] point out, viewing a movie, while helpful for describing the dynamic process

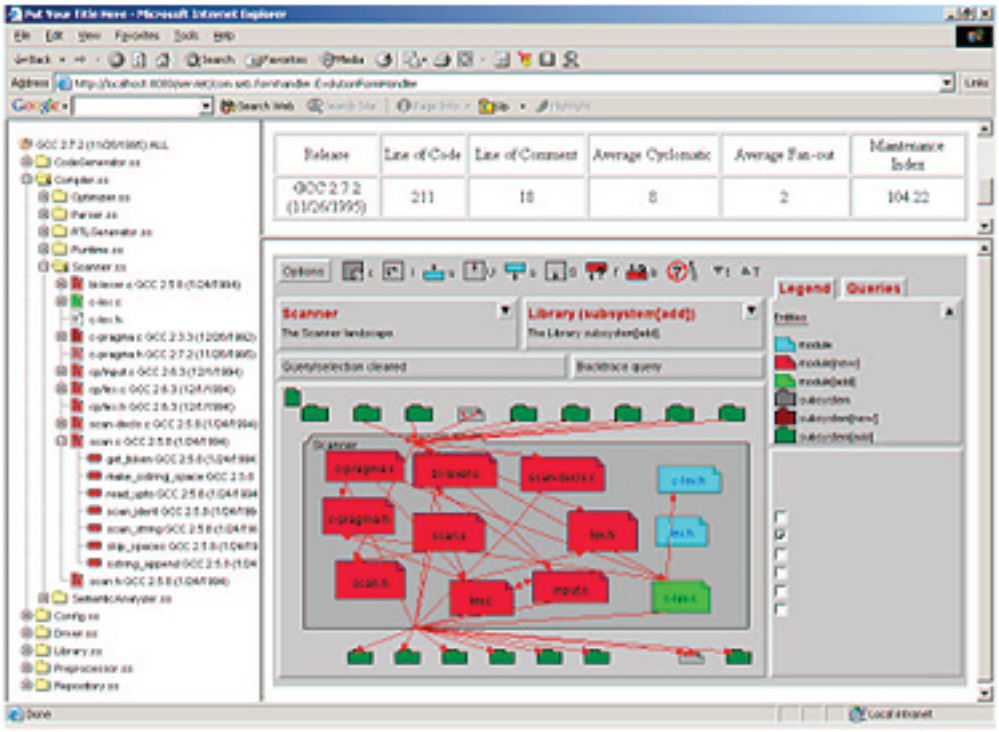


Figure 2.10. *The main view of Beagle [41]. In this case Beagle is being used to compare GCC 2.0 to GCC 2.7.2 The panel on the left shows the structure of GCC 2.7.2. The panel on the right shows a dependency diagram comparing GCC 2.7.2 to GCC 2.0. Color indicates how each entity/relationship was changed between the two versions.*

of evolution, means that developers are likely to miss interesting events and have difficulty focusing on part of the evolution. Approaches that visualize software evolution using animation must provide the ability for users to control the animation.

An example of an animation approach is the flipbook animation style used in GEVOL [5]. GEVOL shows the inheritance, control flow and call graphs for a software system. The user can flip between time intervals (the default interval is one day) to show how these graphs have changed. Some of these graphs may be enormous, with millions of nodes. To make these graphs manageable, the user can filter the graph by specifying with a regular expression what nodes the user is currently interested in. Before display the graph will be preprocessed with this filter. As well, nodes can be colored to indicate how they have been modified and who has modified them.

YARN [21] uses animation to display the architectural dependency graph evolving over time. YARN currently includes a play and pause button, and several different coloring schemes to emphasize different aspects of the evolution. More sophisticated user interaction is also planned.

One interesting item to note about YARN is that each transaction is represented as a frame in the animation, rather than, for example, the Beagle approach where only releases of a system can be viewed, or the GEVOL system where the user must specify the time slice that they are interested in. Key to supporting this feature is that, unlike a lot of other approaches, the fact extractor used by YARN can tolerate code that won't compile.

2.4 Metric-centric Visualization

A number of the artifact- and architecture-centric tools previous discussed incorporate metrics. Although this classification is fuzzy, we identify Metric-centric visualizations as being primarily centered around viewing metrics. An example of a simple approach is the Evolution Matrix [25] aimed at showing how each class in the system has evolved over time. Classes are drawn as rectangles laid out in columns and rows, with each column being one

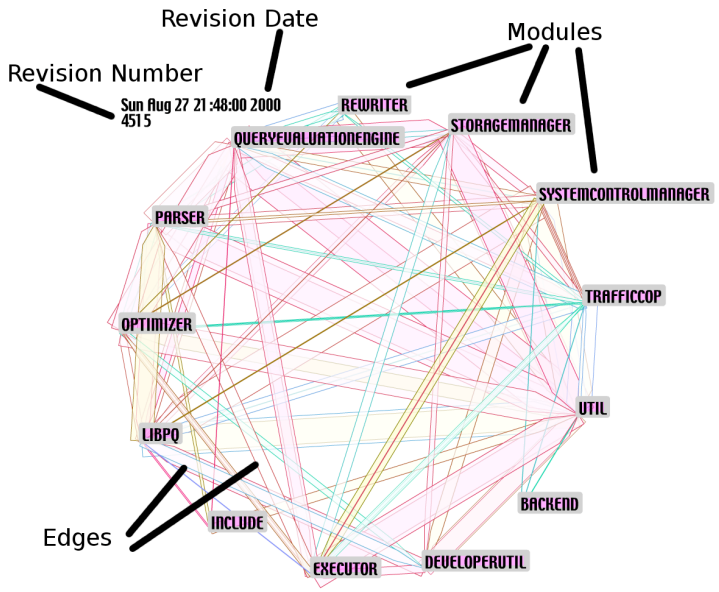


Figure 2.11. *A frame of a YARN animation [21] showing a period of evolution to PostgreSQL.*

version, and each row showing the evolution of one class. As well as showing when classes were added and removed to and from the system, the approach allows for viewing how metrics about each class have changed over time by making the width of each rectangle proportional to one metric value and the height another.

The approach allowed Lanza to identify several types of classes that might exist in a software system. For example, a class that explodes in size is termed a supernova, and Lanza suggests this sudden change may need to be carefully examined to guard against bug introduction. A similar approach was used in Yesterday's Weather [19] to detect classes that have changed a lot in the recent past and so may prove to be the most *evolution-prone* parts of the system, demanding particular attention when attempting to understand the system.

A significant limitation with the Evolution Matrix is that it does not indicate relationships between classes or metrics about those relationships. An example of where this would

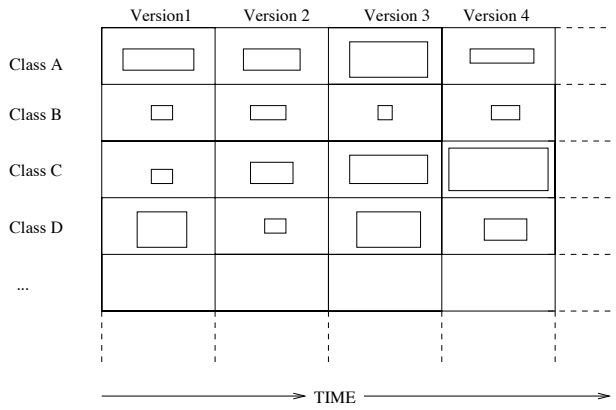


Figure 2.12. *A sketch of the Evolution Matrix visualization [25]. Classes are laid out on the y-axis, and versions of the software system on the x-axis. The size of the rectangle in every cell indicates metrics about the class at a particular version.*

be useful is in identifying implicit relationships between classes that cause these classes to need to change together. Bieman, Andrews, and Yang [3] describe an approach to help detect these evolutionary couplings. First they compute three metrics. For each class they compute the *local change-proneness (LCP)*, the number of change reports that involve only that class. Between pairs of classes they compute the *pair change coupling (PCC)*, the number of times the classes have been involved in the same change report. For each class they compute the *sum of pair couplings (SPC)*, the sum of all pair change couplings that include that class.

The results can be displayed in boxplots, diagrams of the architecture of the system highlighting the change prone classes, or, as in Figure 2.13, the change architecture of change-prone classes. In the figure, the number for each relationship is the PCC, and each class box displays the LCP and SPC values. Classes that the researchers found to play a role in a design pattern are shaded.

Many metric-centric approaches are restricted by the number of metrics that can be displayed in one view. RelVis [31] is an approach that allows the visualization of multiple metrics over time using *kiviat* graphs. Each metric is drawn along a line from a central

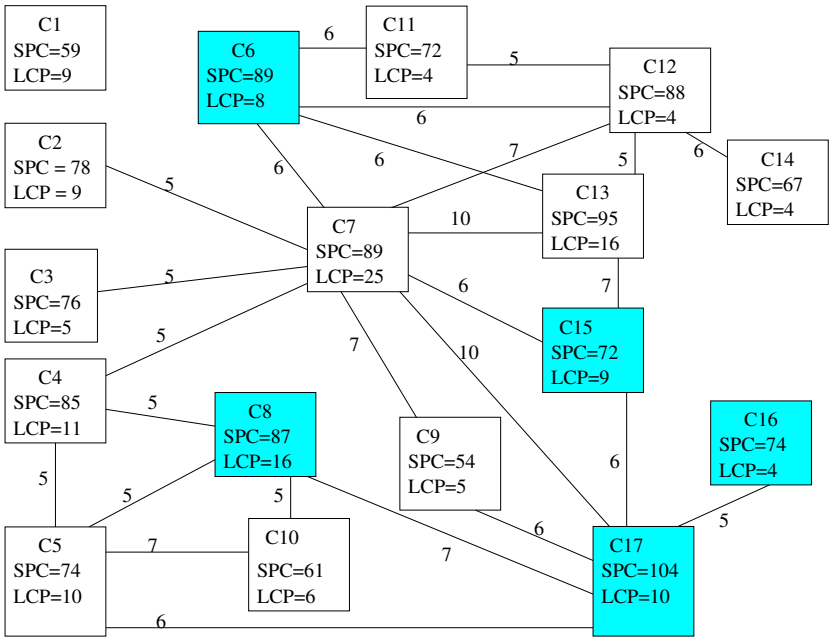


Figure 2.13. *A view of the change architecture of change-prone classes [3] . The nodes are classes, the lines indicate a relationship between two classes, and the numbers indicate different metric values.*

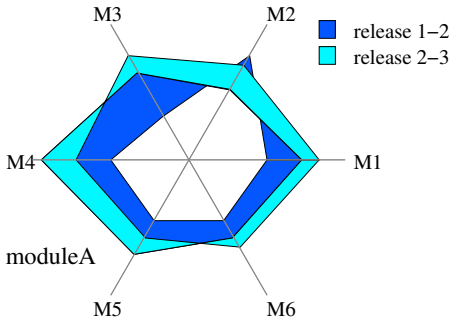


Figure 2.14. *A kiviat graph [31] of how 6 metrics have changed over 3 releases*

point. The value of the metric for a particular release is the offset from the central point along the metric line, normalized to a maximum length. Color can be used to indicate how a collection of metric values has changed between two releases.

Figure 2.14 shows a collection of six metrics for one module over three releases. Between releases 1 and 2, the value for metric 2 has decreased, while all other metrics have increased. Between releases 2 and 3, metrics 2, 3, and 6, have increased, and metrics 1 and 4 have decreased. The figure only indicates how metrics of one module have changed over time, but diagrams can be created with multiple graphs for each module in the system, and rectangles between the modules to indicate how some relationship metrics, such as coupling, have evolved. Alternatively, if the user is interested in viewing multiple metrics about a relationship, kiviat graphs can be drawn with each graph representing multiple metrics about a module's participation in a relationship.

2.5 Feature-centric Visualization

A feature of a software system can be described as an observable and relatively closed behavior or characteristic of a software part [32]. Unlike the three other categories of visualization tools we presented, the features of a software system are a target that is not possible to extract directly from the source code stored in a SCM system's repository.

Thus, it is necessary that feature-centric approaches include multiple software trails in their analysis.

Fischer and Gall [10] developed a feature evolution view by using data from a software repository, as well as data from an issue tracking repository and execution traces. Their approach allows a mapping of features to the source code that implements the features, shows how features have evolved over time, shows coupling between features, and can suggest where the software is undergoing architectural deterioration. After extracting the MRs from the software repository and the problem reports (PRs) from the issue tracking system, a re-linking was created between the MRs and the PRs they were designed to fix. Next, they defined some test cases designed to exercise certain features. They executed these tests, and by examination of the call-graph they were able to map the abstract concept of features onto the files that implement those features.

Fischer and Gall introduce two views of how features have evolved over time. The *feature view* is a graph designed to show dependencies between features. Here, the dependency between two features is considered to be proportional to the number of PRs that were related to both features. In the graph, an example of which is shown in Figure 2.15, features are nodes, dependencies between features are edges, and the width of an edge indicates the degree of dependence. To make the graph manageable, there needs to be a minimum number of PRs shared by two features for a dependency to be shown in the graph.

The *project view* shows the relationship between the directory structure of the project tree and the PRs of the project. Directories are shown as nodes. Dashed lines between the nodes indicate that they are related in terms of the project tree directory structure (some part of their path is common). PRs that have affected two directories are shown as a solid line between two nodes, with the width of an edge indicating how many PRs the directories share. Connected nodes are attempted to be placed closer together. Figure 2.16 shows an example project view.

Another approach to visualizing feature evolution was developed by Greevy, Ducasse, and Gîrba [20]. Their analysis uses the SCM system's repository and execution traces

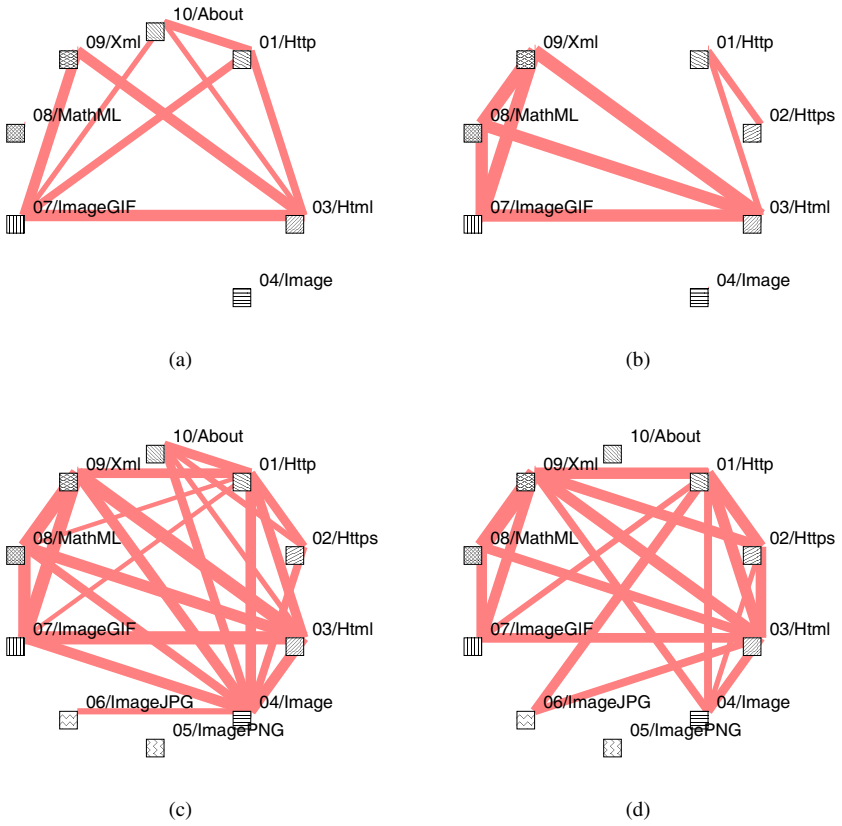


Figure 2.15. *A feature view [10] showing features evolving over time: (a) 1999; (b) 2000; (c) 2001; and (d) 2002. Features are shown as nodes, and dependencies between features shown as edges.*

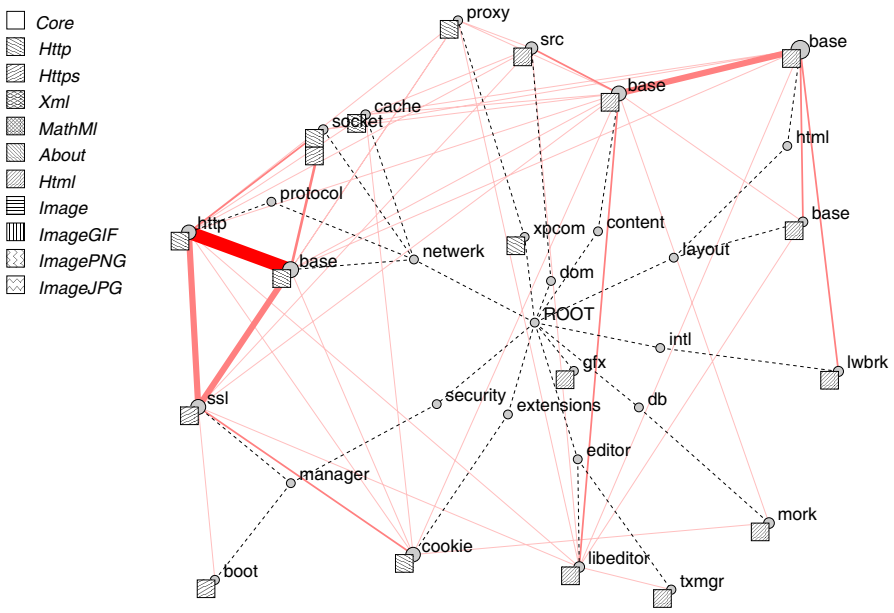


Figure 2.16. *A sample project view [10]. Nodes represent directories, dashed lines represent a relationship between the directory structure of two nodes, and solid lines indicating PRs that have affected multiple directories.*

to identify features and feature evolution. Their visualization shows the features of the system in terms of collection of classes that participate in implementing each feature, with classes shown as small colored rectangles. The colors of the rectangle indicate the feature, with one class belonging to one or more features. Their *feature history view* describes the evolution of a feature over time, showing the order that classes were added to the feature, and what classes have been removed from the feature.

2.6 Summary

There is a huge amount of data stored in software trails, and an important part of making effective use of this data is providing visualizations that can summarize the data to users. No one approach to visualization can hope to display all the information that a user might be interested in, indeed most tools contain multiple views. As well, most tools provide means to query and filter the view to respond to the widely varying needs of different users.

A common means of filtering the data stored in software trails is a *time-based* approach. Time-based filtering has been used by artifact-, architecture-, metric-, and feature-centric approaches, and is an intuitive way of limiting the data a user is shown.

Our change-set approach can be seen as a filtering mechanism more flexible than time-based filtering. As a filtering mechanism the change-set approach is designed to make it easier for a user to examine information the user is interested in by hiding information the user is not interested in. Time-based filtering assumes what a user is interested in can be expressed as an unbroken period of time. We believe change-sets are useful in situations where that assumption is not valid.

Chapter 3

Model and Method

The goal of our approach is to allow a user to examine the impact of the MRs in the change-set on the architecture of the system, while ignoring the effect of other MRs that are not part of the change-set. Accomplishing this goal requires both computing the effect of a change-set, and illustrating its effects through annotated architectural diagrams. Our computation and visualization are designed to be easily used in combination with existing architectural diagrams. This chapter discusses the model of how we show the impact of a change-set on a software system's architecture. The following chapter describes the prototype tool we developed that implements our model using some specific diagrams and annotations.

Garlan and Perry define software architecture as, "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [14]. As with most architecture-centric approaches, when we attempt to show the evolution of the architecture we focus on the first two parts of this definition, showing the evolution in terms of the entities (components) of a system (such as packages, classes, methods, functions, etc.), and their interrelationships. We assume that some extraction has been performed on the SCM system and we have access to information about, for every MR:

MR System State. The entities that exist in the system at each MR, and what relationships exist between those entities.

MR Impact. The changes made in the MR to the entities and relationships of the system.

MR Metadata. Information about the MR itself, such as the author who made the MR and a log description of the MR.

We refer to the state of the system immediately before the first MR in the change-set as the *starting version* of the system, and the state immediately after the last MR in the change-set as the *final version* of the system. The period of time between the starting and final version is referred to as the *period of interest*. We term the MRs in this period of interest as the *total-set*, with the change-set a subset of the total-set.

In order to determine the impact of the change-set on the system architecture, we have developed a process that takes in the starting version of the system, the total-set, and the change-set. The starting version of the system and the total-set are used to keep track of a representative architectural description of the system over the period of interest. The change-set is used to identify the changes to the architecture we need to track. The result of the process is a list of entities/relationships that existed in the system over the period of interest and, for each entity/relationship, a list of changes that MRs in the change-set have made to the entity/relationship, as well as a *state* that summarizes how the entity/relationship was affected by the change-set.

In order to show the impact of the change-set on the system architecture, we have developed a method for annotating an architectural diagram to create what we term an *architectural impact view*. The views can be based on any standard diagram notation for describing a system architecture. The annotations to the diagram indicate how each entity/relationship was affected by the change-set.

3.1 Change-set Selection

Our computation begins when a user selects a change-set of interest. As a change-set is just a list of MRs, selecting a change-set is a simple operation - the user just needs to specify what MRs belong in the change-set. However, specifying a change-set by enumerating

the MRs that belong to it is both cumbersome and, when a system has thousands of MRs, unfeasible. In practice, change-sets are often built by creating a query to select MRs which satisfy certain properties. These properties are mostly specified from the MR Metadata, the MR Impact, or synthesized information about the MR.

The MR Metadata provides data about the MR. This includes a:

Date. The date the MR was made.

Time. The time the MR was made.

Author. The developer who made the MR.

Log. A textural description entered by the developer to describe the purpose of making the MR.

Examples of change-sets that might be composed by querying metadata properties include, "changes made after this date", or "changes made by this particular author".

The MR Impact describes how the MR has affected the architecture by specifying what changes were made in the MR to the system's architectural entities and relationships. Examples of change-sets that might be composed by querying MR Impact properties include, "MRs that added a function", or "MRs that modified a class".

Synthesized information can be built by a more in-depth examination of the impact of the MR on the architecture. For example, Dig et al. [8] have developed an algorithm to detect likely refactorings between two versions of a software component. Applying this algorithm to the impact of each MR, it could be determined when an MR represents a refactoring. This synthesized information might then be used to create the change-set of "MRs that involved refactoring".

3.2 Computing the Impact of a Change-set

The user-selected change-set provides one of the inputs to the computation process, and allows us to derive the other inputs. We determine the starting version of the system by

finding the first MR in the change-set and retrieving the system state at that point. We determine the total-set by finding all MRs in the system between the first and last MR of the change-set.

The starting version of the system is used to build a "current list" of entities and their relationships, and, for each entity/relationship associating an empty "annotation list". Although we are only interested in the effect of the change-set, our process works over the total-set. Evaluating the impact of some MRs while ignoring others is not trivial. An ignored MR might affect the state of the system by adding, changing, removing, renaming, or moving entities/relationships - changes that need to be kept track of so as to create a meaningful description of the evolution of the state of the system. In total we have identified four main types of changes that might occur to an entity/relationship in an MR:

Addition. Entities/relationships were added in the MR. In this case it is necessary to keep track of the existence of the entity/relationships, whether or not the MR is part of the change-set.

Deletion. Entities/relationships were deleted in the MR. This change only needs to be recorded if the MR is part of the change-set.

Modification. Entities/relationships were affected in the MR. This change only needs to be recorded if the MR is part of the change-set.

Metadata modification. Entities/relationships had their metadata, such as their name, changed in the MR. It is necessary to keep track of the most recent value of the metadata, whether or not the MR is part of the change-set.

The current list and annotation lists are built up by iterating over each of the MRs in the total-set in chronological order:

- For every entity/relationship in the current list that has had its metadata altered: add this event to the appropriate annotation list. As an example, this would record if an MR has changed the name of an entity.

- For every entity/relationship added in the MR: add it to the current list, and create an empty corresponding annotation list.
- If the MR is in the change-set, for every entity/relationship in the current list that was added, modified, or deleted in the MR: add the event to the appropriate annotation list.

At this point in the process, each entity/relationship present at some point during the period of interest will be present in the current list, with its metadata reflecting its most current value (for example, renaming an entity/relationship will update its metadata). We determine the state of each entity/relationship from its annotation list:

- If the entity/relationship did not have any annotation then set its state to *unchanged* (this could include an entity/relationship added by an MR not part of the change-set);
- otherwise, if its first annotation is *added* and the last one *deleted*, then set its state to *phantom*;
- otherwise, if its first annotation is *added*, then set its state to *added*;
- otherwise, if its last annotation is *deleted* then set its state to *deleted*;
- otherwise, if it contains a *modified*, *deleted*, or *added* annotation, then set its state to *modified* (this could include an entity/relationship that was deleted and then added again);
- otherwise set its state to *metadata*.

So, each entity/relationship will be assigned to one of the following states:

Added. Entities/relationships were added in some MR. They do not exist in the starting version of the system.

Deleted. Entities/relationships were deleted in some MR. They do not exist in the final version of the system.

Phantom. Entities/relationships were added in some MR, and then deleted in a later MR. They do not exist in the starting or final version of the system.

Modified. Entities/relationships were affected by at least one MR (and are not added, deleted, or phantom).

Metadata. Entities/relationships had their metadata, such as their name, altered by at least one MR (and are not added, deleted, or phantom).

Unchanged. Entities/relationships were not affected by any MRs.

The result of the computation is a description of the entities/relationships that existed in the system over the period of interest (stored in the current list), and, for each entity/relationship, a filled annotation list that keeps track of changes to the entity/relationship, and a state value summarizing the effect of the change-set on the entity/relationship.

3.3 Architectural impact view

We show the impact of the change-set on the system's architecture using architectural impact views. These diagrams can be broadly defined as annotated architectural diagrams. There are many types of architectural diagrams that could be used, including UML diagrams, E-R diagrams, and boxes and lines diagrams. As these methods of describing a system's architecture do not rely on the use of color to represent information, we use color to annotate the architectural diagram with a description of how the system was affected by the change-set. However, other methods of annotation could be used if the desired architectural diagram already made use of color.

The architectural diagram is used to show the entities/relationships in the system that were affected by the MRs in the change-set, as well as to provide context about how these entities/relationships fit into the overall architecture of the system. For example, if a particular entity was changed by an MR in the change-set we definitely want that entity to appear in the architectural diagram. We also want to show some relationships between that entity and other entities in the system, even if those relationships were not modified.

The annotations to the diagram are used to indicate how the entities and relationships in the diagram were affected by the MRs in the change-set. The primary type of anno-

tation, discussed in the examples below, uses color to show the final state of each entity/relationship. Other annotations can indicate more details about the change-set, making use of the annotation list of each entity/relationship. These annotations may also use color, or other means of annotation such as overlays.

3.4 Examples

To exemplify the architectural impact view we will use a simple system that is composed of five architectural entities: A, B, C, D, and E (they could represent classes in the system). There exist three authors (A1, A2, and A3) that have made changes (a total of 6 MRs) to the system between the system's two releases (R1 and R2). This timeline does not include any branching; to simplify our visualization requirements we are currently only considering trunk changes (for more details about branches and trunk changes please see Section 4.1.1).

The entities in our example system and their changes are depicted in Figure 3.1 (in this scenario we do not track changes to the relationships, except when an entity is added or deleted). Figure 3.2 shows the state of the system at R1 and R2. In this example we are using simplified relationship diagrams which display entities and the existence of some relationship between them. By comparing the "before" and "after" diagrams one can infer the entities and relationships that have been added and deleted (but not modified or phantom entities/relationships).

3.4.1 All MRs

One example of a change-set that a user might be interested in is the total amount of change to the system, including all MRs. Table 3.1 shows the process of computing the net-impact of this change-set by describing the annotation lists of each entity/relationship at every step in the process.

1. In the Start State (MR 0), each entity/relationship existing at that point in the system

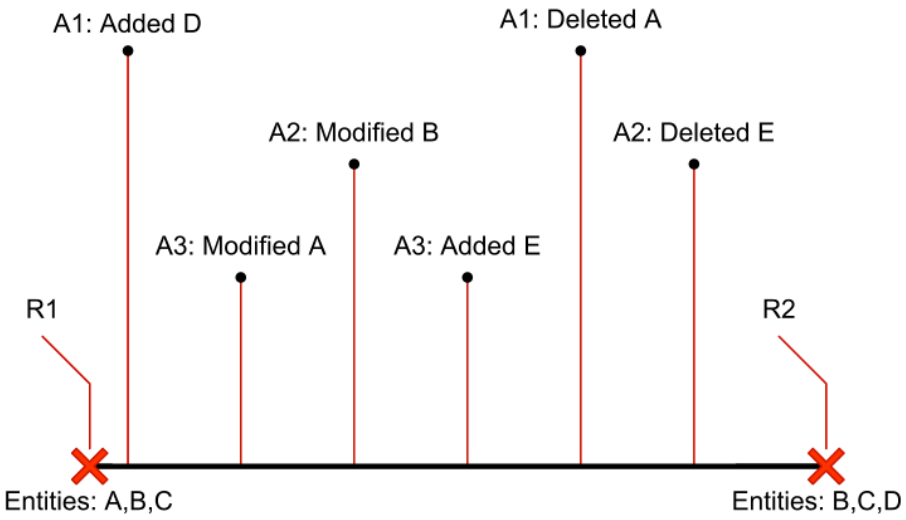


Figure 3.1. *Evolution of the example system over revisions R1 and R2. There have been six MRs made between the two revisions, with the author and a description of the effect of the MR indicated on the diagram.*

is assigned an empty annotation list.

2. In MR 1, D, and its relationship, were added to the system. Corresponding annotation lists were created, and an "added" change was inserted into the lists.
3. In MR 2, A was modified. The "modified" change was inserted into A's annotation list.
4. In MR 3, B was modified. The "modified" change was inserted into B's annotation list.
5. In MR 4, E and a relationship it participates in were added. Corresponding annotation lists were created, and an "added" change was inserted into the lists.
6. In MR 5, A was deleted. The "deleted" change was inserted into A's annotation list, and the annotation list of the relationship that A participates in.
7. In MR 6, E was deleted. The "deleted" change was inserted into E's annotation list, and the annotation list of the relationship that E participates in.

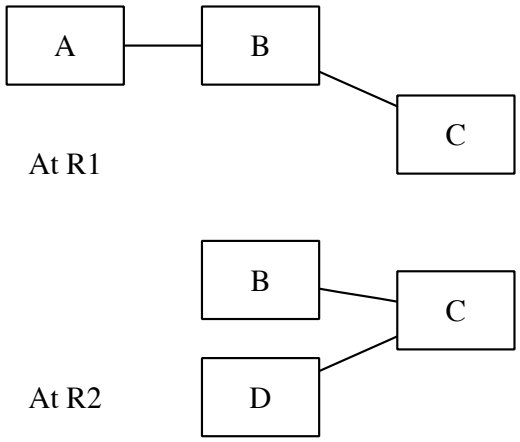


Figure 3.2. *Architecture diagrams for the example system at R1 and R2. Nodes indicate entities, with lines between nodes showing some relationship between the entities.*

The state of the annotation lists at MR6 determine the resulting architectural impact view, shown in Figure 3.3. We use the following colors to depict the state of each entity/relationship: green represents *added*; yellow, *modified*; black, *deleted*; pink, *phantom*; grey, *unchanged*; and blue, *metadata*. This view provides a comprehensive overview of what architectural change occurred between R1 and R2: A has an annotation list ending in a deletion event, and so is shown as deleted (black), B has an annotation list containing one modification, and so is shown as modified (yellow), C has an empty annotation list, and so is shown as unchanged (grey), D has an annotation list containing one addition, and so is shown as added (green), and E has an annotation list with an addition and then a deletion, and so is shown as phantom (pink). From this view it is also possible to derive the architectural state of the system at points R1 and R2. All the entities that are not added nor phantom exist at R1, and all the entities that are not deleted nor phantom exist at R2.

3.4.2 The impact of MR 5

Table 3.2 shows the process of computing the net-impact of MR 5.

1. In the Start State (MR 4), each entity/relationship existing at that point in the system

Table 3.1. *Computing the impact of all MRs*

$CurrentList$	$StartState(MR0)$	$MR1$	$MR2$	$MR3$	$MR4$	$MR5$	$MR6$
A	{}	{}	{M}	{M}	{M}	{M,D}	{M,D}
B	{}	{}	{}	{M}	{M}	{M}	{M}
C	{}	{}	{}	{}	{}	{}	{}
D	N/A	{A}	{A}	{A}	{A}	{A}	{A}
E	N/A	N/A	N/A	N/A	{A}	{A}	{A,D}
A to B	{}	{}	{}	{}	{}	{D}	{D}
B to C	{}	{}	{}	{}	{}	{}	{}
D to C	N/A	{A}	{A}	{A}	{A}	{A}	{A}
E to D	N/A	N/A	N/A	N/A	{A}	{A}	{A,D}

is assigned an empty annotation list.

2. In MR 5, A was deleted. The "deleted" change was inserted into A's annotation list, and the annotation list of the relationship that A participates in.

The state of the annotation lists at MR 5 determine the resulting architectural impact view, shown in Figure 3.4. A has an annotation list ending in a deletion event, and so is shown as deleted. Every other entity has an empty annotation list, and so is shown as unchanged. As in the previous example, it is possible to determine the state of the system directly before and directly after MR5.

3.4.3 Author A2's MRs

Table 3.3 shows the process of computing the net-impact of Author A2's MRs. Unlike the previous two examples, this is a case in which the change-set is not equal to the total-set.

1. In the Start State (MR 2), each entity/relationship existing at that point in the system is assigned an empty annotation list.

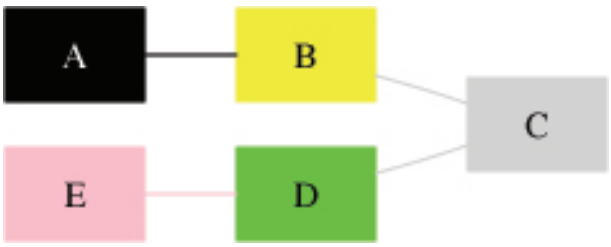


Figure 3.3. *An architectural impact view comparing R1 to R2 (change-set is equal to all the MRs in between both releases). A is shown as black (deleted), B yellow (modified), C grey (unchanged), D green (added), and E pink (phantom).*

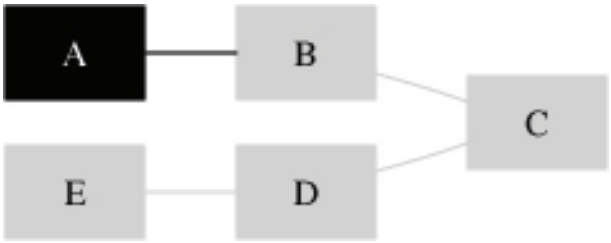


Figure 3.4. *The impact of the fifth MR. In this case the change-set and the total set both contain only one MR. A is shown as black (deleted)*

2. In MR 3 (part of the total-set and change-set), B was modified. The "modified" change was inserted into B's annotation list.
3. In MR 4 (part of the total-set, but not part of the change-set), E and a relationship it participates in were added. Corresponding annotation lists were created, though, because the MR is not part of the change-set, the "added" event was not inserted in the annotation lists.
4. In MR 5 (part of the total-set, but not part of the change-set), A was deleted. As MR 5 is not part of the change-set no modification was made to any annotation list.
5. In MR 6 (part of the total-set and change-set), E was deleted. The "deleted" change was inserted into E's annotation list, and the annotation list of the relationship that E participates in.

Table 3.2. *Computing the impact of MR 5*

$CurrentList$	$StartState(MR4)$	$MR5$
A	{}	{D}
B	{}	{}
C	{}	{}
D	{}	{}
E	{}	{}
A to B	{}	{D}
B to C	{}	{}
D to C	{}	{}
E to D	{}	{}

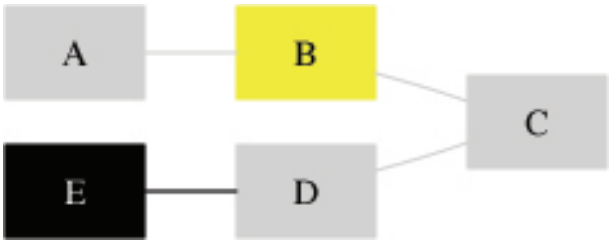


Figure 3.5. *An architectural impact view of the changes of Author A2. B is yellow (modified), E is black (deleted), all other entities are unchanged.*

The state of the annotation lists at MR 6 determine the resulting architectural impact view, shown in Figure 3.5. B has an annotation list containing one modification, and so is shown as modified. E has an annotation list containing one deletion, and so is shown as deleted. Because the effect of some intermediary changes (MR 4 and MR 5) were not included, the diagram does not indicate the state of the system before and after. For example, although A was deleted in MR 5, it is shown as unchanged.

Table 3.3. *Computing the impact of Author A2's MRs*

$CurrentList$	$StartState(MR2)$	$MR3$	$MR4$	$MR5$	$MR6$
A	{}	{}	{}	{}	{}
B	{}	{M}	{M}	{M}	{M}
C	{}	{}	{}	{}	{}
D	{}	{}	{}	{}	{}
E	N/A	N/A	{}	{}	{D}
A to B	{}	{}	{}	{}	{}
B to C	{}	{}	{}	{}	{}
D to C	{}	{}	{}	{}	{}
E to D	N/A	N/A	{}	{}	{D}

3.4.4 Summary

This chapter described the method and model we use to examine a software system's architectural evolution. Our model is centered around the concept of a change-set, a subset of the MRs in the system. Each MR has a large amount of data associated with it, including the MR Impact properties, the MR Metadata properties, and synthesized information about the MR. Useful change-sets can be built up by performing queries to select MRs that satisfy particular criteria.

Once a change-set has been selected, we describe a method for computing the impact that a change-set had on the system's architectural evolution. This method works over the total-set, all the MRs made in the period between the starting version of the system (at the first MR in the change-set) and the final version of the system (at the final MR in the change-set). Our computation process uses information about how the system changed over the total-set to describe the effect that the change-set had on the system. The result of the process will be that each architectural entity and relationship is assigned a state summarizing the net-effect the change-set had on the entity/relationship.

To visualize the effect of a change-set we have developed architectural impact views.

These views work by extending existing architectural visualization methods with annotations that describe the state of each entity/relationship. We showed examples of architectural impact views built on top of a simple boxes and lines architectural diagram, with the impact of the change-set indicated with color annotations.

Chapter 4

Prototype Tool

The previous chapter discussed our method for computing the impact of the MRs of a change-set, and the architectural impact views we use to illustrate the impact. This chapter discusses implementation details of the prototype tool we developed that implements our model and method. This chapter includes details of how we transform data into a form that enables us to perform the computation process, how we create architectural impact views, and the specifics of the particular architectural impact views we create. The chapter begins by discussing the preprocessing that we do to the SCM system. This results in a database from which we can extract information about the impact of change-sets. The chapter continues by describing the prototype tool that makes use of our database to allow a user to select a change-set of interest and then view a descriptive architectural impact view.

We have termed our prototype tool *Motive*. We hope that by allowing users to examine the net-effect of various change-sets Motive will help users achieve a better understanding of the software evolution of a particular software system, and gain insight into the motivations of the original authors of the system.

4.1 Preprocessing

In the previous chapter we assumed that extraction had been performed on the SCM system and we had access to information about, for every MR, the MR System State, the MR Impact, and the MR Metadata. This section describes how we recover that information.

Architecture-centric visualization tools commonly use a preprocessing phase to transfer data from an SCM repository to a database. The visualization tool can then operate on the database rather than directly on the data in the repository. Zimmermann and Weissgerber have described the two main advantages the inclusion of a preprocessing phase confers [48]. The first is performance - accessing an SCM repository is slow in comparison to accessing the same information stored in a database. The second is that it enables the study of changes to architectural entities. SCM systems typically store information about changes in the form of what files and lines of code have changed. This must be further analyzed in a preprocessing phase to extract the architectural entities and relationships.

A third advantage of the preprocessing phase is that it decouples our visualization from our data extraction process. We are currently limiting our approach to one SCM system (concurrent versions system - CVS [6]) and one programming language (Java). CVS was chosen because it is a popular choice of SCM system for open source software developers, and supporting CVS allows us access to a large number of software systems. Java was chosen because it is a popular language and it is far easier to statically analyze than languages that support pointers such as C++. However, as our visualization database design is not directly dependent on either CVS or Java, our implementation could easily be extended in the future to support more languages and SCM systems.

Zimmermann and Weissberger [48] summarize the four tasks commonly performed in the preprocessing of CVS repositories as:

1. Data Extraction

   Transfer the data in the CVS repository to a database.

2. Restoring Transactions

When a developer checks in multiple files at the same time into CVS, the information that these files were checked in together, and are therefore most likely part of the same logical transaction, is lost. Often researchers wish to recover this information.

3. Mapping Changes to Entities

   CVS stores changes in terms of what files were changed. In many approaches, researchers are interested in what entities, such as classes, methods, or functions, have changed.

4. Data Cleaning

   Infrastructure changes, such as the merging of branches or the renaming of an include file may seem to affect a huge number of entities despite the fact that their logical effect can be described very simply. These types of changes may require special treatment so as to not obscure interesting patterns about the evolution of the software.

Our preprocessing broadly follows this outline, though the details are different. As Figure 4.1 shows, we use a three-step process. In Step 1, we use softChange to carry out both the Data Extraction and Restoring Transactions tasks. In Step 2, the source code of the system at each MR that affects the trunk is extracted from the softChange database, scanned, and stored in a visualization database as the state of the system's architecture at that MR. In Step 3, for every trunk MR, the state of the architecture at that point is compared to the previous state, and the changes added to the database. Steps 2 and 3 together accomplish the Mapping Changes to Entities task. Currently, we are not performing the Data Cleaning task.

4.1.1 Step 1- Data Extraction and Transaction Recovery

softChange is a tool created by Mockus and German for studying software trails. Although we are using softChange as a CVS fact extractor, softChange does include the capabilities to consider other software trails, including, for example, issue tracking repositories such as Bugzilla [38]. softChange is a mature piece of software that has successfully been used to

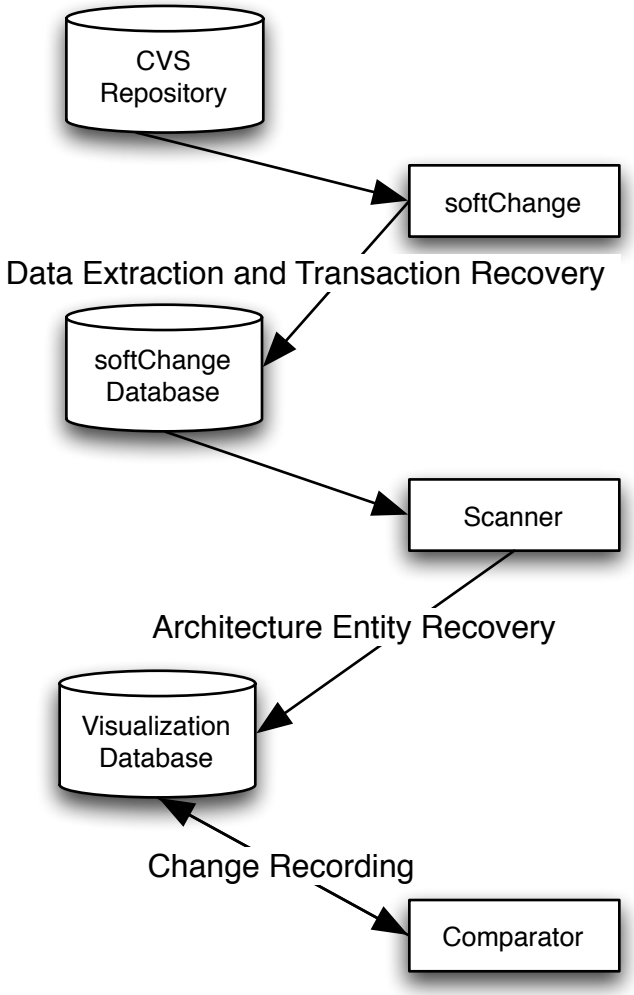


Figure 4.1. *An overview of preprocessing. Arrows show how data, starting at the top of the diagram, flows from a data source, through some data manipulation process, and to another data source. There are three main phases of preprocessing indicated on the diagram.*

mine the repositories of several large open-source software projects [15].

The output of softChange is the softChange database. From this database we can obtain the MR Metadata information - for each MR, its author, log, date, and time. As well, we can find the file revisions associated with each MR, information we can use in Step 2 to discover the MR System State, and in Step 3 to discover the MR Impact.

4.1.1.1 Branching

CVS allows the user to make a copy of some set of files in the software system. This copy is called a branch. Each CVS repository has a main set of files, called the trunk, used for primary development. Branches can be created to facilitate concurrent development processes. For example, one common use of branches is to create a release branch when the software seems close to being ready for a release to customers. While the release branch is undergoing thorough testing, developers can continue normal work in the trunk without destabilizing the release.

softChange extracts information about all MRs in the CVS repository. However, in further preprocessing steps, we currently limit the changes we consider to those that affected the trunk of the CVS repository.

4.1.2 Step 2 - Architecture Entity Recovery

In Step 2, for each trunk MR the most recent version of any files modified in that MR are added to a local directory of the software system's source code, scanned, and the current model of the software system's architecture is updated accordingly. The advantages of using a scanning-based approach, that extracts interesting software elements from the source code, over the more common parser-based approach, that attempts to build a full parse tree from the source code, are described in [21]. The primary advantage is that it is more robust, certain changes to the CVS repository may have made "broken the build" and frustrate a parser-based approach. The primary disadvantage is that it gives us less information,

for example parser-based approaches could allow for generating call-graphs of the system. Scanning is sufficient for the static architectural impact diagrams we are currently using.

The scanner we use is QDox [33]. This tool extracts a lot of information from the source file without trying to parse method bodies. Applying it to a Java system, we recover information about both entities (packages, classes, interfaces, methods, and fields) and relationships (implementation of interfaces and extension of parent classes).

Another relationship we are interested in is dependency. In our approach, class-to-class dependencies between two classes (A and B) are identified when Class A contains a field of type Class B, has a method that returns type Class B, has a method with a parameter of type Class B, or has an import statement that includes Class B. Class-to-interface, interface-to-class, and interface-to-interface dependencies are dealt with in a similar manner. Class-to-package and Interface-to-package dependencies are identified when a class or interface uses wildcard notation to import another package. Package-to-package dependencies are identified based on the dependencies of classes and interfaces within packages. As we are not parsing method bodies we may miss some dependencies if fully qualified names are used within a method.

The architecture entity recovery step concludes by storing the current architecture entities and relationships in the visualization database. Relationships are only stored if both entities currently exist in the database. So, for example, dependencies on the standard Java classes provided by Sun are not stored.

4.1.3 Step 3 - Change Recording

Our change recording step compares the entities and relationships existing in the software system in two consecutive MRs to identify the impact of the latter MR. Entities may have been added, modified, deleted, or left unchanged. Relationships may have been added, deleted, or left unchanged.

In the previous chapter we described the need to detect metadata changes, such as the moving or renaming of an entity. There do exist techniques to detect these types of changes,

such as the Bertillonage analysis approach developed by Tu [41]. Unfortunately, we have not yet integrated any such method, making our implementation currently more limited than our model. We consider packages and files the same if they have the same name, classes the same if they have the same name, file, and package, fields the same if they have the same name and class, methods the same if they have the same signature and class, and relationships the same if they are of the same type and between the same two entities.

Once the change recording step is completed, changes associated with the MR currently under examination are added to the database. It is not recorded what entities and relationships are unchanged, that can be inferred by comparing the entities that were modified to the entities existing in the system.

4.1.3.1 Incomplete Changes

Based on some preliminary testing we found a fairly common reason for breaking the build is not checking in all changes at once. For example, suppose that a new package, Package B, has been added to the system, and Package A has been made dependent on Package B. Further, suppose that the user who made the change first adds the dependency to Package A (in MR 1), then later realizes she has only partially checked in her changes and adds the new Package B to the system (in MR 2).

In early versions of our comparison module, the addition of the A to B dependency would never be added to the database. In MR 1, where Package A was modified, the comparison step would detect there is a new dependency, but, as Package B didn't exist in the system it would not add the dependency to the database. When Package B was added to the system in MR 2, it would not detect the addition of a dependency, which was related to the change of Package A, not the addition of B. Our solution is that if a relationship cannot be added to the database because its participatory entities do not both exist, we add the relationship to a local list. From each MR from then on, or until the relationship is successfully added, after the addition of every entity relationships in this list will be checked to see if any have become valid (both its entities exist) and can be added to the

system.

4.1.4 Visualization Database

The result of the extraction process is the population of the visualization database. Figure 4.2, describes the key entities of the database. From the database, any MR's Metadata, System State, or Impact can be easily retreived.

The MR Metadata is stored in the mr table. As well as the previously discussed attributes of author, log, date, and time, the mr table includes two booleans, trunkchanges and classchanges. The value for trunkchanges indicates whether the MR made any changes to the trunk, or whether it only affected a branch. The value for classchanges indicates whether the MR affected any source code files. These values can be useful in excluding MRs from a desired change-set.

The MR System State can be retrieved from the classstate and packagestate tables. These record what classes and packages exist at a particular MR. The actual data describing each class and package are stored in the packagerev and classrev tables. This data is only updated when the package or class is changed. Currently, the state of methods and fields is stored within the classrev table.

The MR Impact is stored in the packagechange, classchange, methodchange and fieldchange tables. The changes to the model are stored as either added, modified, or deleted. Changes to sub-entities are reflected in the changes stored about parent entities. So, for example, if a class is modified then that class's package will also be considered to have been modified. So as not to lose the artifact-level information, we also store the information about the changes to files.

The relationship information that is part of the MR System State is stored in the entity that is the source of the relationship. The relationship information that is part of the MR Impact is stored in the relationshipchange table. Unlike entities, which can be added, modified, or deleted, relationship can only be added or deleted.

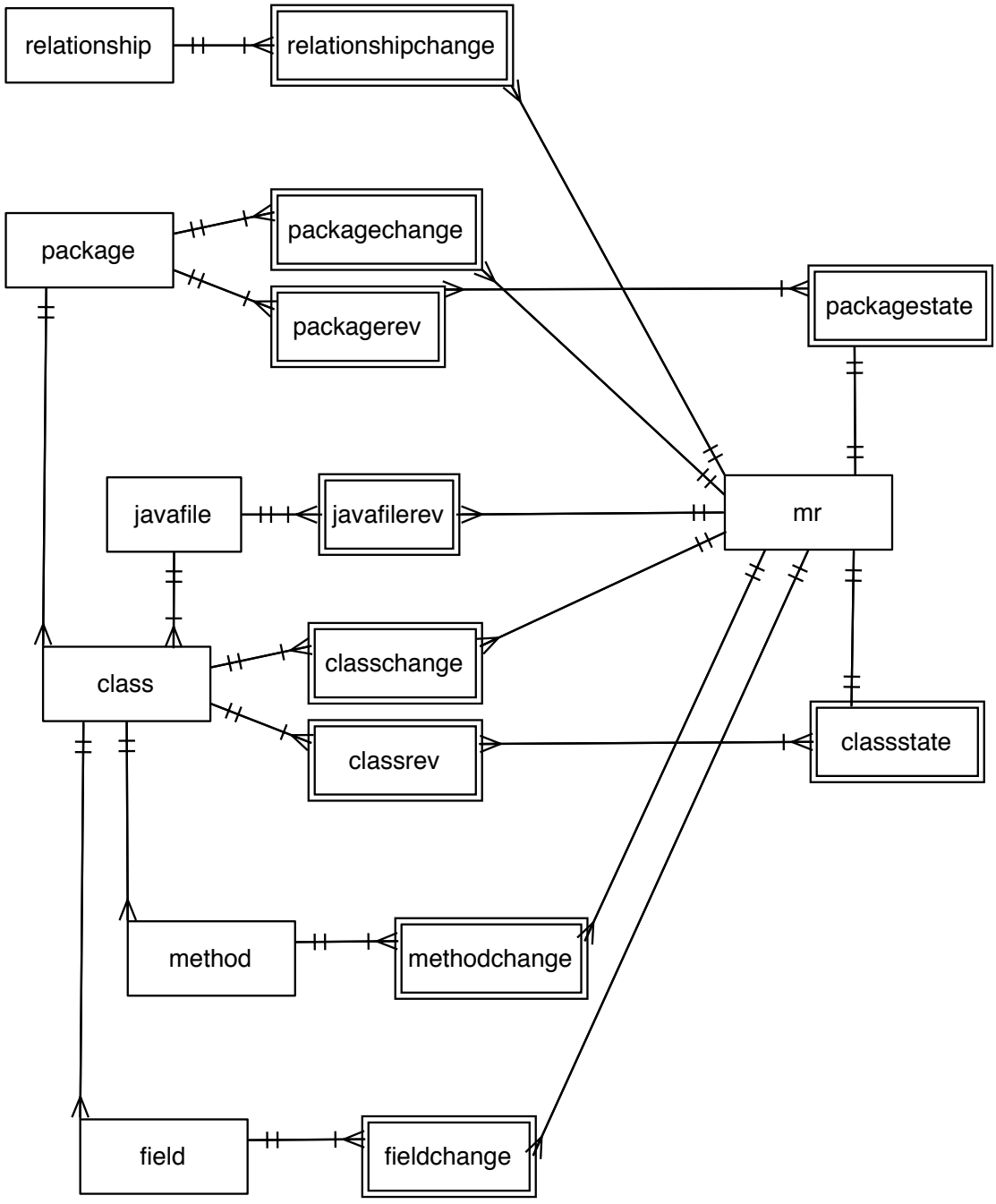


Figure 4.2. *An ER diagram showing the key entities of the visualization database*

4.2 Motive

We have developed a prototype tool allows a user to select a change-set to examine from our visualization database, and then view the impact of the change-set using architectural impact views. We named our tool *Motive*, as one of our main goals is to provide greater understanding of the reasons for architectural drift and evolution.

4.2.1 Change-set Selection

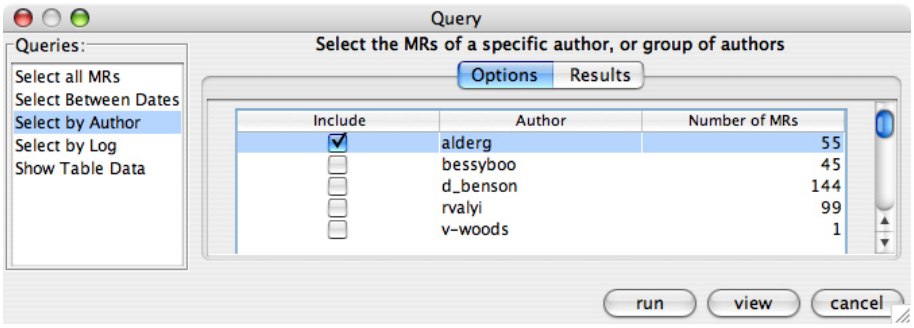


Figure 4.3. *The Query Dialog. The available queries are shown on the left side. The right side of the dialog shows query-specific options the user can fill out.*

As discussed in the previous chapter, a change-set can be built by enumeration or by using a query to select MRs which satisfy certain properties. It is possible for a user to directly retrieve a desired change-set by using an SQL query. However, this requires the user be familiar with SQL, and with our database schema. Motive's Query Dialog, shown in Figure 4.3, allows an expert to create and store a reusable query, which end-users can then run to select a change-set.

The left side of the Query Dialog shows what queries are available for a user to run. In Figure 4.3, the user has chosen "Select by Author." Once a query has been selected, a short title is displayed that indicates the purpose of the query, and the Options Panel is filled. The Options Panel is a query-specific user interface that allows the user to customize specific attributes of the query. In this case, it allows the user to choose which authors' MRs they

are interested in. The Options Panel may also need to perform SQL statements to build its interface; for example, "Select by Author" needs to perform SQL statements to discover what authors are in the visualization database, and count the number of MRs made by each author.

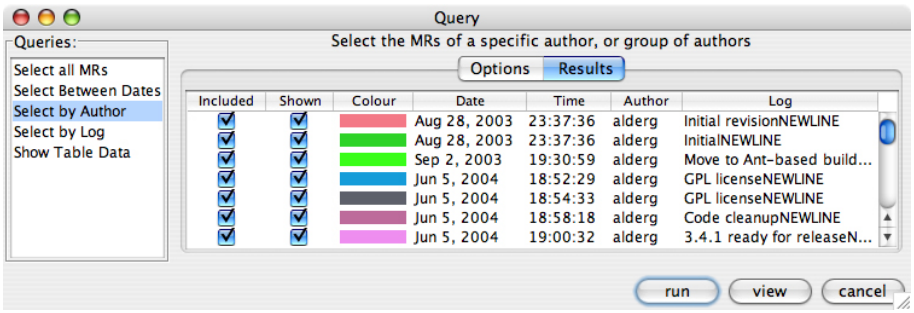


Figure 4.4. *The Query Dialog Results Panel. This shows the MRs returned by the query, and allows the user to perform some selective filtering of what MRs should be part of the change-set.*

Once the options have been filled out, the user presses the "run" button to run the query. This performs a query-specific SQL call to the database. Most queries, including the "Select by Author" query are focused on returning a list of MRs from which a change-set can be made. Figure 4.4 shows the results of such a query. The table displays each MR's associated date, time, author, and log. As well, each MR has a value for *Included*, which indicates whether the MR is part of the change-set, and *Shown* and *Color* which are related to the MR's appearance on the Temporal Slider (discussed in Section 4.2.3). Once the user has finished modifying these values, the "view" button is used to view the impact of the change-set.

Queries can also be created to display other types of information about the database. For example, Figure 4.5 shows the results of the "Show Table Data" query. This can be used to discover information about the system that may be more efficiently displayed in table form rather than in a diagram. Figure 4.5 shows, for each package, how many MRs the package was involved in, and how many classes in each package have been added,

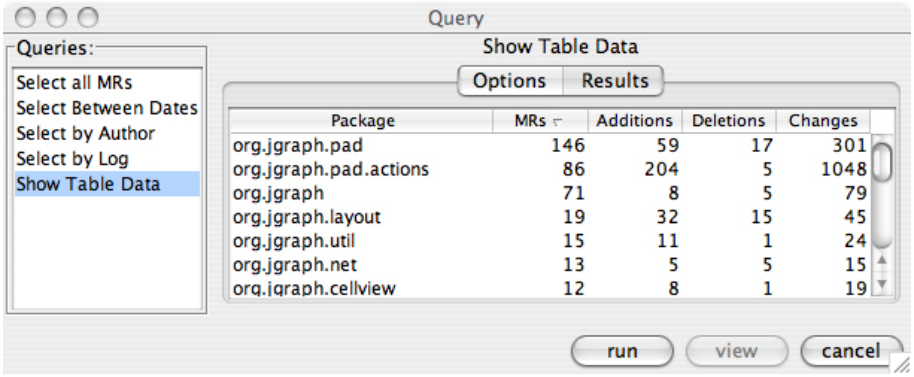


Figure 4.5. *The results of the 'Show Table Data' Query. In this case the returned results are not a list of MRs that can be used to make a change-set.*

deleted, or modified. Because the "Show Table Data" query is not designed to show the impact of a change-set, the "view" button is disabled.

The five queries shown in the Query Dialog are sample queries developed to begin to explore the usefulness of change-sets. To provide a point of extensibility to Motive, we allow users to create new queries by extending the abstract QueryView class. The key methods of the class are shown below:

1. QueryView(String name, String description, boolean canView)

   The constructor. A QueryView has a name, which describes it in the list of queries, a description, which gives a user more insight into what the query is intended to do, and a boolean which specifies whether or not the query is designed to create a change-set.

2. JPanel getQueryViewGUI()

   Called to get the user interface of the Options panel. This may result in some SQL calls being made to the database. The method returns a JPanel, meaning that it can contain almost any user interface element, such as buttons, drop down lists, etc.

3. JTable getQueryResultsGUI()

   Called to get the table that contains the results of the query. If the query is designed to create a change-set the format of the table will be the same as in Figure 4.4. If

not, any other type of information might be represented.

4.2.2 Viewing the Impact of a Change-set

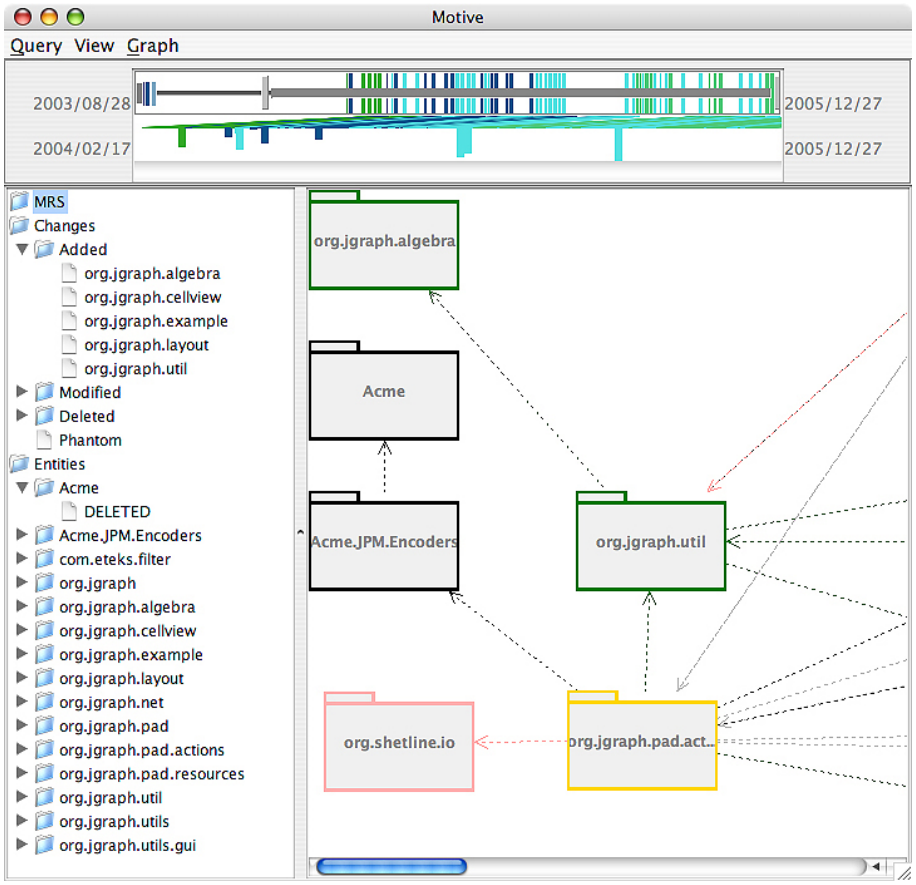


Figure 4.6. *A screenshot showing the three main panels of Motive. The Temporal Slider is at the top of the tool, the Hierarchical Summary is shown on the left, and the Graph View is shown on the right.*

Figure 4.6 shows an overview of the three main panels of the Motive tool:

1. Temporal Slider.

   This shows the MRs in the change-set, and allows the user to quickly adjust the period under consideration by dragging the ends of the slider.

2. Hierarchical Summary.

   This shows a textural view of the details of the MRs in the change-set, including each MR's date, author, and list of entities it affected, and the net effect of all the MRs in the change-set on each architectural entity.

3. Graph View.

   This shows the net effect of the MRs in the change-set on the architecture of the system using the architectural impact views described in the previous chapter. Currently these views are limited to UML class and package diagrams.

4.2.3 Temporal Slider

The Temporal Slider shows the MRs in the change-set and allows the user to quickly adjust the period under display by dragging either end of the slider. The slider itself consists of two halves. The upper half shows all the MRs in the change-set, colored according to the author of each MR. On the bottom half we highlight MRs that stand out from the other MRs in the change-set. We plan to allow the user to select different metrics by which "stand out" can be defined; currently we show the top 10 MRs, of those currently under consideration, that have modified the most files.

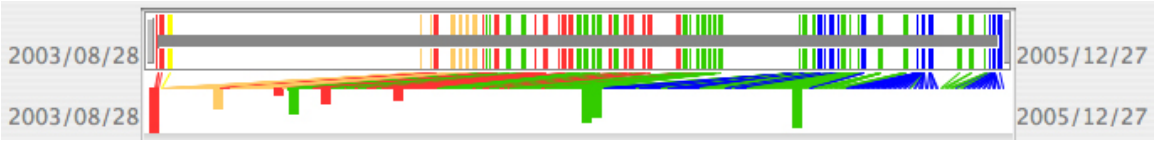


Figure 4.7. *A Temporal Slider showing all changes in the system. Time is shown on the x-axis. MRs are shown as colored rectangles, with the color of the rectangle indicating the author of the MR. The position of the MR on the top half of the slider indicates the time the MR occurred. The bottom half of the slider highlights the 10 MRs that have modified the most files, with the height indicating where the MR falls in the top 10 (taller MRs have modified more files).*

An example of a Temporal Slider is shown in Figure 4.7. Here, a query was used to

select all MRs to the system. The slider is displaying all these changes, and so the dates on both halves of the slider indicate that the period under consideration goes from 2003/08/28 to 2005/12/27.

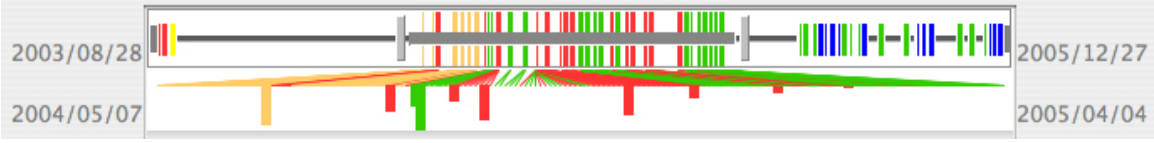


Figure 4.8. *A Temporal Slider showing changes that occurred in the middle of the system's lifetime. The user has dragged the ends of the slider to modify the change-set under consideration.*

The ends of a Temporal Slider can be dragged to change the period under consideration. In Figure 4.8, the period has been changed to show the changes that occurred in the middle of the software system's lifetime. A comparison to Figure 4.7 shows the dates on the top half have not changed, this is because the change-set was not built from a new query. Similarly, all MRs that occurred to the system over its lifetime are shown, though MRs that are not part of the change-set are shown in reduced size. However, the dates on the lower half have changed to indicate the period of MRs in the new change-set. As well, a new set of 10 MRs has been highlighted on the lower half.

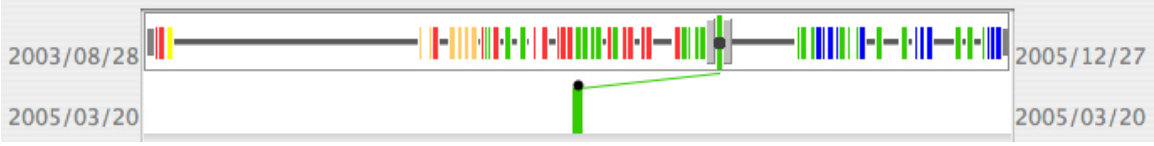


Figure 4.9. *A Temporal Slider showing the selection of one MR.*

It may also be that a user wants to focus in on a particular MR. By dragging the two ends against each other, they become "locked", indicated by the grey circle between them. In this mode, one MR at a time can be selected. For example, Figure 4.9 shows the highlighting of an MR that occurred on 2005/03/20.

The bottom half of the slider in Figure 4.9 may seem redundant. As only one MR

is selected, it seems obvious that only that MR exists in the change-set, and, further, that the MR selected will have had the most change impact on the change-set. However, even when the ends are "locked" we are still able to provide some useful information on the bottom half of the slider. First, more than one MR might be shown in the lower half. The "locked" ends may cover more than one MR that happened within the same short period. The dot above the MR in the bottom half indicates which MR is currently making up the change-set, this can be dragged to choose different MRs.

As well, even though only one MR is part of the change-set it does not necessarily mean it will be shown as having the maximum amount of change. It is possible for the user to include MRs in the change-set which either made changes exclusively to non-source files or to a non-trunk branch. In both cases such changes will be shown with an empty rectangle on the bottom half of the slider, to indicate that if the MR is selected Motive will not display any change.

4.2.4 Hierarchical Summary

Figure 4.10 shows a highlight of a Hierarchical Summary of the effect of a change-set that includes only the initial MR. The Hierarchical Summary is a tree view composed of three branches: the MRs branch, the Changes branch, and the Entities branch. The MRs branch shows a view of all the MRs in the change-set, as well as summarizing what entities were added, modified, or deleted in each MR. This enables a user interested in the effect of a particular MR to learn more about it without needing to create a new change-set. In Figure 4.10, as the only MR in the change-set is the initial MR, the only changes to entities are the addition of the new packages.

The Changes Branch shows a summary of what changes have occurred to what entities. In Figure 4.10, the only node with any content is the Added node, which, if expanded, would show which packages have been added. In this situation, where the change-set only contains one MR, the Changes Branch is unnecessary. However, a change-set may be composed of hundreds or thousands of MRs, and so it would in general not be practical

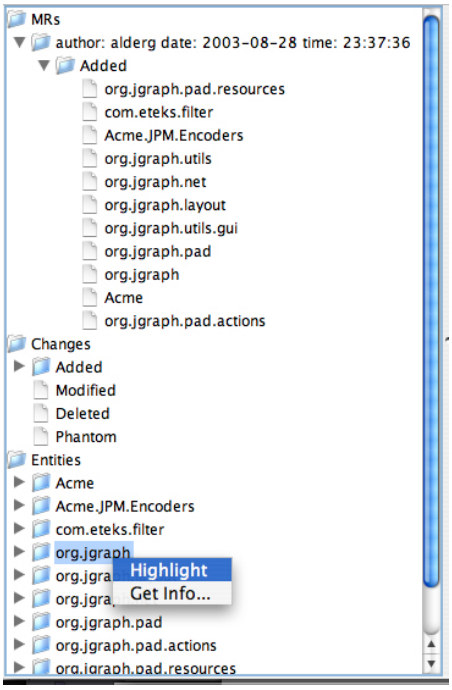


Figure 4.10. *A Hierarchical Summary of a change-set including only the initial MR. The user can see how the change-set has modified the system by starting from the MRs in the change-set, the changes that occurred to the system, or the entities and relationships in the system.*

to use the MRs branch to discover how entities have changed. Information displayed in the Changes Branch, can also be discovered from the architectural impact diagrams; however, the Changes Branch organizes entities by types of change, and so may prove more convenient in some situations.

The Entities Branch stores the same information as the changes branch, though it is organized by entity, not by change. This view might be useful when the user is interested in the changes to a particular entity. The Changes and Entities branches can also aid with navigation. Right-clicking on a particular entity gives the user the options to "Highlight" the entity, which scrolls the Graph View to show the current entity, and to "Get Info", which shows an Information Dialog.

Figure 4.11 shows an example Information Dialog showing what MRs have affected

the org.jgraph package. The dialog shows information about all MRs, not just those in the change-set, with the value for "Included" indicating what MRs are part of the current change-set. The Information Dialog can be used to filter what MRs are in the change-set by changing this value.

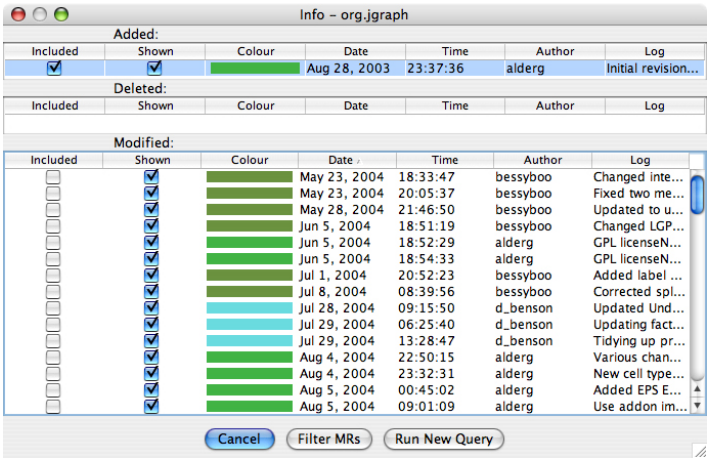


Figure 4.11. *An Information Dialog showing the MRs in the change-set that affected the org.jgraph package*

The Hierarchical Summary is designed to show a textural view of the same information displayed in the current Graph View. Figure 4.12 shows a textural representation of a class diagram displaying the changes made in the initial MR to the org.jgraph package. In the MRs and Changes branches, changes are now expressed in terms of their effect on classes and interfaces. The Entities Branch includes this information, as well as showing the associated fields and methods. The user can use the "..." node at the top of the view to navigate from the class diagram to a package diagram.

4.2.5 Graph View

The Graph View is the main view of Motive, showing the impact of a change-set using architectural impact views. These views can be package diagrams, as shown in Figure 4.6, or class diagrams, as shown in Figure 4.13. The class diagram shows the impact of

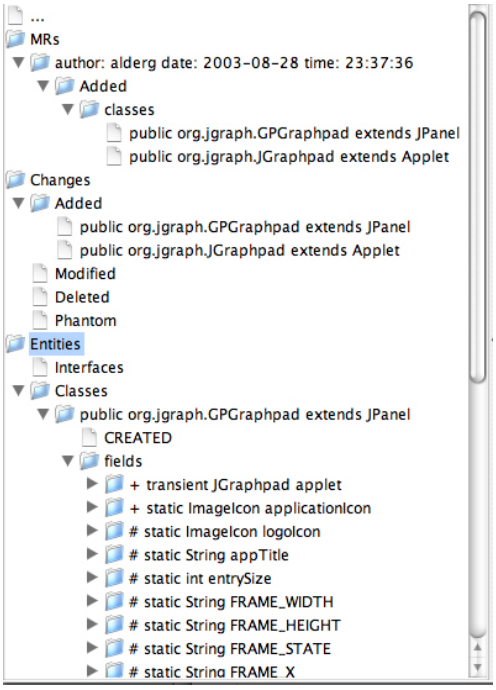


Figure 4.12. *A Hierarchical Summary of the changes made in the initial MR to the org.jgraph package*

the initial MR on the org.jgraph.net package. The GraphNetworkModel class is dependent on GraphNetworkModelPeer and GraphCellIdentityMap, and implements GraphNetwork-ModelListener. GraphNetworkModelPeer is dependent on GraphNetworkModel.

To conserve space, the class diagram in Figure 4.6 does not include information about how the fields and methods in the classes have changed. The user can choose to display this information, as shown in Figure 4.14. Here, all the classes are in the added state, shown by their green color. That each field and method has been added is implied by their being contained in an added class.

When the state of a field or method cannot be determined directly by a user from the state of the enclosing class, the state of the field or method is indicated using the same coloring scheme we use for other entities. For example, Figure 4.15 show the changes to the org.jgraph.net package from 2004/05/08 to 2005/03/30. Here all the classes are in the modified state, as shown by their yellow color. However, this does not imply anything

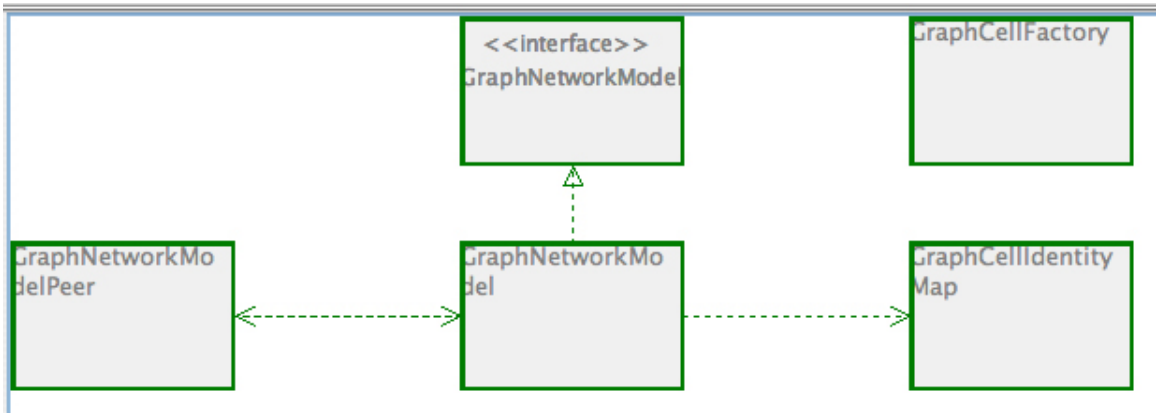


Figure 4.13. *A class diagram showing the impact of the initial MR on the org.jgraph.net package. All nodes are relationships are colored green, indicating that they were all added in the change-set.*

about the state of a particular method or field. In the diagram some methods and fields are unchanged, and colored grey, while others are modified, and colored yellow.

4.2.5.1 Background Coloring

An architectural impact view indicates change by means of color. For example, in Figure 4.13 all the nodes and relationships are colored green, to indicate that they have been added. However, because the nodes are colored only along their borders, this leaves the grey background of the node not displaying any useful information. We used this space to implement the second Motive extension point, background coloring.

Background coloring is used to indicate further details about how the change-set has affected the nodes in the class view. An example of this approach is shown in Figure 4.16, which displays the change to the org.jgraph.net package from 2004/05/08 to 2005/03/30. The yellow border colors indicate that every class has been modified. The grey relationships indicate that no relationship has been added or deleted. The class diagram's background coloring has been chosen so as to show which author made the most changes to each class. Four of the classes were most changed by the blue author, with GraphCellFactory modified

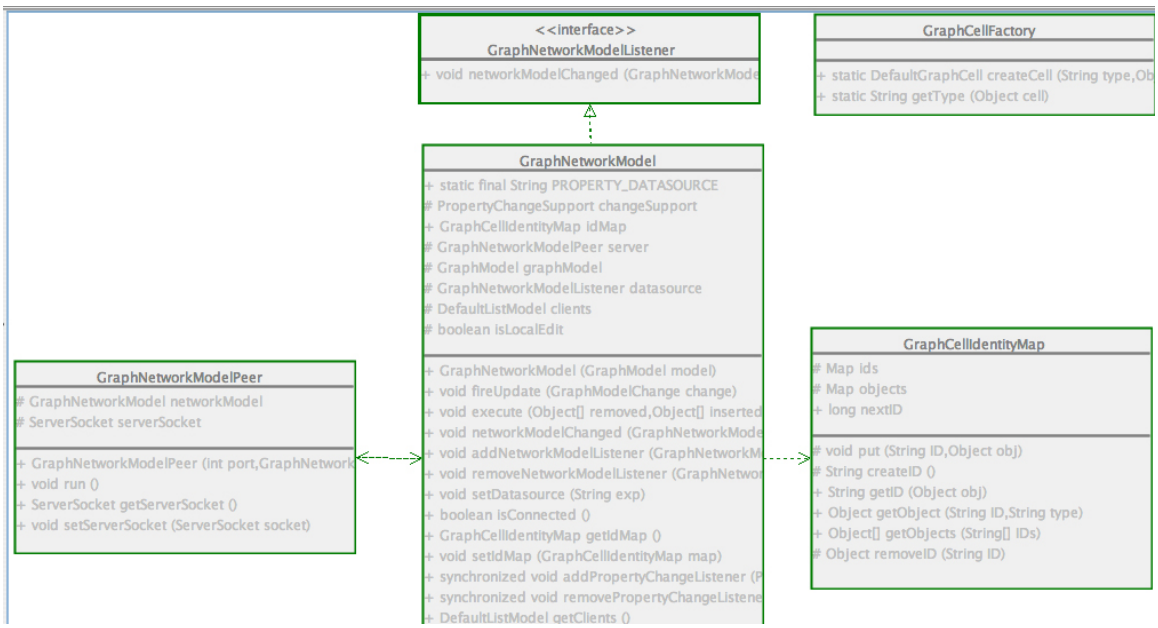


Figure 4.14. *A detailed class diagram showing the impact of the initial MR on the org.jgraph.net package, including what fields and methods exist in the classes. Every node and relationship is shown as green, indicating it was added. The fields and methods belonging to an added class must also be added, so to make the diagram more readable we do not color all fields and methods as green.*

most by the green author. The color mapping is the same as that the TemporalSlider uses to color MRs according to author, and can be examined to find the names of the corresponding authors.

Currently we have implemented five methods of background coloring:

- Amount Modified. This indicates through a greyscale coloring how much each node has been affected by the MRs in the change-set. Darker nodes were modified in more MRs, lighter nodes in less. The range of color is dependent on a user-defined number of MRs specified as the maximum; nodes with the maximum value of MRs are colored black.
- Author Most Modified. This is the background coloring used in Figure 4.16. Nodes are colored according to which author has made the most MRs in the change-set that

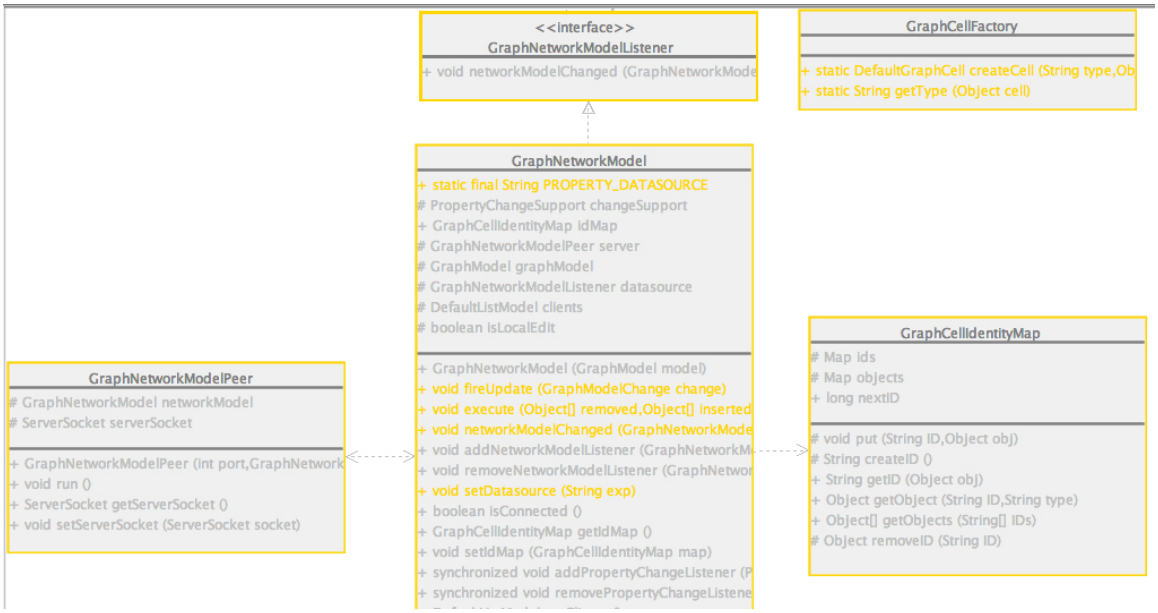


Figure 4.15. *A detailed class diagram showing the changes to the org.jgraph.net package from 2004/05/08 - 2005/03/30. All nodes are shown as yellow, indicating they were modified. Unchanged fields/methods are shown in grey, and modified fields/methods shown in yellow.*

have affected the node. The color mapping is the same as that in the Temporal Slider.

- Default. The background is grey and does not indicate any information.
- Distinct Authors Modified. This indicates through a greyscale coloring how many distinct authors have made MRs in the change-set that have affected the node. Darker nodes were changed by more authors, lighter nodes by less. The range of color is dependent on a user-defined number of authors specified as the maximum.
- Last Modified. This indicates through a greyscale coloring when the last MR in the change-set that affected the node was made. Darker nodes were affected more recently, lighter nodes further in the past. The range is dependent either on a user-defined date or number of MRs specified as the maximum. For example, the user could specify that nodes last modified on or before 2004/05/08 are colored black, or nodes that haven't been modified in 100 MRs are colored black.

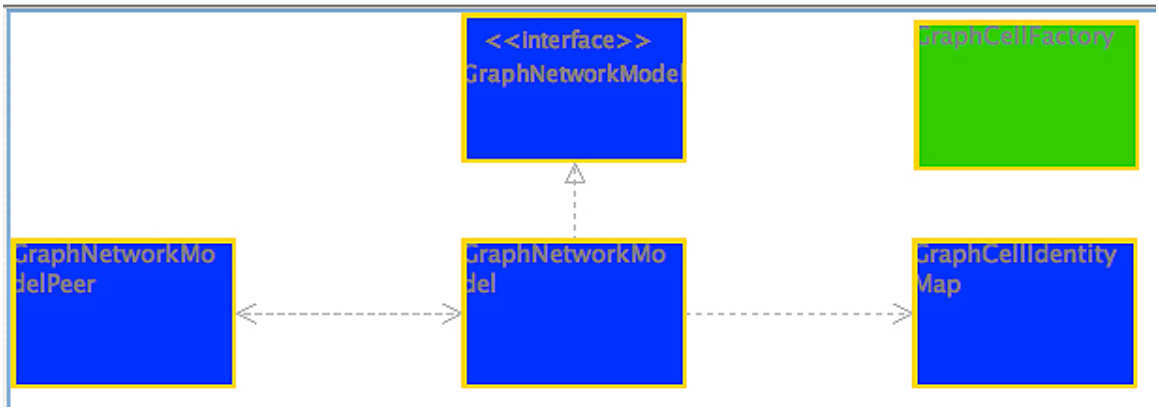


Figure 4.16. *A class diagram showing the change from 2004/05/08 - 2005/03/30 to the org.jgraph.net package, using a background coloring according to the author that has made the most changes to each node. As with the other architecture impact views, the border color of the node and the color of the relationship describes how the node/relationship was modified in the change-set.*

Users interested in creating their own method of background coloring extend the abstract BackgroundColoring class. The key methods of the class are shown below:

- BackgroundColoring(String name)

  The constructor. A BackgroundColoring has a name which describes what it does.

- JPanel getUserInterface()

  Called to get a user interface that allows the user to change query properties. For example, the Amount Modified coloring allows the user to set a value for the maximum number of MRs, after which the node is colored black.

- Color getBackgroundColor(List annotationList)

  Called to get a background color for a particular node according to the strategy of the current BackgroundColoring class. It passes in the annotation list, which contains all the MRs in the total-set that have affected the node, as well as indicating whether or not an MR is also included in the change-set.

4.2.6 Text View

To further help a user understand the evolution of a particular class, we have created a Text View that shows how a change-set has affected a particular file. A class diagram may indicate, for example, that a method in a class has changed. The Text View enables the user to see how the method has changed.

As well, a Text View can be used to show more detail about types of changes that are not well-represented in the Graph View. For example, Figure 4.15, showing the changes to the org.jgraph.net package from 2004/05/08 to 2005/03/30, indicates that the GraphNetworkModelListener interface has been modified. However, the only method in the interface, networkModelChanged(), is shown as not being modified. This raises the question of what could have been modified in the interface.

The answer is provided by the Text View, as shown in Figure 4.17. This view is opened by double-clicking on a class or interface. As the figure shows, two of the three main panels of Motive remain unchanged in the Text View. The Temporal Slider remains, showing the MRs in the change-set, and indicating which MRs in the change-set have had the most impact to the entities under consideration. From the Temporal Slider it can be seen that only one MR has affected the GraphNetworkModelListener.java file. The Hierarchical Summary view also remains, showing the impact to the fields and methods of the current class or interface.

However, the Graph View has been replaced with the Text View. The lower portion of the Text View shows the text of the earliest file revision in the change-set on the left, and the latest on the right. In this case, as only one MR has affected the file the two revisions are consecutive.

The top pane of the Text View displays the output of the *diff* command [12]. A challenge when creating the Text View was being consistent with the other views by highlighting the impact of MRs in the change-set and not showing the effect of the other MRs. The solution is that for each MR in the change-set that has affected the selected file, the diff command is run to compare that file revision with the previous revision, even if the previ-

ous revision belongs to an MR not in the change-set. The output of diff is appended to the text in the top pane. In the case of Figure 4.17, as only one MR in the change-set affected the file, only one diff output is shown.

4.2.7 Summary

This chapter described how we implement our model and method for showing the impact of a change-set. Our implementation is currently limited to one SCM system (CVS) and one programming language (Java). The first section of the chapter describes the preprocessing that we do to a CVS repository to transform information about the MRs into a database which better facilitates creating and visualizing change-sets. Our preprocessing phase begins by using softChange [15] to recover the logical transactions from CVS and store the result in a softChange database. From the softChange database, we extract the state of the software system at every MR using QDox [33], an open-source Java scanner, and record the state at every MR in our database. Each MR's state is compared to the state of the previous MR to discover and record the architectural effect of the MR.

This chapter also described Motive, a prototype tool we developed that uses our visualization database to allow a user to select and then view the effect of a change-set. Change-sets are selected by running a query that a developer has written using our query plugin interface. Motive displays the impact of a change-set using architectural impact view built on top of standard UML class and package diagrams, with the impact of the change-set indicated by annotating the UML diagrams with color.

Motive integrates some features that assist architectural impact views in summarizing what has happened in a change-set. The Temporal Slider shows all the MRs in the change-set, and includes metric views that highlight some of the most interesting MRs within a change-set. If the user is interested in the details of a specific file, a text view build using diff [12] can show the evolution of the source code over the change-set.

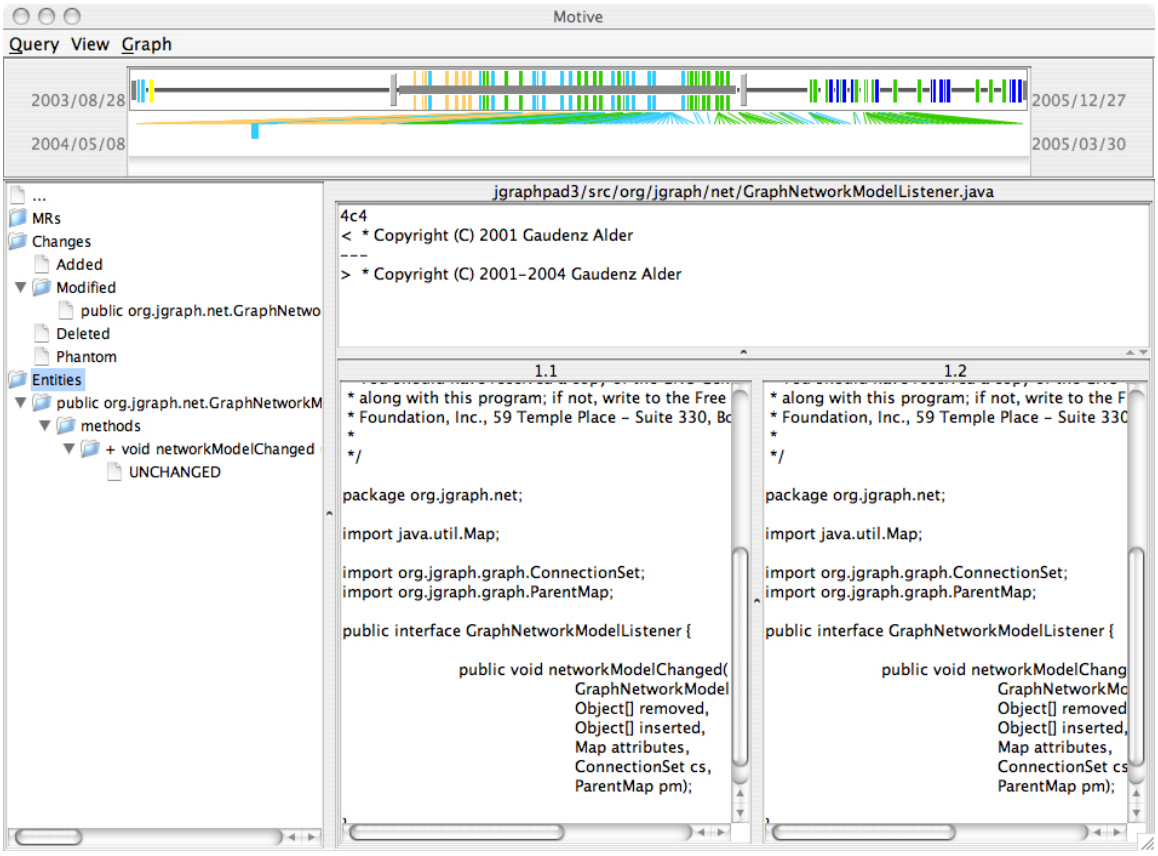


Figure 4.17. *A Text View showing the changes from 2004/05/08 - 2005/03/30 to the org.jgraph.net.GraphNetworkModelListener interface. The lower portion of the Text View shows the text of the earliest file revision in the change-set on the left, and the latest on the right. The upper portion shows the output of running diff to compare each file revision in the change-set with the previous revision.*

Chapter 5

Evaluation

Our visualization method and Motive, the prototype tool that implements our visualization, were developed to summarize information about a software system's evolution for three of the groups most interested in software evolution: researchers, developers, and managers. In order to evaluate whether this approach was useful to helping to meet the needs of these three groups, we conducted two case studies. In each case study we asked the same questions about the evolution of the software system, then tried to answer these questions with Motive. This chapter first presents the questions that used and the systems studied. The chapter continues by describing the results of the two case studies. The chapter concludes with a discussion section in which we examine the advantages and limitations of our approach.

5.1 Evaluation Questions

The questions asked were drawn from the motivations of researchers, developers, and managers. The questions were largely created based on plausible use cases rather than empirical data. A notable exception is the collection of scenarios developed by Wu [45] that are based on a survey Wu et al. conducted about the concerns of developers and managers [46].

Researcher Questions

1. What packages are highly coupled?

   Tonu, Ashkan, and Tahvildari [40] introduced the terms *evolution-critical* and *evolution-sensitive* to describe highly coupled modules, with modules where the coupling seems logical considered evolution-sensitive and modules where the coupling seems to be as a result of poor programming practice or design considered evolution-critical. Evolution-critical modules in particular may need to be evolved; however, determining if coupling is logical requires a good understanding of the conceptual architecture of the system. We focus on identifying the highly coupled packages, after which other methods could be used to evaluate the reason for the coupling. We consider a package highly coupled if it contains a high number of packages dependent on it, relative to the other packages in the system.

2. What packages were frequently modified in the past?

   Identifying the packages that were most prone to change in the past indicates what parts of the architecture were most affected by software evolution. We consider a package frequently modified if it has been modified a large number of times relative to the other packages in the system.

3. What were the architecturally disruptive changes in the past?

   A well-designed architecture should confine the concepts that are prone to change [30]. Architecturally disruptive changes are changes that were not confined to a few concepts, and so were likely not anticipated by the original developers. To identify these changes, we considered the term "architecturally disruptive" from a number of different perspectives, including the top 10 changes in regards to the number of files modified, packages changed, existing packages changed (meaning we are only counting a package as changed in the MR if the change was not an addition), and dependencies added.

4. What packages have been modified by a broad group of developers?

One of the goals in designing a system is to produce a maintainable architecture. There may be a need for some highly specialized sections of code only modifiable by particular developers, but these should be kept to a minimum. Our goal in identifying modules that have been modified by a broad group of developers is to gain some evidence about what modules seem to be both understandable and modifiable.

5. Given a keyword, what changes used it in their commit log and how did they affect the architecture?

   As described by Chen et al. [4], CVS log comments often indicate both the purpose of a change and, indirectly, the purpose of code. Each system will have its own repeating keywords in the log comments that correspond to categories of changes and features of the system. By visualizing groups of changes based on keywords in the log entries we believe we can gain an indication of the maintenance effort that different types of changes and different features require.

   For example, Chen et al. observe that the CVS log "added footnote feature" indicates that the corresponding change is related to footnotes and so it would be expected that source code modified in the corresponding MR is related to the footnote feature. Viewing all the changes with the keyword "footnote" indicates both the architectural entities that are related to footnotes, and how much effort it has taken to maintain the footnote feature.

Developer Questions

6. When did a particular architectural entity appear?

   Voinea, Telea, and van Wijk [42] observed that identifying the context in which a piece of code appeared is an important use case for a software maintainer. We consider the context of the entity appearing the other entities and relationships that were changed as part of the addition MR, and the other details of the MR, in particular its log comment.

7. Who has made the most modifications to a package, and who made the last modification?

   Wu [45] noted the importance of knowing who is making changes to which part of the software. For example, a developer making changes to a package she does not have a lot of familiarity with may wish to consult another, more knowledgeable, developer for help. It is likely that the most suitable developer to ask for help is the developer who has made the most modifications to the package or the developer who last modified it.

8. What has been changed in the last week globally or in a specific package?

   It is often the case, as reported by Wu [45], that a particular developer will be inactive for a small period of time, such as when she goes on a vacation. When the developer becomes active again, she will want to know what has changed in the project during the time she was away. As well, if the developer primarily works in a specific package, what has happened within that package will be of particular interest.

9. What happened in a particular commit?

   Wu et al. [46] found that developers will often want to examine a particular commit. Arguably, this is the most common use case for a developer trying to understand the evolution of a software system.

Manager Questions

10. What packages have developers recently modified?

    A manager interested in the progress developers are making towards current goals may be interested in the packages developers are currently working on. As well, Girba, Ducasse, and Lanza [19] found some evidence that classes changed recently are likely to be changed in the future, so this information could help a manager trying to schedule future work. We identify the recently modified packages by finding what developers have been active in the last 2 weeks, and, of those developers, what

packages have they modified in their last 5 MRs.

11. What packages have not been modified in the recent past?

    Packages that have not been modified a lot in the recent past may be stable architectural entities that will not require a lot of maintenance effort in the near future, or dead code that should be removed. We consider the recent past as being a user-defined concept relative to the MRs in the project (for example, the recent past can be defined as the last 10 MRs).

12. How productive has a particular developer been?

    Numerous researchers have noted how data in a version control system could help monitor development progress [17] [45] [39]. Productivity can be measured in many ways, such as by the number of MRs, or by the amount of architectural entities modified.

13. How much change was there between the previous two releases?

    The amount of effort involved in readying the system for previous releases may help with future planning. This requires identifying when previous releases of the system occurred, and measuring the changes that happened between those releases in terms of the packages and classes affected.

5.2 Systems Studied

Motive was used to answer the previous questions during the study of two systems: *JGraphpad* [23] and *SQuirreL SQL* [35].

JGraphpad is an open source diagram editor included with JGraph. JGraphpad is a small software system, consisting of 357 classes and 16 packages. We studied its evolution over 344 MRs that were spread over more than 2 years, from August 2003 to December 2005.

SQuirreL SQL is an open source graphical Java program designed to visualize the structure of, and interact with, any java database connectivity (JDBC) compliant database.

SQuirreL SQL is a medium-sized system, with over 1500 classes and 150 packages. We examined its evolution over roughly 4 years, from December 2002 to January 2007. Over those 4 years we identified more than 6000 MRs. However, due to a problem with how we extracted the MR information using softChange, in some cases individual changes with the same log and timestamp were not properly combined into the same MR. This did not affect how we answered the questions or the operation of the tool, but it did increase the numerical value of the MRs in some of our answers.

5.3 JGraphpad

5.3.1 What packages are highly coupled?

Figure 5.1 shows an overview of how JGraphpad has evolved over all the MRs in the system. As all MRs are included in the change-set, the annotation list of each entity/relationship will start with the MR in which the entity/relationship was added, and so every entity/relationship will be displayed as being in either the added or phantom states.

Figure 5.1 shows 16 packages in the system during the period studied, with 11 packages still existing in the system as of the last MR. The figure also indicates the most coupled modules. As shown in Figure 5.2, the packages with the most incoming dependencies are org.jgraph and org.jgraph.pad.resources.

5.3.2 What packages were frequently modified in the past?

Figure 5.3 shows an overview of what has been most modified in JGraphpad. The visualization used shades nodes so that the darker the node is, the more times it has been modified. The visualization allows the user to specify a threshold - in this case a package shaded black has been modified over 100 times. From the figure, the most modified packages are org.jgraph.pad, org.jgraph, and org.jgraph.pad.actions.

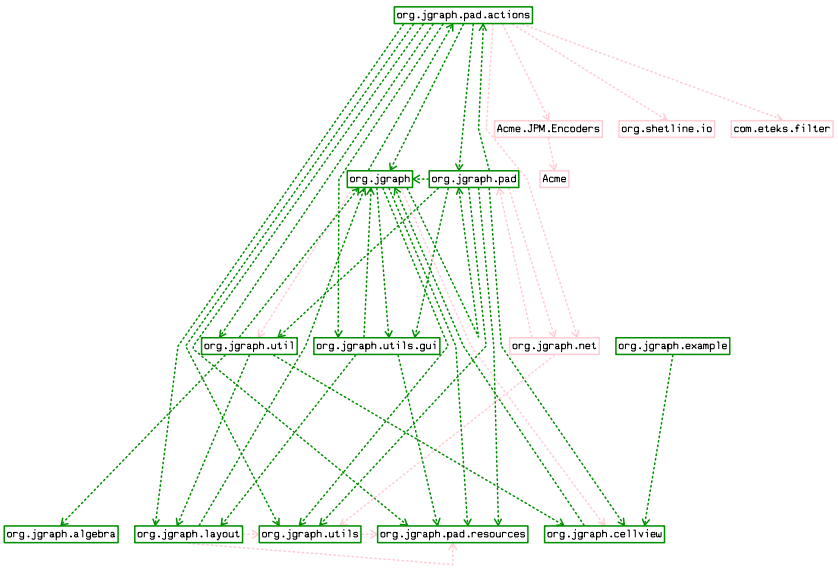


Figure 5.1. *A package diagram showing an overview of how JGraphpad has evolved. The relationships show dependencies between the packages.*

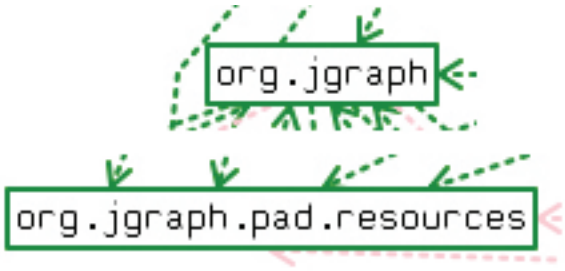


Figure 5.2. *The packages with the most dependencies in JGraphpad*

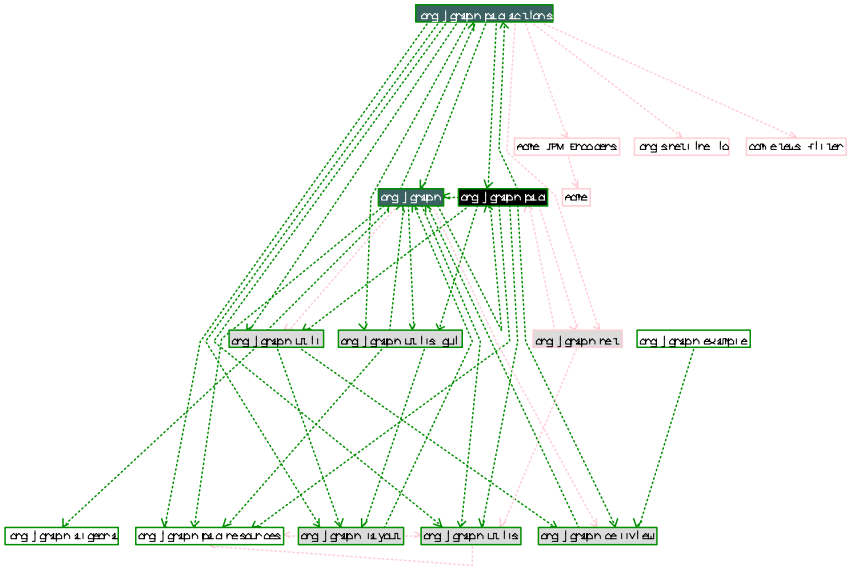


Figure 5.3. *An overview of JGraphpad shaded according to the number of modifications. The darker a node, the more MRs that modified it.*

5.3.3 What were the architecturally disruptive changes in the past?

Figure 5.4 shows the Temporal Slider summarizing a change-set with all the MRs of JGraphpad, with the bottom half highlighting the 10 changes that have modified the most files. The coloring of an MR indicates the developer who made the change, and the size of its highlight indicates how many files it has modified. For example, it can be seen that the most change occurred at the beginning of the project (the initial project creation). As this change is colored teal, it can be derived from the coloring mapping (not shown) that the

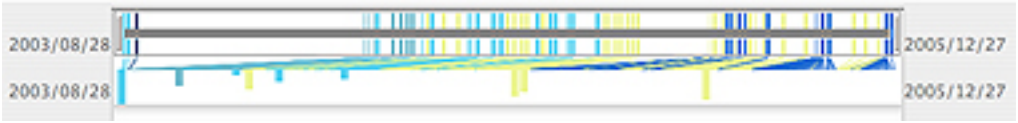


Figure 5.4. *The Temporal Slider for the change-set of all MRs of JGraphpad. The bottom half of the slider highlights the 10 changes that have modified the most files, with the height of the 10 MRs indicating where it falls in the top 10 list (taller MRs modified more files). The color of the MR indicates what author made the change.*

change was made by alderg.

However, the number of files modified was only one of the criteria selected for identifying architecturally disruptive changes. Table 5.1 shows changes that met at least one of these criteria, being in the top 10 changes in regards to the number of: files modified (FM), packages modified (PM), existing packages modified (EPM), or dependencies added (DA). The broad criteria lead to the identification of what are clearly some false positives, for example MR 46 which changed the copyright text at the top of the source files. That MR is an example of a widespread change that affects almost every file in the project, but has no effect on the architecture of the system.

5.3.4 What packages have been modified by a broad group of developers?

Figure 5.5 shows an overview of what are the most broadly modified packages in JGraphpad. The visualization used shades nodes so that the darker the node is, the more authors have modified it. The visualization allows the user to specify a threshold - in this case a package shaded black has been modified by at least 5 authors. The visualization shows the most broadly modified packages are org.jgraph.pad, org.jgraph, and org.jgraph.pad.actions.

5.3.5 Given a keyword, what changes used it in their commit log and how did they affect the architecture?

During our investigation, we found that a number of the MRs made reference to keeping up-to-date with the JGraph library, for example, “part of JGraph 4.0 port” and “compiles with JGraph 5.4”. JGraphpad is built using the JGraph library and we would like to see the extent of the evolutionary coupling between JGraphpad and JGraph. A change-set was created by searching for MRs whose log included the phrase “JGraph”. This does not include some changes that seem to be related to JGraph being updated, such as “updated files to use new applyLayout() signature”, but it gives an indication of how widespread the

Table 5.1. *Potential architecturally disruptive changes to JGraphpad*

MR	FM	PM	EPM	DA	Log
58	222	6	6	0	Tidying up project
343	199	1	1	0	...removed [unused] constructors, optimized imports
46	179	1	1	0	Changed copyrights
261	123	9	9	2	Removed some Javadocs warnings
80	77	2	2	0	Changed AttributeMap to GPAttributeMap
84	69	1	1	0	Removed GPAttributeMap
176	47	8	3	10	Moved addons into JGraphpad
33	40	2	2	0	Part of JGraph 4.0 port
34	35	1	1	0	Further parts of JGraph 4.0 port
180	34	7	7	0	Updated CVS from distribution
27	30	5	5	0	Changed LGPL license headers to GPL
144	13	4	4	0	Compiles with JGraph 5.4
71	9	5	5	0	New splash, progress dialog...
265	9	4	4	0	Removed unused parameters
253	8	4	4	0	...removed broken network features...
65	6	4	4	3	Various changes, see ChangeLog
100	6	5	5	0	Added export functions, fixed drag and drop...
235	6	4	4	0	Tidied up some warning and removed unused method
281	3	1	1	1	...objects have been moved...
129	2	2	2	2	Updated files to use new applyLayout() signature
182	1	1	1	1	Updated CVS from distribution
230	1	1	1	1	Corrected class cast exception

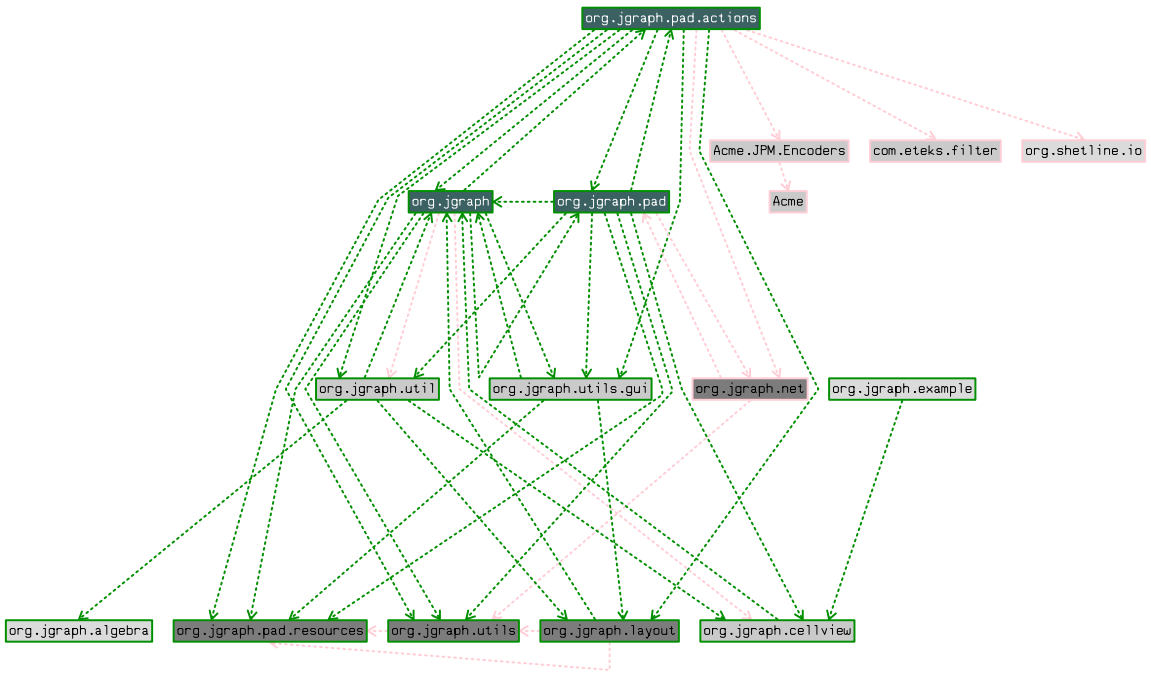


Figure 5.5. *JGraphpad shaded according to the number of authors modifying a package. The visualization used shades nodes so that the darker the node is, the more authors have modified it.*

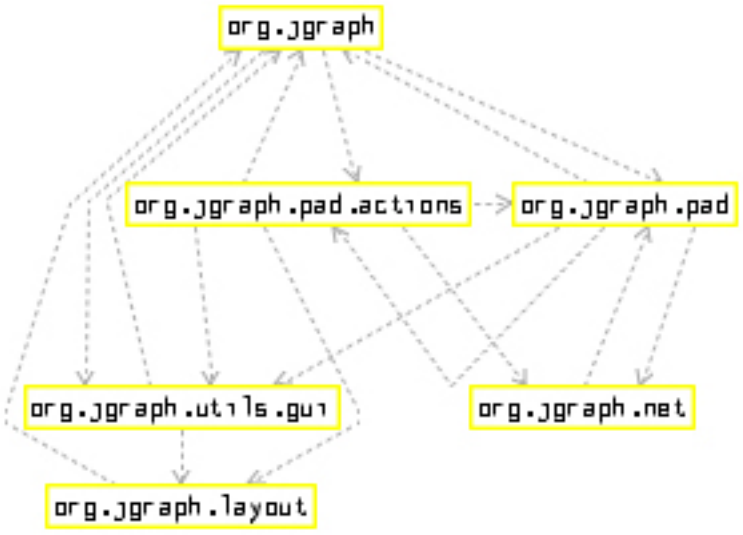


Figure 5.6. *A package diagram showing the effect of all MRs whose log contained the phrase "JGraph". All six packages shown in the diagram were affected by the change-set, though no inter-package relationships were added or deleted.*

JGraph dependencies are.

Figure 5.6 shows the net-effect these changes had on the architecture. No inter-package relationships have been added or deleted, but 6 packages have been affected.

5.3.6 When did a particular architectural entity appear?

The class org.jgraph.pad.GPGraph was selected as the architectural entity of interest. The Properties Dialog shows the MR in which the class was added and allows in one click viewing a new change-set that just includes that MR. The resulting diagram, filtered so that it includes only changed nodes and relationships, is shown in Figure 5.7. The diagram shows that GPGraph was added to the package at the same time as the GPUserObject class.

The log of the MR that added GPGraph is, "moved there again because of new instantiation policy [classes] have been moved out the pad package and got a protected constructor in order that they could be instantiated only via JGraphpad". In this case the context has

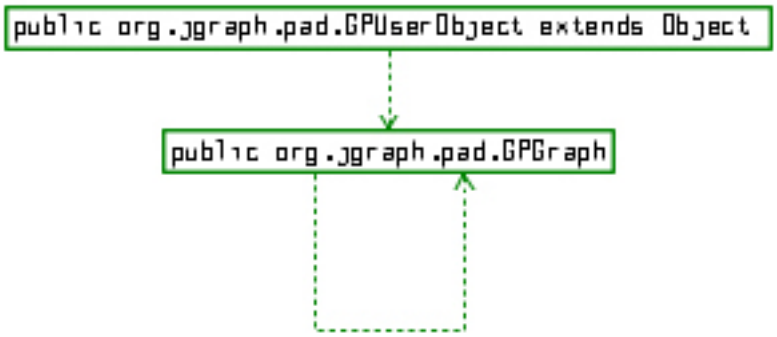


Figure 5.7. *A class diagram showing the effect of the change that added GPGraph. The GPUserObject class was added in the same MR as GPGraph.*

provided some information - that the addition is actually a move of the class from one package to another.

5.3.7 Who has made the most modifications to a package, and who made the last modification?

The package org.jgraph.pad.actions was selected as the package of interest. Figure 5.8 shows the packages in JGraphpad colored according to the author who made the most modifications to the package. The figure colors org.jgraph.pad.actions light blue, which from the color mapping (not shown) means that alderg made the most modifications to it. [1]

Right-clicking on the package allows us to bring up the Properties Dialog, which shows that the last modification was made by rvalyi.

[1] Currently, each time we run a query a new color mapping is created. That is why this color mapping of alderg to light blue is different from the color mapping of Question 3, where alderg was mapped to teal.

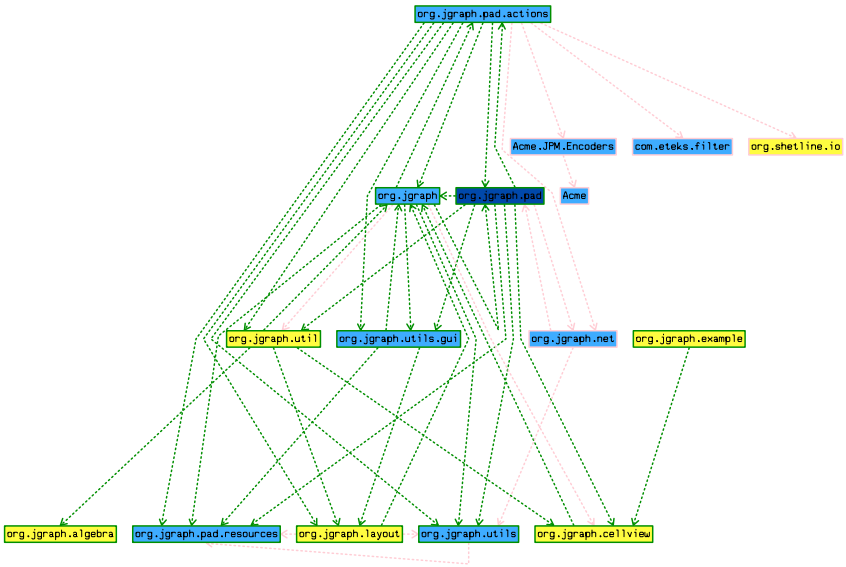


Figure 5.8. *JGraphpad packages colored according to the author that made the most modifications. For example, the color of org.jgraph.pad.actions is light blue, which from our color mapping (not shown) means that alderg is the author who made the most modifications to the package.*

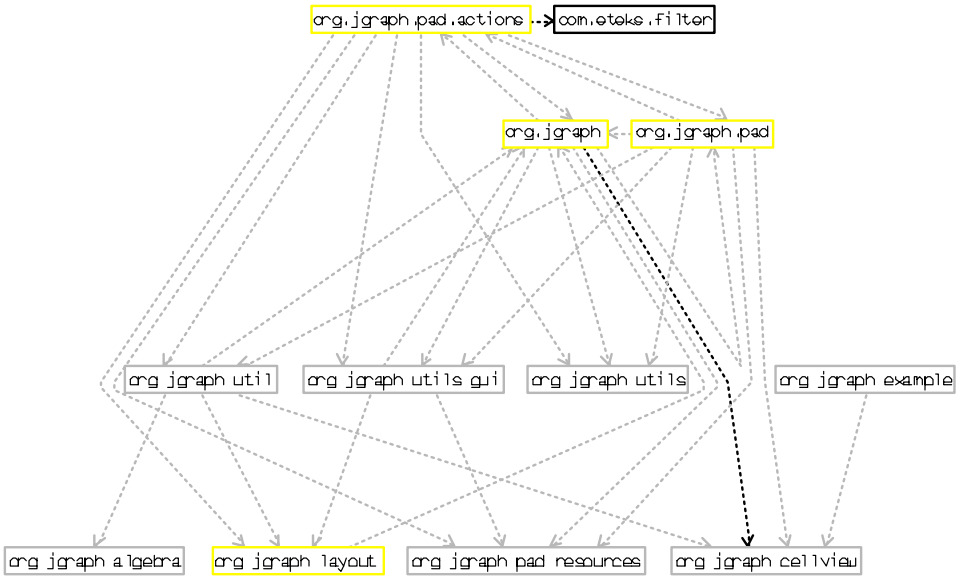


Figure 5.9. *A package diagram summarizing the changes to JGraphpad in the last week studied. Four packages have been modified (colored yellow), one package has been deleted (colored black), and two package dependencies have been deleted (colored black).*

5.3.8 What has been changed in the last week globally or in a specific package?

The last week of data that we have for JGraphpad includes 17 MRs. Figure 5.9 shows what change occurred to the architecture in that time. Four packages have been modified, one package has been deleted, and two package dependencies have been deleted.

The package org.jgraph.pad was selected as the package of interest. Figure 5.10 shows how the package has changed in the last week - five classes were added, three classes changed, and one class deleted.

5.3.9 What happened in a particular commit?

One of the changed classes in the last week to the package org.jgraph.pad is AbstractDefaultEdgeCreator. Double-clicking on the file gives a Text View of how the class's source

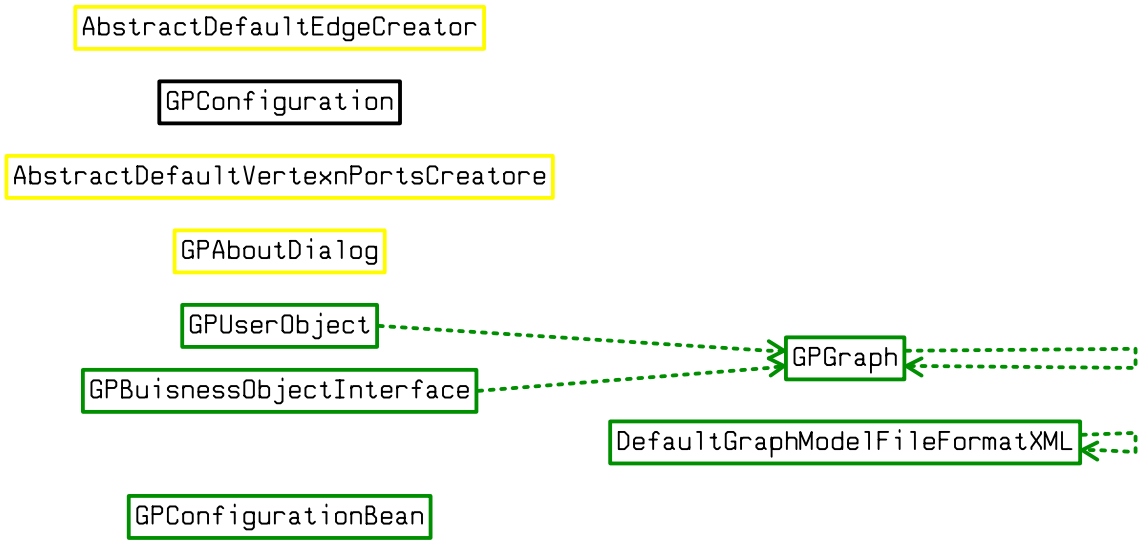


Figure 5.10. *A class diagram showing the changes to the org.jgraph.pad package in the last week studied. Five classes were added (colored green), three classes changed (colored yellow) and one class deleted (colored black).*

code has changed in the period studied. Figure 5.11 shows how the file has changed between versions 1.7 and 1.8., with the top pane showing a diff output of the two versions and the bottom pains showing each version. From this view it appears that the change made was in response to the moving of the getFileFormatProvider() method to GPFactoryProvider.

5.3.10 What packages have developers recently modified?

Querying for MRs in the last two weeks returns shows 22 MRs, 6 by d_benson and the rest by rvalyi. In turn, we viewed diagrams to show the effect of the last 5 MRs of each author. The diagram of d_benson showed one package modified, org.jgraph.layout. The diagram of rvalyi showed 2 packages, org.jgraph.pad.actions and org.jgraph.pad.

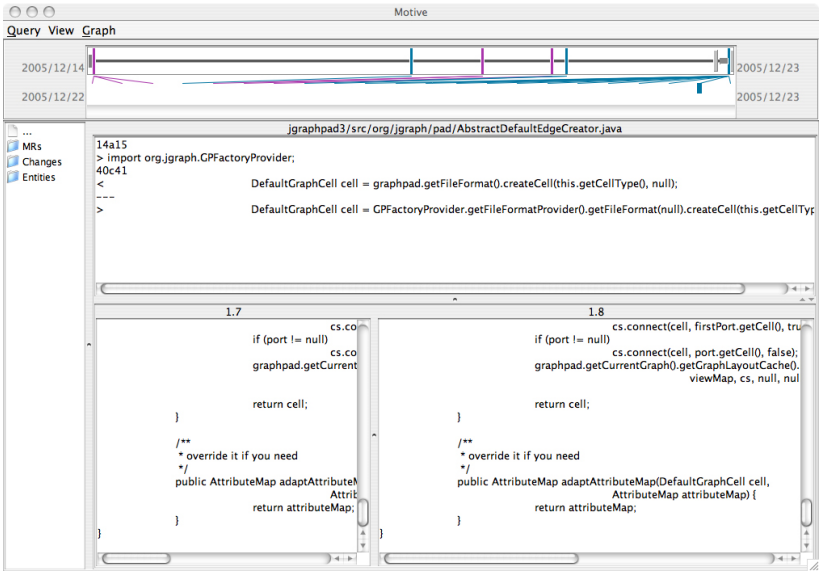


Figure 5.11. *A Text View showing the change between two versions of the AbstractDefaultEdgeCreator class*

5.3.11 What packages have not been modified in the recent past?

Figure 5.12 shows an overview of how JGraphpad has evolved, shaded according to when each package was last modified. Lightly shaded packages have not been modified in over 100 MRs, while darkly shaded packages were modified in the last 10 MRs. The three packages that are lightly shaded are acme.jpm.encoders, org.jgraph.algebra, and org.jgraph.example.

The package org.jgraph.example, highlighted in Figure 5.13, stands out because it has no incoming dependencies. Looking inside the package we see one file, IconExample, that seems to be an example of how to use a feature of JGraph. Figure 5.14 shows a highlight of the other package that still exists (Acme.JPM.Encoders is shown as phantom) and is lightly modified. This package has one incoming dependency, from org.jgraph.util. Our tool allows us to more closely examine the source of an inter-package dependency; as shown in Figure 5.15, we see that only two of the classes in the package are currently being used by the rest of the system. This may indicate a possible refactoring opportunity.

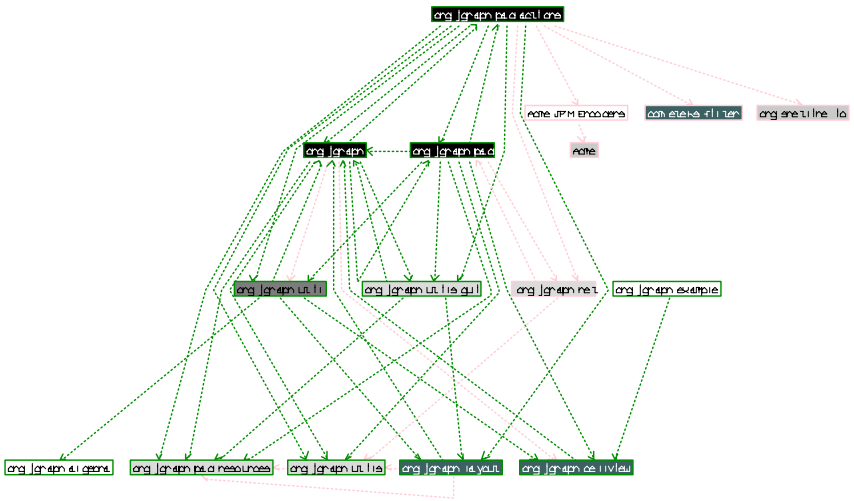
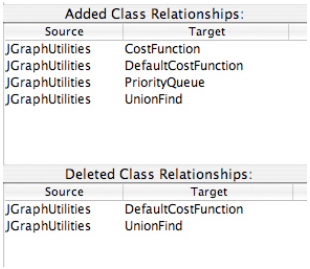


Figure 5.12. *JGraphpad shaded according to when packages were last modified. The darker a package is colored, the more recently it was modified. Three packages are lightly shaded, indicating they have not been modified in over 100 MRs.*

Figure 5.13. *A highlight of the org.jgraph.example package from Figure 5.12. The package has no incoming dependencies, indicating that no other packages are dependent on it.*

Figure 5.14. *A highlight of the org.jgraph.algebra package from Figure 5.12. The package has a single dependency.*



Added Class Relationships:

Source	Target
JGraphUtilities	CostFunction
JGraphUtilities	DefaultCostFunction
JGraphUtilities	PriorityQueue
JGraphUtilities	UnionFind

Deleted Class Relationships:

Source	Target
JGraphUtilities	DefaultCostFunction
JGraphUtilities	UnionFind

Figure 5.15. *A summary of the dependencies between the org.jgraph.util and org.jgraph.algebra packages. Four dependencies have been added, and two dependencies have been deleted. This means that as of the final version studied, only two of the classes in the org.jgraph.algebra package have dependencies remaining on them from outside the package.*

5.3.12 How productive has a particular developer been?

The author d_benson was selected as the author of interest. A change-set was created of the MRs made by d_benson over the last 3 months. Figure 5.16 shows a high-level overview of the effect d_benson has had on the architecture in that time. Four packages have been modified - org.jgraph, org.jgraph.layout, org.jgraph.pad, and org.jgraph.pad.actions - and no new dependencies have been added. Figure 5.17 shows a highlight of the changes to the SugiyamaLayoutAlgorithm class in the org.jgraph.layout package. The class has had a number of fields and methods modified, and the findMinimumAndSpacing() method added.

5.3.13 How much change was there between two releases?

To answer this question the user must know when releases occurred. We assume that the last modification recorded represents the latest release of the system, with the previous release exactly 6 months prior to that. Figure 5.18 shows what changes occurred to JGraphpad in that time. Four packages were deleted, along with their dependencies. Among the remaining packages, 2 dependencies were removed and 1 new dependency was added. The coloring used in the diagram shows the amount each package was modified, so Figure 5.18

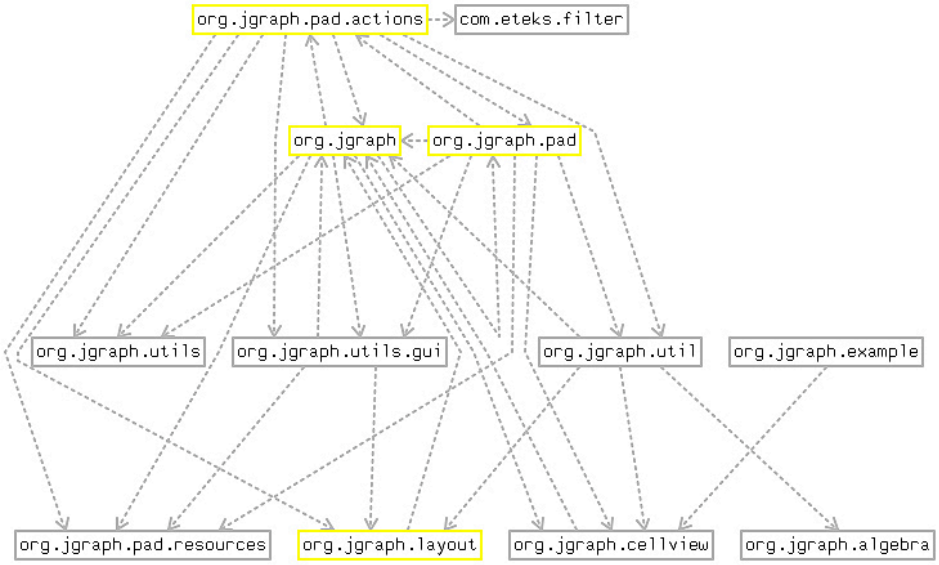


Figure 5.16. *A package diagram showing the modifications made by d_benson in the last 3 months studied. Four packages (colored yellow) were modified.*

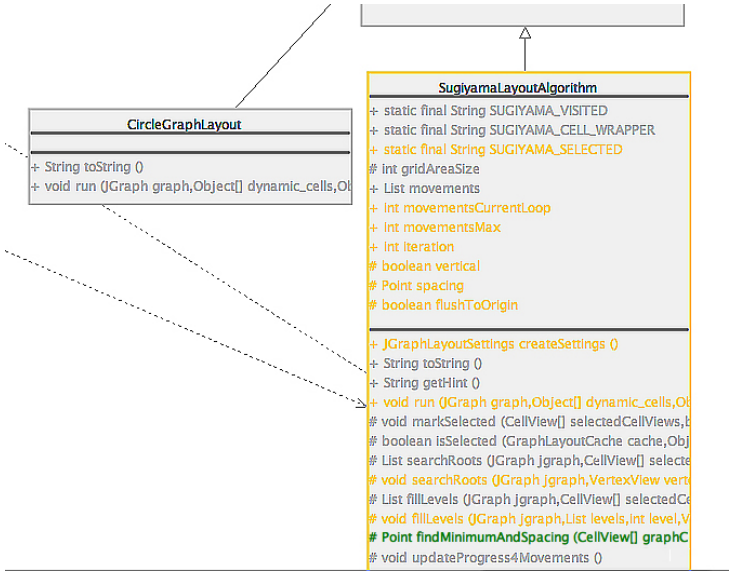


Figure 5.17. *A highlight of modifications made by d_benson in the last three months to the SugiyamaLayoutAlgorithm class in the org.jgraph.layout package. A number of fields and methods (colored yellow) were modified and the findMinimumAndSpacing() method added (colored green).*

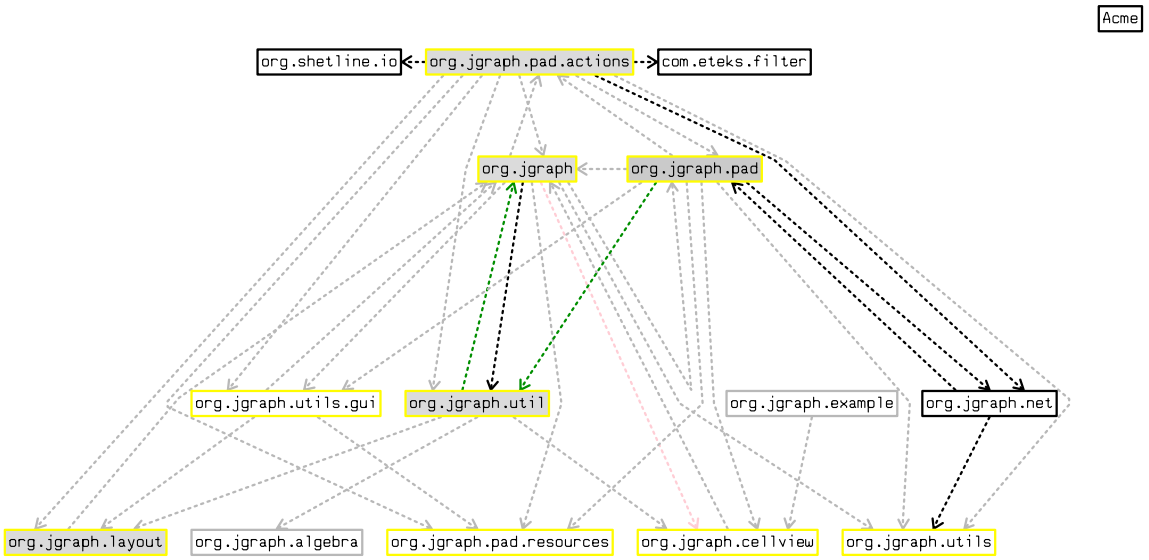


Figure 5.18. *A package diagram summarizing modifications made to JGraphpad in the last 6 months. The coloring used shades packages so that the darker they appear, the more MRs modified them. The diagram shows nine packages were modified (colored yellow), four packages and their dependencies were deleted (colored black), two other dependencies were deleted (colored black), and one new dependency was added (colored green).*

also shows that five packages were involved in a moderate amount of MRs, and the rest very few MRs.

5.4 SQuirreLSQL Evaluation

5.4.1 What packages are highly coupled?

SquirreLSQL is significantly larger than JGraphpad, with 160 packages. Although it was possible to identify the most coupled packages of JGraphpad from a package diagram, the larger size of SquirreLSQL made this approach impractical. Instead, our Visualization Database was queried directly to find the 10 packages with the most dependencies on them. The results are displayed in Table 5.3.

Table 5.3. *The 10 most coupled packages of SQuirreL SQL*

Package	*Dependencies*
squirrel_sql.fw.util	116
squirrel_sql.fw.util.log	105
squirrel_sql.client.session	98
squirrel_sql.fw.sql	98
squirrel_sql.client	80
squirrel_sql.fw.gui	68
squirrel_sql.client.plugin	53
squirrel_sql.client.action	48
squirrel_sql.client.gui.session	43
squirrel_sql.client.preferences	33

Figure 5.19. *A highlight of three packages of SquirreLSQL, shaded according to the number of times they have been modified. The darker the background color of the package, the more MRs have affected the package.*

[2]

5.4.2 What packages were frequently modified in the past?

A visualization was used that shade packages according to the number of MRs that had affected the package, with packages involved in over 100 MRs shaded black. Figure 5.19

[2]In this table, and throughout this section, to save space we have changed the naming scheme of certain packages. Packages whose name begins with "net.sourceforge.squirrel_sql," have had this prefix shortened to "squirrel_sql." For example, the first package in the table, squirrel_sql.fw.util is actually named net.sourceforge.squirrel_sql.fw.util. The shorter names allow us to conserve space with figures and tables.

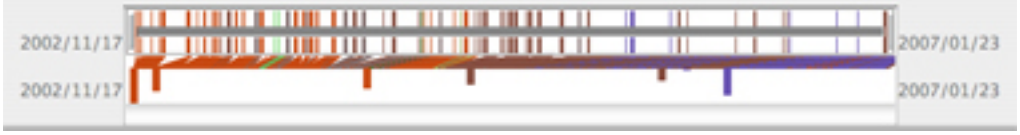


Figure 5.20. *The Temporal Slider for the change-set of all MRs of SQuirreL SQL. The bottom half of the slider highlights the 10 changes that have modified the most files, with the height of the 10 MRs indicating where it falls in the top 10 list (taller MRs modified more files). The color of the MR indicates what author made the change.*

shows a highlight of three packages shaded according to this view, showing two of the packages that were shaded black - squirrel_sql.client.preferences and squirrel_sql.client.action. In total, 15 packages were shaded black and were involved in over 100 MRs.

5.4.3 What were the architecturally disruptive changes in the past?

Figure 5.20 shows the Temporal Slider summarizing a change-set with all the MRs of SQuirreL SQL. The slider is designed to highlight the top 10 changes that affected the most files, though in this case some of these changes occurred very close together and are not clearly separated.

Table 5.4 shows changes that met at least one of the criteria for architecturally disruptive changes - being in the top 10 changes in regards to the number of files modified (FM), packages modified (PM), existing packages modified (EPM), and dependencies added (DA). This broad criteria lead to the inclusion of some changes that, from their logs, appear to not be architecturally disruptive. For example, changes that added new plugin packages without affecting the existing packages are changes that validate the modifiability of the architecture rather than being architecturally disruptive.

Table 5.4. *Potential architecturally disruptive changes to SQuirreL SQL*

MR	FM	PM	EPM	DA	*Log*
5101	76	9	0	66	Initial import from ... 0_2_4 tag
2026	32	2	0	15	New plugin
2012	30	7	0	33	First upload of ... MSSQL plugin
5532	30	2	2	0	Relocated dialect ... for ... refactoring plugin
2071	28	5	0	34	New plugin
1804	27	3	0	5	...patch - adds SQL parser
668	25	3	0	17	Initial import
5524	25	1	0	4	Relocated ... for ... refactoring plugin
2339	23	5	5	1	... Massive GUI code cleanup
2069	21	2	0	14	New plugin
4910	21	4	3	13	...configurable Schema ...
6401	21	1	1	0	Relocated ... to the refactoring plugin...
5978	18	4	4	0	EOL problems....license headers...
6423	18	4	0	25	New PostgreSQL plugin
6409	17	4	0	25	New plugin for DB2
6448	17	4	0	25	New H2
2894	16	7	7	0	Created API for Plugins...
2609	12	3	0	24	...SGA tracing ...
5522	8	3	0	24	Initial release for refactoring plugin
6096	8	3	0	21	Plugin for Derby
193	7	6	6	0	Cleanup
201	7	7	7	0	Cleanup
4533	7	7	7	0	Fixed concurrency bugs and clean up
4751	6	6	5	1	Test for ObjectTree...
6059	6	4	4	0	Relocated ... copies of BaseSourceTab...
1338	5	4	4	0	i18n
2582	5	2	0	20	Explain plan functionality of oracle
1890	4	4	4	0	Mods to get 1.2beta5 out the door

5.4.4 What packages have been modified by a broad group of developers.

Figure 5.21 shows an overview of what are the most broadly modified packages in SQuirreL SQL, with a visualization that shades nodes according to the number of developers that have modified them. The diagram was filtered so that it shows only the nodes colored black, which in the current visualization means they have been modified by at least 5 authors. There are 9 packages that meet this criteria.

5.4.5 Given a keyword, what changes used it in their commit log and how did they affect the architecture?

One term that we frequently saw was in MR logs was, "i18n", a standard abbreviation for internationalization. From querying the database, 783 MRs had this term in their logs. If we make the assumption that this term only exists when the change has something to do with internationalization, then over 10% of the changes to the system had to do with internationalization.

Motive was used to view the effect of MRs which had the term "i18n" in their log, except for two MRs removed because their very long logs seemed to indicate that internationalization was only a small part of the reason for their change. Figure 5.22 shows a summary of how internationalization-related changes have affected the software architecture. To make the diagram more clear, most packages have had their name replaced with "...". From the diagram it can be seen that the added sql_squirrel.plugins.i18n package is involved in 11 new dependencies and the modified package squirrel_sql.fw.util is involved in 21 new dependencies. Examining the dependencies squirrel_sql.fw.util is involved in shows they almost all come from two added classes, StringManager and StringManagerFactory, both of which, from their comments, have to do with loading internationalization strings.

We also noticed that a lot of the logs seemed to indicate specific work for certain

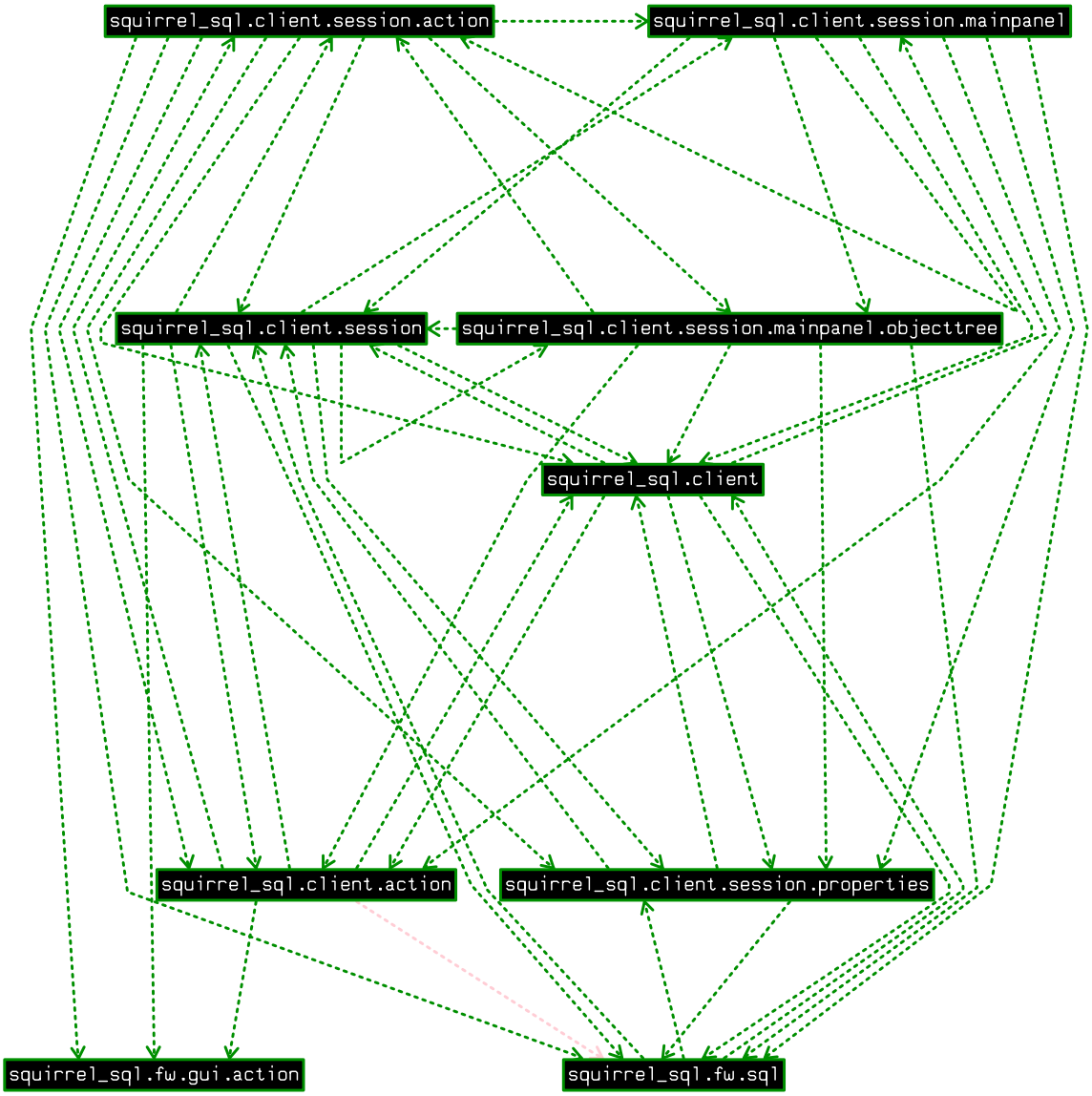


Figure 5.21. *A highlight of the package diagram of SQuirreL SQL showing the packages modified by at least 5 developers. The diagram was colored so that the darker a node the more developers have modified the package.*

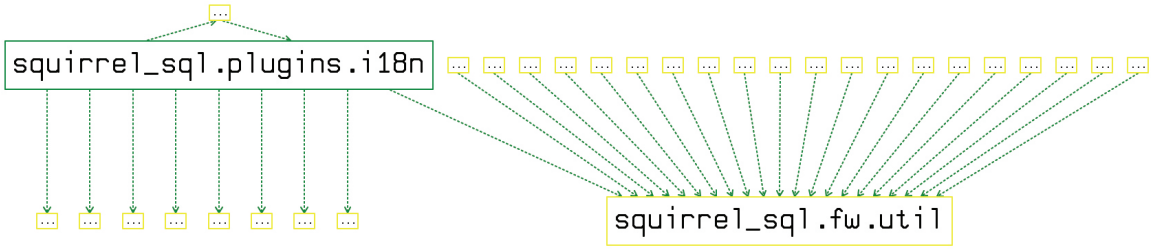


Figure 5.22. *A package diagram showing new dependencies that were created in MRs that had the term "i18n" in their log. Most packages have had their name replaced with "..." to highlight the sql_squirrel.plugins.i18n and squirrel_sql.fw.util packages, which are each involved in a large number of the introduced dependencies.*

databases. For example, "Introduced Oracle specific code to allow editing of tables that contain Date columns." We chose three databases that we knew were supported because plugins existed for them, and ran a query for all MRs whose log included the term "oracle", "mysql", or "db2".

Figure 5.23 shows the changes, with the packages filtered so as to remove the database specific plugins, such as plugins.oracle. This still results in a surprisingly large number of changed classes. As well, the figure shows there were some dependencies added in these changes. From looking at some of the logs there seemed to be a varied number of motivations for making these changes:

- "Fix problem with unique id columns in select * in Oracle..."

  This is an example of an MR that seems to be associated with fixing a database-specific bug.

- "Improved startup performance for Oracle..."

  This is an example of an MR that seems to be associated with tweaking general code to improve the performance of a specific database.

- "Provide an work-around for databases (like MySQL) that do not return tables in other schemas/catalogs where null is specified for schema and/or catalog"

  This is an example of an MR that seems to be associated with different databases

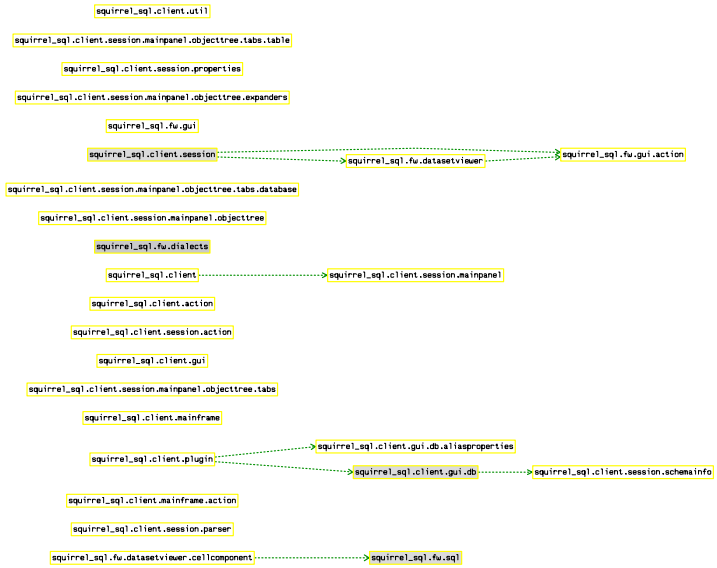


Figure 5.23. *A package diagram showing packages modified in database-specific changes. Database specific plugins have been removed from the diagram (e.g. plugins.oracle). The diagram is colored so that the darker a package is colored, the more MRs have affected it.*

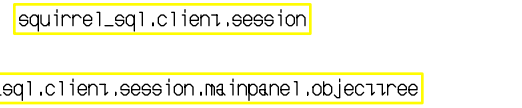


Figure 5.24. *A highlight of the package diagram showing all SquirreL SQL packages modified in MR 1134. The two packages shown (colored yellow) were modified.*

interpreting a standard slightly differently.

5.4.6 When did a particular architectural entity appear?

The class squirrel_sql.client.session.SessionWindowManager was selected as the entity of interest. The class was added in MR 1134, by the developer uenus, with the log, “cleanup session handling”. Figure 5.24, shows there were two packages changed in that MR. From Figure 5.25 we can see that another class, SessionInternalFrame, was added to squirrel_sql.client.session at the same time as SessionWindowManager.

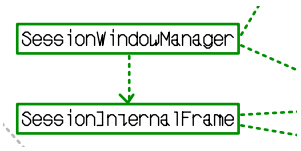


Figure 5.25. *A highlight of the class diagram showing the changes to the squirrel_sql.client.session package in MR 1134. The two classes shown (colored green) were added.*

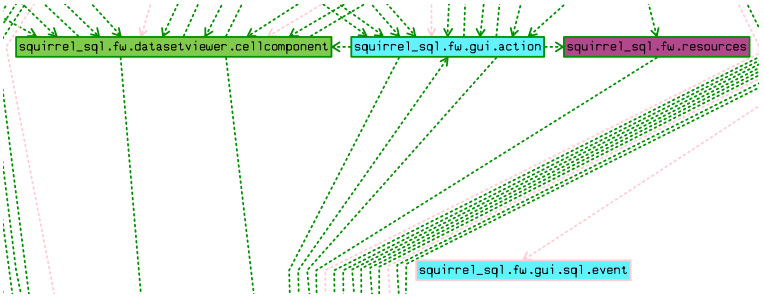


Figure 5.26. *SQuirreL SQL packages colored according to the author that made the most modifications*

5.4.7 Who has made the most modifications to a package, and who made the last modification?

The package squirrel_sql.fw.datasetviewer.cellcomponent was selected to be examined. Figure 5.26 shows a highlight of the packages in SQuirreL SQL colored according to the author who made the most modifications to the package. The package of interest is colored green, which in our color mapping means that gwghome has made the most changes to the package. From the Properties Dialog, the last change to the package was made by manningr.

Figure 5.27. *A highlight of the package diagram summarizing the changes to SQuirreL SQL in the last week studied. Eight packages (colored green) were added and five packages (colored yellow) were modified.*

Figure 5.28. *A highlight of the class diagram showing changes to the squirrel_sql.fw.sql package in the last week studied. The two packages shown (colored yellow) were modified.*

5.4.8 What has been changed in the last week globally or in a specific package?

There were 51 MRs in the last week of data we have for SQuirreL SQL. Figure 5.27 shows the changes in that period to the architecture of the system. Eight packages were added, and five already existing packages where modified.

Figure 5.28 shows the changes during the last week to the squirrel_sql.fw.sql package. Two classes were changed, SQLDatabaseMetaData and QueryTokenizer.

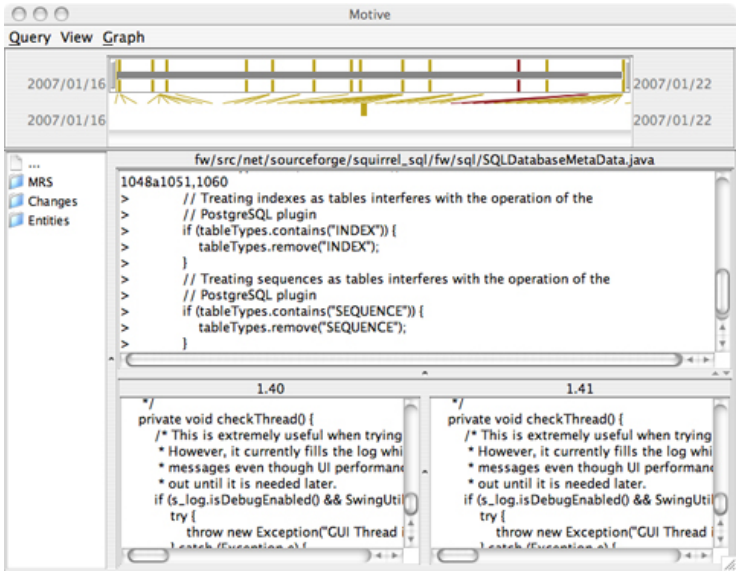


Figure 5.29. *A Text View showing the changes between two versions of the SQLDatabaseMetaData class.*

`squirrel_sql.client.plugin` `squirrel_sql.fw.gui.action` `squirrel_sql.fw.sql`

Figure 5.30. *A highlight of the package diagram showing the changes made by author gerdwagner's last 5 MRs. The three packages shown (colored yellow) were modified.*

5.4.9 What happened in a particular commit?

The class squirrel_sql.fw.sql.SQLDatabaseMetaData was selected to examine. Figure 5.29 shows the change to the file when manningr committed MR 6436, with the log, "for the PostgreSQL plugin to work effectively with indexes and sequences, it is necessary that they are not identified as Table types". The top panel shows the output of a diff of the two versions, and the lines of code that were added.

5.4.10 What packages have developers recently modified?

Querying for MRs in the last 2 weeks shows 93 MRs by manningr and 7 by gerdwagner. In turn, we viewed diagrams that showed what packages they had each modified in

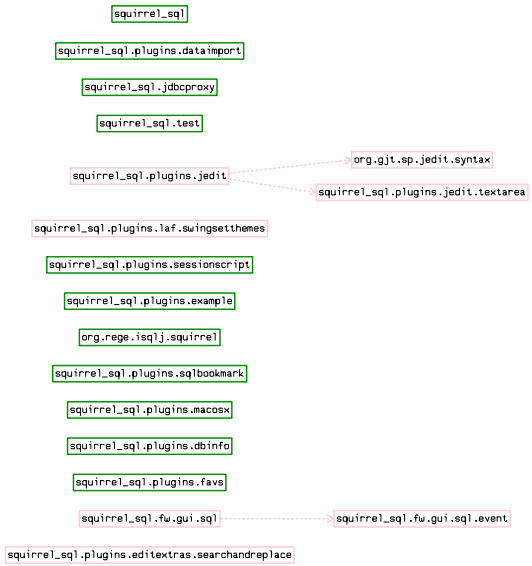


Figure 5.31. *A highlight of the package diagram showing packages with no remaining dependencies that were not modified in the last 1000 MRs. Seven packages (colored pink) are phantom and eleven packages (colored green) were added.*

their last 5 MRs. As shown in Figure 5.30, gerdwagner has made changes to 3 packages (squirrel_sql.client.session, squirrel_sql.fw.gui.action, squirrel_sql.fw.sql, and squirrel_sql.client.plugin).

The diagram of manningr shows no changes to the architecture. Examination of the MRs shows that 4 of them were changes to branches of the system. The most recent MR by manningr, 6460, has the log comment "Updated documentation" and did not affect any source-code.

5.4.11 What packages have not been modified in the recent past?

A diagram was created whose visualization shades packages according to when they were last modified, with packages shaded white if they have not been modified in over 1000 MRs. In total, almost half the packages in the system (76) were shaded white. Figure 5.31 shows the packages with no remaining dependencies (either having no dependencies, or all

the dependencies on the package being phantom). Eleven of these packages are shown as added, which means they still exist. Investigation of these packages to identify dead code found that squirrel_sql, squirrel_sql.jdbcproxy, and squirrel_sql.test have no dependencies because they contain dynamically loaded testing code. The other added packages are related to plugins (including org.rege.isqlj.squirrel), and it is expected that the system would not be statically dependent on a dynamically loaded plugin. It is conceivable some of the plugins could be removed because nobody uses them, but that cannot determine just from examination of the source code.

5.4.12 How productive has a particular developer been?

Figure 5.32 shows the changes made by colbell in the last 3 months. Fifteen MRs were made in that time. The effect was fifteen packages modified, one deleted, and some inter-package dependencies removed.

5.4.13 How much change was there between two releases?

As in the JGraphpad case study, we assume that the last modification recorded represents the latest release of the system, with the previous release exactly 6 months prior to that. Figure 5.33 shows the packages added to the system in that period, with Figure 5.34 showing the other changes to the system. Both diagrams are colored by the amount the package was modified. For example, Figure 5.33 shows a moderate amount of change to the refactoring, postgres, and informix plugins, and a heavy amount of change to squirrel_sql.fw.dialects.

5.5 Discussion

As discussed in our section on implementation details, Motive has two main extension points: the queries that can produce change-sets, and the coloring of the entities under display. Table 5.6 shows how these extension points were used to answer the questions we

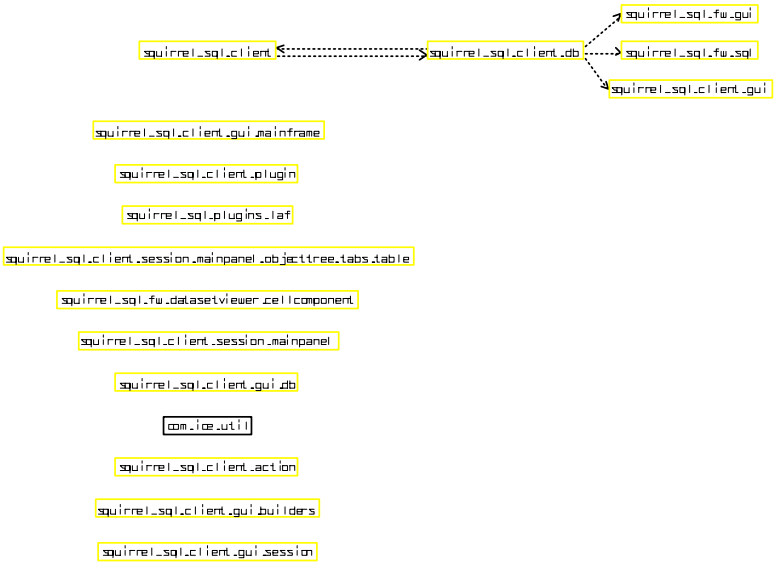


Figure 5.32. *A package diagram summarizing the modifications made by colbell in the last 3 months. Fifteen packages (colored yellow) were modified and one package (colored black) was deleted. There were also five inter-package dependencies (colored black) deleted.*

Figure 5.33. *A package diagram showing the packages added to SQuirreL SQL in the last 6 months. All package borders and relationships are added (colored green). The background color of the package indicates how many times it was was modified, with the more MRs affecting a package the darker it is colored.*

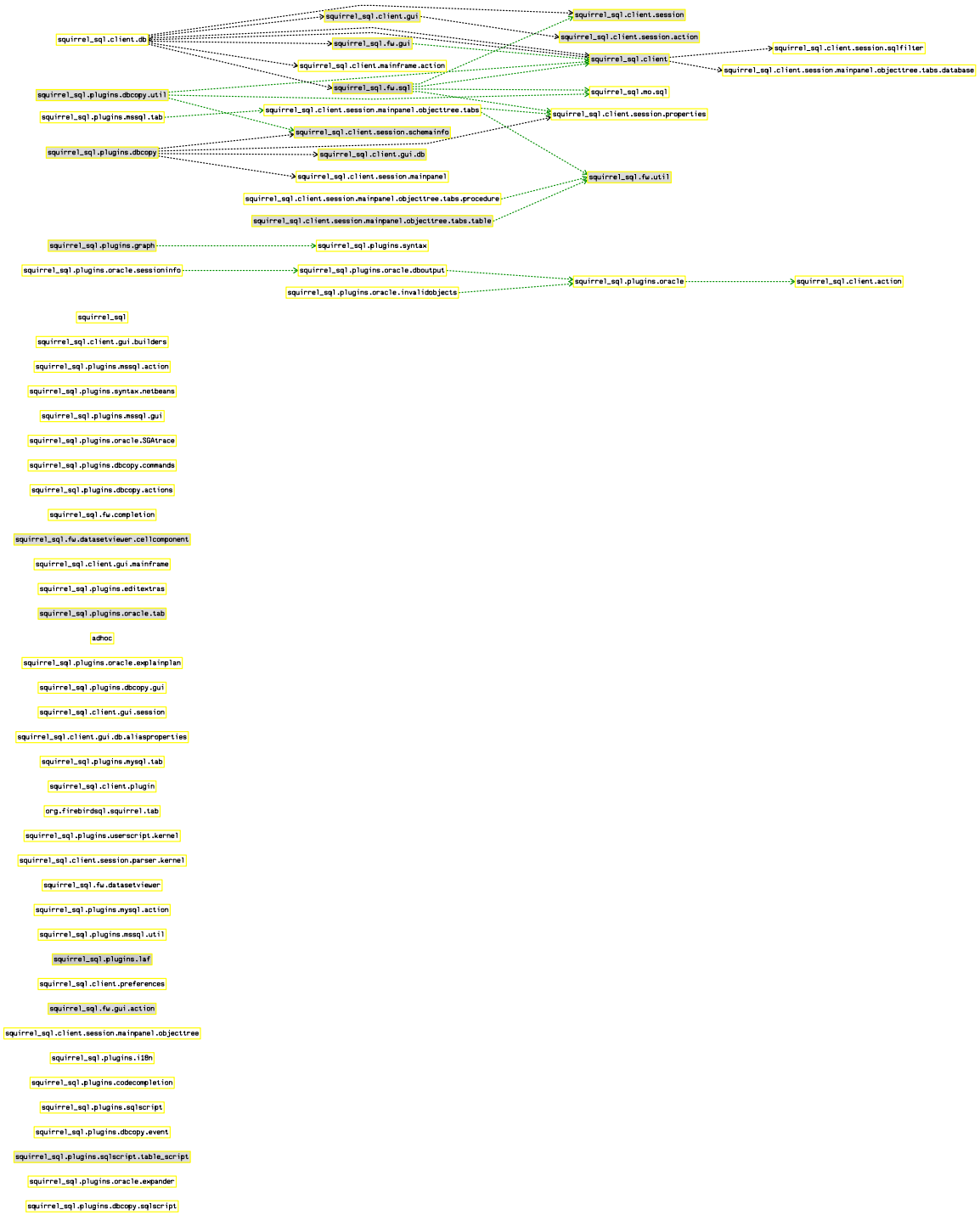


Figure 5.34. *A package diagram showing the packages changed in SQuirreL SQL over the last 6 months. All packages have been modified and are colored yellow. The modifications to some packages have removed some dependencies (colored black). The darker a package is colored the more MRs modified it.*

Table 5.6. *How the evaluation questions were answered*

$Question$	$Change-set$	$Shading$
1	All MRs	Default
2	All MRs	Amount modified
3	All MRs	N/A
4	All MRs	Distinct authors that modified
5	Keywords	Default
6	Particular MR	Default
7	All MRs	Author that most modified
8	Time Period	Default
9	Particular MR	Default
10	Time Period, Author	Default
11	Time Period	Last Modified
12	Time Period, Author	Default
13	Time Period	Default

posed in the case studies.

There were five types of coloring used: *default*, *amount modified*, *distinct authors that modified*, *author that most modified*, and *last modified*. Three of these, *amount modified*, *distinct authors that modified*, and *last modified*, were used to group the entities into rough categories based on how heavily shaded their background color was. The other non-default coloring mechanism, *author that most modified*, is used to show a specific attribute of the entity.

There were five types of change-sets that used: *all MRs*, *particular MR*, *time period*, *keyword*, and *author*. Although we have not conducted an empirical study that compares our approach with other architecture-centric visualizations, we would expect that other approaches would have difficulty answering the questions that required the *keyword* and *author* change-sets, as these change-sets cannot be represented in an unbroken period of

time.

5.5.1 Change-set usefulness

In addition to time-based change-sets, there were three areas where we found the ability to create flexible change-sets useful in helping to highlight some aspect of software evolution:

1. Studying groups of changes based on log entries

   Answering Question 5 required creating different change-sets based on keywords in the log entries. In the JGraph case study, this ability was used to provide an indication of the evolutionary coupling between JGraph and JGraphpad. In the SQuirreL SQL case study, this ability was used to gain a sense of the key architectural entities related to providing internationalization support, and to gain a sense of what architectural entities were most affected by the need for database-specific changes.

2. Examining author changes

   Answering Questions 10 and 12 required highlighting the changes of a particular author. In Question 10 this was used to view what an author has been working on recently. In Question 12 this was used to view the overall impact that a particular author had on the system.

3. Filtering large transactions

   In [48], Zimmermann and Weissgerber talk about the need to deal with "large transactions" during the preprocessing of CVS data. They suggest transactions that modify a large number of files, such as the changing of an include file, should be filtered out of the analysis. An example of this type of change that occurred to JGraphpad was MR 46, which changed the copyright text of files. While in a standard architecture-centric view it may be necessary to filter out such changes as a preprocessing step, some types of analysis might require examining this change. Our approach allows the user to first create a general query that retrieves a superset of the MRs they are interested in, and then selectively remove undesired changes from the change-set.

This eliminates the need to decide in a preprocessing step what changes should be removed.

We used this filtering ability in this evaluation in Question 5 of the SQuirreL SQL case study. The general query that returned all MRs with "i18n" in their logs included some MRs whose very long log entries indicated that internationalization was only one of the logical changes made in the MR. These MRs were filtered out from the change-set.

5.5.2 Limitations

The limitations identified during the evaluation include both the difficulties with answering Questions 1 and 3, and general concerns related to the extraction process, the current visualization, and the performance of the visualization tool.

1. Question 1

   Question 1 called for identifying the most highly coupled packages. While it was possible to detect these packages in JGraphpad's small package diagram, the diagram of SQuirreL SQL proved too large to answer the question. The current coloring extension mechanism allows the user to roughly group nodes into categories. Answering Question 1 requires the ability to rank nodes in a list. One possible extension mechanism to achieve this ranking might be a layout mechanism that orders all the nodes in a list based on a user-defined criteria. This would allow the user to order the nodes by their coupling, and answer this question by visually selecting the ten entities to the left of the list to discover the most coupled packages.

2. Question 3

   Question 3 called for identifying the most architecturally disruptive packages according to a number of detailed criteria. In both case studies it was possible to determine from the Temporal Slider the most architecturally disruptive changes according to one criteria (files modified). However, direct database queries were needed to de-

termine the most disruptive changes according to the other criteria specified in the question. This question shows that rather than hardcoding what MRs are highlighted on the Temporal Slider, the user should have the ability to select both the number of MRs they want highlighted (currently hardcoded to 10) and the mechanism by which they want the MRs highlighted (currently hardcoded to the files modified in the MR).

3. Extraction limitations

   In the answer of Question 12 in the SQuirreL SQL, the view showed no changes to the architecture of the system in manningr's last 5 MRs. This is accurate, but not useful. The reason there are no changes shown is that all the MRs in that change-set either were changes to branches or were changes that did not affect source code files. Adding support for viewing changes to branches would require significant effort, as the decision to not support branches was made at the design stage. However, including changes to other types of files, such as build files, would be a fairly simple modification to our extraction process, and would help make what has happened in those MRs that did not affect source code.

4. Graph export limitations

   Our focus in this work was not on developing another graph visualization tool. Instead, we created a simple interface that allowed users to create the view that they wanted, then export that view to more mature graph visualization tools. Our tool currently supports both the graph description language (GDL) and the graph exchange language (GXL) file formats. However, there are two main limitations with this export ability.

   The first problem is that we are not using the ability of GDL or GXL to export subgraphs. In Java, packages are hierarchically organized by their name. For example, in SQuirreL SQL there are 4 packages that are part of the PostgreSQL plugin. The names of these packages all begin with "net.sourceforge.squirrel_sql.plugins.postgres." The hierarchical naming indicates that the packages belong to the SQuirreL SQL project, that they are part of the plugins subsystem, and that the particular plugin

they belong to is the PostgreSQL plugin. Ideally, these hierarchical names would be exported into different subgraphs so the user could, for example, collapse the 4 packages that are part of the PostgreSQL plugin into one node.

Another problem is that we do not provide support for exporting detailed class diagrams. Currently, Motive allows the user to select simple class diagrams, which only show the class names on the node, and detailed class diagrams, which include the fields and methods of the class. We have not yet investigated how difficult it will be to export these types of diagrams.

5. Performance

   The algorithm for computing the impact of a change-set works by iterating over all the MRs in the total-set, and, for each entity/relationship modified in the MR, recording the change in the relevant annotation list. Assuming that the information about what entities and relationships modified in an MR can be found in constant time, our algorithm works in *O((e+r)*MRs)* where *e* is the number of entities in the diagram, *r* is the number of relationships, and *MRs* is the number of MRs in the total-set. The *(e+r)* term comes from the fact that any MR will have modified at most every entity and relationship in the system. The multiplication of *(e+r)* by the number of MRs in the total-set comes from looping over every MR in the total set.

   In the small-sized case study the implementation of the computation algorithm seemed adequate. Running on a 1.42GHz G4, computing the impact on 16 packages and 47 relationships over 344 MRs took about 7 seconds. In the medium sized-case study, however, computing the impact on 160 packages and 1334 relationships over 6463 MRs took about 124 seconds.

   We believe that the performance of the computation algorithm could be dramatically improved with a better implementation of the algorithm. For example, currently the implementation of our algorithm uses one database query for every MR in the total-set to select its effect. A better implementation strategy would be to select all the necessary data about the total-set in one initial query and then select the effect of

each MR from the in-memory query results.

Another performance boost would come from caching the impact of a change-set in the database once it has been computed once. It may also be of great benefit to research how the tool would be used in practice, and possibly pre-compute the effect of certain commonly used change-sets (for example all the MRs in the system).

5.6 Summary

In the two case studies performed, we found some evidence that the ability to view change-sets was useful in helping to answer questions posed by researchers, developers, and their managers. While the two extension points that are currently part of Motive were sufficient to allow answering almost all the questions of the case study, the two questions that could not be answered indicate the need to develop other extension points. The case studies were also of great benefit in identifying numerous opportunities to improve the implementation of Motive.

Chapter 6

Conclusions

6.1 Summary

This thesis focused on examining the evolution of a software system from the perspective of change-sets. Change-sets allow a user to examine the net impact of any group of MRs, including groups that do not represent the changes between two periods of time. The motivation of our work was an intuition that change-sets could be useful in helping to answer the diverse questions that different groups have about software evolution.

To explore the usefulness of change-sets we developed a process for computing the net-effect of a change-set on a software system, and a method for displaying the net-effect using annotated architectural diagrams. The process and diagrams were designed to be lightweight and easily extensible. We explored one specific implementation in Motive, the prototype tool we created.

With a working tool we were able to apply our change-set approach to examining the software evolution of two real-world systems. We used Motive to attempt to answer questions drawn from the motivations of the three main groups interested in software evolution: researchers, developers, and their managers. With only a few exceptions, Motive did prove capable of answering these questions, including those we would expect time-based approaches to have great difficulty answering. Our evaluation demonstrates that change-sets

do seem to have a great deal of potential in the perspective they can bring to the study of software evolution.

6.2 Contributions

This thesis has made the following contributions:

- A process for computing the net-impact of a change-set on a software system's architectural evolution was developed.
- Architectural impact views which visualize the net-impact of a change-set using annotated architectural diagrams were developed.
- Motive, a prototype tool that allows a user to select a change-set and view its net-impact was implemented.
- A survey of related evaluations was conducted, and a set of questions based on those evaluations was created with the goal of being representative of the needs of researchers, developers, and managers.

6.3 Future Work

The previous chapter described future work that could be done to improve our implementation. In this section we describe future research directions for applying change-sets to the study of software evolution. These research directions fall into two main categories: *selection* of the change-set to be studied, and *visualization* of the desired change-set.

6.3.1 Change-set selection

In our evaluation, the query criteria we used to select change-sets was mostly based on the MR metadata. Some examples of criteria used were: building change-sets by log entries, building change-sets by author, and building change-sets by modification date. As well

as further research into how these change-sets can be made more useful to researchers, developers, and their managers, there is much work that could be done discovering what other interesting change-sets can be created.

One change-set that we found useful was the changes of a particular author. However, it became clear based on our case studies that the identification of what author made a particular change could be improved. As noted by Taylor and Munro [39], the developer who commits a particular change may not be the author of the change, with credit to the actual author possibly being indicated in a log comment. For example, the log comment of MR 3671 of SQuirreL SQl begins, "Alexander Buloichik's patch." We are currently identifying the author of the change as manningr, the developer who checked in the code. Natural language processing techniques might be useful in helping to automatically determine the most likely author for a particular MR. For example, they could recognize that MR 3671 was really made by Alexander Buloichik, not manningr.

However, it is also inaccurate to assume that one MR is equal to one logically related change. Although this is a common assumption among researchers studying software evolution, our method is based on grouping logically related changes, and so may be particularly affected by an inappropriate grouping of changes. For example, in answering Question 5 of the SQuirreL SQL case study some MRs were identified by their long logs to include logical changes other than the one we were interested in. Although the user always has the ability to remove these MRs from the change-set manually, a more desirable approach would be some automatic detection that the set of files changed in the same physical commit represent several different logical changes, and splitting this commit into several MRs.

In addition to more research on developing accurate MR metadata, a future direction of research is to look into what other useful change-sets might be created by grouping MRs based on analysis of the MR Impact, and synthesized information. For example, we mentioned in the model description that synthesized information might be used to create the change-set of "MRs that involved refactoring" by using an algorithm developed by Dig

et al. [8]. We have not yet had an opportunity to examine what existing algorithms exist that could be used as a criteria for building change-sets.

Change-sets might also be created by analysis of different kinds of software trails. This work focused on the data stored in an SCM system. However, other approaches have shown that by using data from issue tracking systems and dynamic execution traces, it is possible to map source code to features. We could use this type of mapping approach to build change-sets based on the evolution of features.

Critical to deciding what change-sets to invest time exploring is to know what change-sets users might find of value. Empirical studies of the needs of researchers, developers, and managers, would be of immense value in guiding our work.

6.3.2 Change-set visualization

This thesis focused on showing the impact that change-sets had on the architecture of a software system. We presented a model for computing the impact of a change-set on the architecture, and visualizing this impact using architectural impact views. We also presented a prototype tool that implemented one type of architectural impact view. There is a lot of opportunity of future research both in exploring what other types of architectural impact views might be useful, and finding other ways the impact of a change-set might be computed and visualized.

As described in our related work, there are a number of other architecture-centric approaches that have developed their own visualization techniques. These approaches might be integrated into our architectural impact views. For example, the YARN [21] approach to animating the evolution of a dependency graph could be used to animate the effect of a change-set. Rather than "re-invent the wheel", it would be valuable to separate our change-set computation and change-set visualization into two different tools, and look into how they can best be integrated with other approaches.

Our current change-set computation focuses on what annotations occurred to each architectural entity/relationship. Artifact-centric information could easily be added to this

process by considering the files of the project another architectural entity. Developing a process for computing Metric- and Feature-centric information over a change-set, however, would be very challenging. For example, one metric used by RelVis [31] was nrFuncs, the number of function/methods implemented in an entity. Because we are possibly ignoring intermediate MRs in a change-set approach, studying the change in this metrics value from one MR in the change-set to another would not give any useful information. However, it may be of use to store with each MR the change in nrFuncs compared to the previous MR, and then view the evolution of that metric over a change-set.

A more general computation process would also allow different targets of visualization. For example, if the computation process supported calculating the effect of a change-set in terms of metrics, then this information could be viewed according to methods developed by metric-centric approaches. As is the case with extending the architecture-centric method of visualization, the key to supporting different targets is developing mechanisms for allowing other approaches to tie into our computation tool.

Bibliography

[1] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[2] D. Beyer and A. E. Hassan, "Evolution storyboards: Visualization of software structure dynamics," *icpc*, vol. 0, pp. 248–251, 2006.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in oo software through visualization," in *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2003, p. 44.

[4] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail, "Cvssearch: Searching through source code using cvs comments," in *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 364.

[5] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA: ACM Press, 2003, pp. 77–ff.

[6] C. O. S. V. Control, "Cvs - open source version control," accessed 31 - August - 2007. [Online]. Available: http://www.nongnu.org/cvs/

[7] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components." in *ECOOP*, ser. Lecture Notes in Computer Science, D. Thomas, Ed., vol. 4067. Springer, 2006, pp. 404–428. [Online]. Available: http://dblp.uni-trier.de/db/conf/ecoop/ecoop2006.html#DigCMJ06

[9] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner, "Seesoft-a tool for visualizing line

oriented software statistics," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 957–968, 1992.

[10] M. Fischer and H. Gall, "Visualizing feature evolution of large-scale software based on problem and modification report data," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, pp. 385–403, 2004.

[11] J. Froehlich and P. Dourish, "Unifying artifacts and activities in a visual tool for distributed software development teams," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 387–396.

[12] F. S. F. (FSF), "Diffutils website," accessed 15 - June - 2007. [Online]. Available: http://www.gnu.org/software/diffutils/

[13] H. Gall, M. Jazayeri, and C. Riva, "Visualizing software release histories: The use of color and third dimension," in *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999, p. 99.

[14] D. Garlan and D. Perry, "Introduction to the special issue on software architecture," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, 1995.

[15] D. M. German, "Mining cvs repositories, the softchange experience," in *Proceedings of the First International Workshop on Mining Software Repositories*, 2004, pp. 17–21. [Online]. Available: http://www.turingmachine.org/~dmg/dchurch/mining.pdf

[16] D. M. German, "Using software trails to reconstruct the evolution of software: Research articles," *J. Softw. Maint. Evol.*, vol. 16, no. 6, pp. 367–384, 2004.

[17] D. M. Germán, A. Hindle, and N. Jordan, "Visualizing the evolution of software using softchange." in *SEKE*, 2004, pp. 336–341.

[18] D. M. Germán and A. Mockus, "Automating the measurement of open source projects," in *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, USA, 2003.

[19] T. Girba, S. Ducasse, and M. Lanza, "Yesterdays weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," *icsm*, vol. 00, pp. 40–49, 2004.

[20] O. Greevy, S. Ducasse, and T. Gîrba, "Analyzing software evolution through feature views: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 6, pp. 425–456, 2006.

[21] A. Hindle, Z. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt, "Yarn: Animating software evolutions," *Accepted to 2007 IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT-07)*, 2007.

[22] R. Holt and J. Y. Pak, "Gase: visualizing software evolution-in-the-large," in *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 163.

[23] JGraph, "Jgraph website," accessed 15 - June - 2007. [Online]. Available: http://sourceforge.net/projects/jgraph/

[24] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution: Survey articles," *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 77–131, 2007.

[25] M. Lanza, "The evolution matrix: recovering software evolution using software visualization techniques," in *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM Press, 2001, pp. 37–42.

[26] M. M. Lehman and L. A. Belady, Eds., *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[27] M. M. Lehman and J. F. Ramil, "An approach to a theory of software evolution," in *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA: ACM Press, 2001, pp. 70–74.

[28] M. M. Lehman and J. F. Ramil, "Rules and tools for software evolution planning and management," *Ann. Softw. Eng.*, vol. 11, no. 1, pp. 15–44, 2001.

[29] J. I. Maletic, A. Marcus, and M. L. Collard, "A task oriented view of software visualization," in *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA: IEEE Computer Society, 2002, p. 32.

[30] D. L. Parnas, "Software aging," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 279–287.

[31] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proceedings of the ACM Symposium on Software Visualization*. St. Louis, Missouri: ACM Press, 2005, pp. 67–75.

[32] E. Pulvermueller, A. Speck, J. Coplien, M. D'Hondt, and W. D. Meuter, "Feature interaction in composed systems," in *ECOOP '01: Proceedings of the Workshops on Object-Oriented Technology*. London, UK: Springer-Verlag, 2002, pp. 86–97.

[33] QDox, "Qdox website," accessed 15 - June - 2007. [Online]. Available: http://qdox.codehaus.org/

[34] F. V. Rysselberghe and S. Demeyer, "Studying software evolution information by

visualizing the change history," in *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 328–337.

[35] SQuirreLSQL, "Squirrelsql website," accessed 15 - June - 2007. [Online]. Available: http://squirrel-sql.sourceforge.net/

[36] M. A. Storey, D. Čubranić, and D. M. German, "On the Use of Visualization to Support Awareness of Human Activities in Software Development:A Survey and a Framework," in *Proceedings of the 2nd ACM Symposium on Software Visualization*, 2005, pp. 193–202.

[37] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "Shrimp views: an interactive environment for information visualization and navigation," in *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM Press, 2002, pp. 520–521.

[38] B. B. T. System, "Bugzilla bug tracking system website," accessed 15 - June - 2007. [Online]. Available: http://www.bugzilla.org/

[39] C. M. B. Taylor and M. Munro, "Revision towers," *vissoft*, vol. 00, p. 43, 2002.

[40] S. A. Tonu, A. Ashkan, and L. Tahvildari, "Evaluating architectural stability using a metric-based approach," in *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 261–270.

[41] Q. Tu, "On Navigation and Analysis of Software Architecture Evolution," Master's thesis, University of Waterloo, 2002.

[42] L. Voinea, A. Telea, and J. J. van Wijk, "Cvsscan: visualization of code evolution," in *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*. New York, NY, USA: ACM Press, 2005, pp. 47–56.

[43] L. Voinea and A. Telea, "Mining software repositories with cvsgrab," in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. New York, NY, USA: ACM Press, 2006, pp. 167–168.

[44] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt, "Evolution spectrographs: Visualizing punctuated change in software evolution," in *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop on (IWPSE'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 57–66.

[45] X. Wu, "Visualization of Version Control Information," Master's thesis, University of Victoria, 2003.

[46] X. Wu, A. Murray, M.-A. D. Storey, and R. Lintern, "A reverse engineering approach

to support software maintenance: Version control knowledge extraction." in *WCRE*, 2004, pp. 90–99.

[47] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM Press, 2005, pp. 54–65.

[48] T. Zimmermann and P. Weissgerber, "Preprocessing CVS Data for Fine-grained Analysis," in *Proceedings of the First International Workshop on Mining Software Repositories*, May 2004, pp. 2–6.

[49] L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005, member-Michael W. Godfrey.